R: A Language and Environment for Statistical Computing

Reference Index

The R Development Core Team

Version 2.8.1 (2008-12-22)

Copyright (©) 1999–2003 R Foundation for Statistical Computing.
Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.
Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.
Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the R Development Core Team.

R is free software and comes with ABSOLUTELY NO WARRANTY. You are welcome to redistribute it under the terms of the GNU General Public License. For more information about these matters, see http://www.gnu.org/copyleft/gpl.html.

ISBN 3-900051-07-0

Contents

1 The `base` package **1**

base-package . . . 1
.Device . . . 1
.Machine . . . 2
.Platform . . . 4
.Script . . . 5
abbreviate . . . 6
agrep . . . 7
all . . . 9
all.equal . . . 10
all.names . . . 11
any . . . 12
aperm . . . 13
append . . . 14
apply . . . 15
args . . . 16
Arithmetic . . . 17
array . . . 19
as.data.frame . . . 20
as.environment . . . 21
as.function . . . 22
as.POSIX* . . . 23
AsIs . . . 25
assign . . . 26
assignOps . . . 27
attach . . . 28
attr . . . 30
attributes . . . 31
autoload . . . 32
backsolve . . . 33
base-deprecated . . . 34
basename . . . 35
Bessel . . . 36
bindenv . . . 38
body . . . 39
bquote . . . 40
browser . . . 41
builtins . . . 42
by . . . 43
c . . . 44
call . . . 45

callCC . . . 46
capabilities . . . 47
cat . . . 48
cbind . . . 50
char.expand . . . 52
character . . . 53
charmatch . . . 55
chartr . . . 56
chol . . . 57
chol2inv . . . 59
class . . . 60
col . . . 62
Colon . . . 63
colSums . . . 64
commandArgs . . . 65
comment . . . 66
Comparison . . . 67
complex . . . 69
conditions . . . 70
conflicts . . . 73
connections . . . 74
Constants . . . 80
contributors . . . 81
Control . . . 82
copyright . . . 83
crossprod . . . 84
Cstack_info . . . 85
cumsum . . . 85
cut . . . 86
cut.POSIXt . . . 88
data.class . . . 89
data.frame . . . 90
data.matrix . . . 92
date . . . 94
Dates . . . 94
DateTimeClasses . . . 95
dcf . . . 98
debug . . . 99
Defunct . . . 100
delayedAssign . . . 101
deparse . . . 102
deparseOpts . . . 103
Deprecated . . . 105
det . . . 105
detach . . . 106
diag . . . 108
diff . . . 109
difftime . . . 110
dim . . . 112
dimnames . . . 113
do.call . . . 114
double . . . 115

dput . . . 117
drop . . . 118
dump . . . 119
duplicated . . . 121
dyn.load . . . 122
eapply . . . 125
eigen . . . 126
encodeString . . . 127
Encoding . . . 129
environment . . . 130
EnvVar . . . 132
eval . . . 134
exists . . . 136
expand.grid . . . 138
expression . . . 139
Extract . . . 140
Extract.data.frame . . . 144
Extract.factor . . . 146
Extremes . . . 148
factor . . . 150
file.access . . . 152
file.choose . . . 154
file.info . . . 154
file.path . . . 155
file.show . . . 156
files . . . 157
findInterval . . . 159
force . . . 160
Foreign . . . 161
formals . . . 164
format . . . 165
format.Date . . . 168
format.info . . . 170
format.pval . . . 171
formatC . . . 172
formatDL . . . 174
function . . . 176
funprog . . . 177
gc . . . 179
gc.time . . . 180
gctorture . . . 181
get . . . 182
getDLLRegisteredRoutines . . . 183
getLoadedDLLs . . . 185
getNativeSymbolInfo . . . 186
getNumCConverters . . . 188
getpid . . . 189
gettext . . . 190
getwd . . . 191
gl . . . 192
grep . . . 193
groupGeneric . . . 196

gzcon . . . 198
hexmode . . . 200
Hyperbolic . . . 200
iconv . . . 201
icuSetCollate . . . 203
identical . . . 204
identity . . . 206
ifelse . . . 206
integer . . . 207
interaction . . . 208
interactive . . . 209
Internal . . . 210
InternalMethods . . . 210
invisible . . . 211
is.finite . . . 211
is.function . . . 213
is.language . . . 214
is.object . . . 214
is.R . . . 215
is.recursive . . . 216
is.single . . . 217
is.unsorted . . . 217
isS4 . . . 218
isSymmetric . . . 219
jitter . . . 220
kappa . . . 221
kronecker . . . 222
l10n_info . . . 224
labels . . . 224
lapply . . . 225
Last.value . . . 227
length . . . 227
levels . . . 228
libPaths . . . 230
library . . . 231
library.dynam . . . 235
license . . . 237
list . . . 237
list.files . . . 239
load . . . 241
localeconv . . . 242
locales . . . 243
log . . . 244
Logic . . . 246
logical . . . 248
lower.tri . . . 249
ls . . . 250
make.names . . . 251
make.unique . . . 252
manglePackageName . . . 253
mapply . . . 254
margin.table . . . 255

mat.or.vec . . . 256
match . . . 257
match.arg . . . 258
match.call . . . 259
match.fun . . . 260
MathFun . . . 261
matmult . . . 262
matrix . . . 263
maxCol . . . 265
mean . . . 266
Memory . . . 267
Memory-limits . . . 269
memory.profile . . . 269
merge . . . 270
message . . . 272
missing . . . 273
mode . . . 274
NA . . . 276
name . . . 277
names . . . 278
nargs . . . 280
nchar . . . 281
nlevels . . . 282
noquote . . . 283
NotYet . . . 284
nrow . . . 285
ns-dblcolon . . . 286
ns-hooks . . . 287
ns-load . . . 288
ns-topenv . . . 289
NULL . . . 290
numeric . . . 290
NumericConstants . . . 292
numeric_version . . . 293
octmode . . . 294
on.exit . . . 295
Ops.Date . . . 296
options . . . 297
order . . . 303
outer . . . 305
Paren . . . 306
parse . . . 307
paste . . . 309
path.expand . . . 310
pmatch . . . 310
polyroot . . . 312
pos.to.env . . . 313
pretty . . . 313
Primitive . . . 315
print . . . 315
print.data.frame . . . 317
print.default . . . 318

prmatrix . . . 320
proc.time . . . 321
prod . . . 322
prop.table . . . 323
pushBack . . . 324
qr . . . 325
QR.Auxiliaries . . . 327
quit . . . 328
Quotes . . . 330
R.Version . . . 331
Random . . . 332
Random.user . . . 336
range . . . 338
rank . . . 339
rapply . . . 340
raw . . . 341
rawConnection . . . 343
rawConversion . . . 344
RdUtils . . . 345
readBin . . . 346
readChar . . . 349
readline . . . 350
readLines . . . 351
real . . . 352
Recall . . . 353
reg.finalizer . . . 354
regex . . . 355
remove . . . 359
rep . . . 360
replace . . . 362
Reserved . . . 362
rev . . . 363
Rhome . . . 364
rle . . . 364
Round . . . 365
round.POSIXt . . . 367
row . . . 368
row.names . . . 369
row/colnames . . . 370
rowsum . . . 371
sample . . . 372
save . . . 373
scale . . . 376
scan . . . 377
search . . . 380
seek . . . 381
seq . . . 382
seq.Date . . . 384
seq.POSIXt . . . 385
sequence . . . 386
sets . . . 387
setTimeLimit . . . 388

showConnections . . . 389
shQuote . . . 390
sign . . . 392
Signals . . . 392
sink . . . 393
slice.index . . . 395
slotOp . . . 395
socketSelect . . . 396
solve . . . 397
sort . . . 398
source . . . 400
Special . . . 402
split . . . 405
sprintf . . . 407
sQuote . . . 410
srcfile . . . 412
Startup . . . 414
stop . . . 416
stopifnot . . . 418
strptime . . . 419
strsplit . . . 423
strtrim . . . 425
structure . . . 426
strwrap . . . 427
subset . . . 428
substitute . . . 429
substr . . . 431
sum . . . 432
summary . . . 433
svd . . . 435
sweep . . . 436
switch . . . 437
Syntax . . . 438
Sys.getenv . . . 439
Sys.glob . . . 440
Sys.info . . . 441
sys.parent . . . 442
Sys.setenv . . . 445
Sys.sleep . . . 446
sys.source . . . 447
Sys.time . . . 448
Sys.which . . . 449
system . . . 449
system.file . . . 451
system.time . . . 452
t . . . 453
table . . . 454
tabulate . . . 456
tapply . . . 457
taskCallback . . . 458
taskCallbackManager . . . 460
taskCallbackNames . . . 462

tempfile . . . 463
textConnection . . . 464
tilde . . . 466
timezones . . . 466
toString . . . 468
trace . . . 469
traceback . . . 472
tracemem . . . 473
transform . . . 475
Trig . . . 476
try . . . 477
typeof . . . 478
unique . . . 479
unlink . . . 481
unlist . . . 482
unname . . . 483
UseMethod . . . 484
UserHooks . . . 486
utf8Conversion . . . 487
vector . . . 488
warning . . . 490
warnings . . . 491
weekdays . . . 492
which . . . 493
which.min . . . 494
with . . . 495
write . . . 497
writeLines . . . 498
xtfrm . . . 499
zpackages . . . 499
zutils . . . 500

2 The `datasets` package **501**
datasets-package . . . 501
ability.cov . . . 501
airmiles . . . 502
AirPassengers . . . 503
airquality . . . 504
anscombe . . . 505
attenu . . . 506
attitude . . . 507
austres . . . 508
beavers . . . 508
BJsales . . . 509
BOD . . . 510
cars . . . 511
ChickWeight . . . 512
chickwts . . . 513
CO2 . . . 514
co2 . . . 515
crimtab . . . 515
discoveries . . . 517
DNase . . . 518

esoph . . . 519
euro . . . 520
eurodist . . . 521
EuStockMarkets . . . 521
faithful . . . 522
Formaldehyde . . . 523
freeny . . . 524
HairEyeColor . . . 525
Harman23.cor . . . 526
Harman74.cor . . . 526
Indometh . . . 527
infert . . . 528
InsectSprays . . . 529
iris . . . 529
islands . . . 531
JohnsonJohnson . . . 531
LakeHuron . . . 532
lh . . . 532
LifeCycleSavings . . . 533
Loblolly . . . 534
longley . . . 534
lynx . . . 535
morley . . . 536
mtcars . . . 537
nhtemp . . . 537
Nile . . . 538
nottem . . . 539
occupationalStatus . . . 540
Orange . . . 541
OrchardSprays . . . 542
PlantGrowth . . . 543
precip . . . 543
presidents . . . 544
pressure . . . 545
Puromycin . . . 545
quakes . . . 547
randu . . . 547
rivers . . . 548
rock . . . 549
sleep . . . 549
stackloss . . . 550
state . . . 551
sunspot.month . . . 552
sunspot.year . . . 553
sunspots . . . 553
swiss . . . 554
Theoph . . . 555
Titanic . . . 556
ToothGrowth . . . 557
treering . . . 558
trees . . . 559
UCBAdmissions . . . 560

UKDriverDeaths . . . 561
UKgas . . . 562
UKLungDeaths . . . 563
USAccDeaths . . . 563
USArrests . . . 564
USJudgeRatings . . . 564
USPersonalExpenditure . . . 565
uspop . . . 566
VADeaths . . . 566
volcano . . . 567
warpbreaks . . . 568
women . . . 569
WorldPhones . . . 569
WWWusage . . . 570

3 The `grDevices` package **573**
grDevices-package . . . 573
as.graphicsAnnot . . . 573
boxplot.stats . . . 574
cairo . . . 576
check.options . . . 577
chull . . . 578
cm . . . 579
col2rgb . . . 580
colorRamp . . . 581
colors . . . 583
contourLines . . . 584
convertColor . . . 585
dev.interactive . . . 587
dev.size . . . 588
dev.xxx . . . 588
dev2 . . . 590
dev2bitmap . . . 592
devAskNewPage . . . 594
Devices . . . 594
embedFonts . . . 595
extendrange . . . 596
getGraphicsEvent . . . 597
gray . . . 599
gray.colors . . . 600
hcl . . . 601
Hershey . . . 603
hsv . . . 606
Japanese . . . 607
make.rgb . . . 608
n2mfrow . . . 609
nclass . . . 610
palette . . . 611
Palettes . . . 612
pdf . . . 613
pdf.options . . . 617
pictex . . . 618
plotmath . . . 619

png . . . 623
postscript . . . 627
postscriptFonts . . . 632
ps.options . . . 635
quartz . . . 636
quartzFonts . . . 638
recordGraphics . . . 639
recordPlot . . . 640
rgb . . . 641
rgb2hsv . . . 642
savePlot . . . 644
trans3d . . . 645
Type1Font . . . 645
x11 . . . 647
X11Fonts . . . 650
xfig . . . 651
xy.coords . . . 653
xyTable . . . 655
xyz.coords . . . 656

4 The `graphics` package **659**
graphics-package . . . 659
abline . . . 659
arrows . . . 661
assocplot . . . 662
Axis . . . 664
axis . . . 665
axis.POSIXct . . . 667
axTicks . . . 669
barplot . . . 670
box . . . 673
boxplot . . . 674
bxp . . . 677
cdplot . . . 680
clip . . . 682
contour . . . 683
convertXY . . . 685
coplot . . . 686
curve . . . 689
dotchart . . . 690
filled.contour . . . 691
fourfoldplot . . . 694
frame . . . 695
grid . . . 696
hist . . . 697
hist.POSIXt . . . 700
identify . . . 702
image . . . 704
layout . . . 706
legend . . . 708
lines . . . 712
locator . . . 714
matplot . . . 715

mosaicplot . . . 717
mtext . . . 720
pairs . . . 722
panel.smooth . . . 724
par . . . 725
persp . . . 732
pie . . . 736
plot . . . 737
plot.data.frame . . . 739
plot.default . . . 740
plot.design . . . 742
plot.factor . . . 744
plot.formula . . . 744
plot.histogram . . . 746
plot.table . . . 747
plot.window . . . 748
plot.xy . . . 749
points . . . 750
polygon . . . 753
rect . . . 755
rug . . . 757
screen . . . 758
segments . . . 760
spineplot . . . 761
stars . . . 763
stem . . . 766
stripchart . . . 767
strwidth . . . 769
sunflowerplot . . . 770
symbols . . . 772
text . . . 775
title . . . 777
units . . . 778
xspline . . . 779

5 The `grid` package **783**
grid-package . . . 783
absolute.size . . . 784
arrow . . . 785
convertNative . . . 785
dataViewport . . . 786
drawDetails . . . 787
editDetails . . . 788
gEdit . . . 789
getNames . . . 790
gpar . . . 790
gPath . . . 792
Grid . . . 793
Grid Viewports . . . 794
grid.add . . . 797
grid.arrows . . . 798
grid.circle . . . 801
grid.clip . . . 802

grid.collection . . . 803
grid.convert . . . 804
grid.copy . . . 806
grid.curve . . . 807
grid.display.list . . . 809
grid.draw . . . 810
grid.edit . . . 811
grid.frame . . . 812
grid.get . . . 813
grid.grab . . . 814
grid.grill . . . 815
grid.grob . . . 816
grid.layout . . . 818
grid.lines . . . 819
grid.locator . . . 821
grid.ls . . . 822
grid.move.to . . . 824
grid.newpage . . . 826
grid.null . . . 826
grid.pack . . . 827
grid.place . . . 829
grid.plot.and.legend . . . 830
grid.points . . . 830
grid.polygon . . . 831
grid.pretty . . . 833
grid.prompt . . . 833
grid.record . . . 834
grid.rect . . . 835
grid.refresh . . . 836
grid.remove . . . 836
grid.segments . . . 837
grid.set . . . 839
grid.show.layout . . . 840
grid.show.viewport . . . 841
grid.text . . . 842
grid.xaxis . . . 844
grid.xspline . . . 845
grid.yaxis . . . 848
grobName . . . 849
grobWidth . . . 849
grobX . . . 850
plotViewport . . . 851
pop.viewport . . . 851
push.viewport . . . 852
Querying the Viewport Tree . . . 853
stringWidth . . . 854
unit . . . 854
unit.c . . . 856
unit.length . . . 857
unit.pmin . . . 858
unit.rep . . . 858
validDetails . . . 859

vpPath . . . 860
widthDetails . . . 861
Working with Viewports . . . 861
xDetails . . . 864

6 The `methods` package **865**
methods-package . . . 865
.BasicFunsList . . . 866
as . . . 866
BasicClasses . . . 870
callGeneric . . . 871
callNextMethod . . . 873
canCoerce . . . 875
cbind2 . . . 875
Classes . . . 877
classRepresentation-class . . . 880
Documentation . . . 881
dotsMethods . . . 883
environment-class . . . 885
findClass . . . 886
findMethods . . . 888
fixPre1.8 . . . 889
genericFunction-class . . . 890
GenericFunctions . . . 891
getClass . . . 895
getMethod . . . 896
getPackageName . . . 898
hasArg . . . 899
implicitGeneric . . . 900
initialize-methods . . . 902
is . . . 903
isSealedMethod . . . 908
language-class . . . 909
LinearMethodsList-class . . . 910
makeClassRepresentation . . . 911
method.skeleton . . . 912
MethodDefinition-class . . . 913
Methods . . . 914
MethodsList-class . . . 918
MethodWithNext-class . . . 919
new . . . 920
ObjectsWithPackage-class . . . 922
promptClass . . . 923
promptMethods . . . 924
representation . . . 925
S3Part . . . 927
S4groupGeneric . . . 930
SClassExtension-class . . . 932
setClass . . . 933
setClassUnion . . . 936
setGeneric . . . 937
setMethod . . . 942
setOldClass . . . 946

show . . . 950
showMethods . . . 951
signature-class . . . 953
slot . . . 954
StructureClasses . . . 956
TraceClasses . . . 957
validObject . . . 958

7 The `stats` package **961**
stats-package . . . 961
.checkMFClasses . . . 961
acf . . . 962
acf2AR . . . 964
add1 . . . 965
addmargins . . . 967
aggregate . . . 969
AIC . . . 971
alias . . . 972
anova . . . 973
anova.glm . . . 974
anova.lm . . . 976
anova.mlm . . . 977
ansari.test . . . 979
aov . . . 982
approxfun . . . 983
ar . . . 985
ar.ols . . . 988
arima . . . 990
arima.sim . . . 993
arima0 . . . 995
ARMAacf . . . 998
ARMAtoMA . . . 999
as.hclust . . . 1000
asOneSidedFormula . . . 1001
ave . . . 1001
bandwidth . . . 1002
bartlett.test . . . 1004
Beta . . . 1005
binom.test . . . 1007
Binomial . . . 1008
biplot . . . 1010
biplot.princomp . . . 1011
birthday . . . 1013
Box.test . . . 1014
C . . . 1015
cancor . . . 1016
case/variable.names . . . 1017
Cauchy . . . 1018
chisq.test . . . 1019
Chisquare . . . 1022
clearNames . . . 1024
cmdscale . . . 1025
coef . . . 1026

complete.cases . . . 1027
confint . . . 1028
constrOptim . . . 1029
contrast . . . 1031
contrasts . . . 1032
convolve . . . 1033
cophenetic . . . 1035
cor . . . 1036
cor.test . . . 1038
cov.wt . . . 1041
cpgram . . . 1042
cutree . . . 1043
decompose . . . 1044
delete.response . . . 1045
dendrapply . . . 1046
dendrogram . . . 1048
density . . . 1051
deriv . . . 1054
deviance . . . 1057
df.residual . . . 1058
diffinv . . . 1058
dist . . . 1059
dummy.coef . . . 1062
ecdf . . . 1063
eff.aovlist . . . 1066
effects . . . 1067
embed . . . 1068
expand.model.frame . . . 1069
Exponential . . . 1070
extractAIC . . . 1071
factanal . . . 1072
factor.scope . . . 1075
family . . . 1076
FDist . . . 1080
fft . . . 1081
filter . . . 1082
fisher.test . . . 1084
fitted . . . 1086
fivenum . . . 1087
fligner.test . . . 1088
formula . . . 1090
formula.nls . . . 1092
friedman.test . . . 1092
ftable . . . 1094
ftable.formula . . . 1096
GammaDist . . . 1097
Geometric . . . 1100
getInitial . . . 1101
glm . . . 1102
glm.control . . . 1106
glm.summaries . . . 1107
hclust . . . 1108

heatmap . . . 1111
HoltWinters . . . 1114
Hypergeometric . . . 1117
identify.hclust . . . 1118
influence.measures . . . 1119
integrate . . . 1122
interaction.plot . . . 1124
IQR . . . 1126
is.empty.model . . . 1127
isoreg . . . 1128
KalmanLike . . . 1129
kernapply . . . 1131
kernel . . . 1132
kmeans . . . 1133
kruskal.test . . . 1135
ks.test . . . 1137
ksmooth . . . 1139
lag . . . 1140
lag.plot . . . 1141
line . . . 1142
lm . . . 1143
lm.fit . . . 1146
lm.influence . . . 1148
lm.summaries . . . 1149
loadings . . . 1151
loess . . . 1152
loess.control . . . 1154
Logistic . . . 1155
logLik . . . 1156
loglin . . . 1157
Lognormal . . . 1159
lowess . . . 1161
ls.diag . . . 1162
ls.print . . . 1163
lsfit . . . 1164
mad . . . 1165
mahalanobis . . . 1166
make.link . . . 1167
makepredictcall . . . 1168
manova . . . 1169
mantelhaen.test . . . 1170
mauchly.test . . . 1172
mcnemar.test . . . 1174
median . . . 1175
medpolish . . . 1176
model.extract . . . 1178
model.frame . . . 1179
model.matrix . . . 1181
model.tables . . . 1182
monthplot . . . 1184
mood.test . . . 1186
Multinomial . . . 1187

na.action . . . 1188
na.contiguous . . . 1189
na.fail . . . 1190
naprint . . . 1191
naresid . . . 1191
NegBinomial . . . 1192
nextn . . . 1194
nlm . . . 1195
nlminb . . . 1197
nls . . . 1199
nls.control . . . 1204
NLSstAsymptotic . . . 1206
NLSstClosestX . . . 1206
NLSstLfAsymptote . . . 1207
NLSstRtAsymptote . . . 1208
Normal . . . 1209
numericDeriv . . . 1211
offset . . . 1212
oneway.test . . . 1212
optim . . . 1214
optimize . . . 1219
order.dendrogram . . . 1221
p.adjust . . . 1222
pairwise.prop.test . . . 1224
pairwise.t.test . . . 1224
pairwise.table . . . 1225
pairwise.wilcox.test . . . 1226
plot.acf . . . 1227
plot.density . . . 1228
plot.HoltWinters . . . 1229
plot.isoreg . . . 1230
plot.lm . . . 1231
plot.ppr . . . 1234
plot.profile.nls . . . 1235
plot.spec . . . 1236
plot.stepfun . . . 1237
plot.ts . . . 1239
Poisson . . . 1240
poly . . . 1242
power . . . 1243
power.anova.test . . . 1244
power.prop.test . . . 1245
power.t.test . . . 1247
PP.test . . . 1248
ppoints . . . 1249
ppr . . . 1250
prcomp . . . 1253
predict . . . 1255
predict.Arima . . . 1256
predict.glm . . . 1257
predict.HoltWinters . . . 1259
predict.lm . . . 1260

predict.loess . . . 1262
predict.nls . . . 1264
predict.smooth.spline . . . 1265
preplot . . . 1267
princomp . . . 1267
print.power.htest . . . 1270
print.ts . . . 1270
printCoefmat . . . 1271
profile . . . 1273
profile.nls . . . 1273
proj . . . 1274
prop.test . . . 1276
prop.trend.test . . . 1278
qqnorm . . . 1279
quade.test . . . 1280
quantile . . . 1282
r2dtable . . . 1284
read.ftable . . . 1285
rect.hclust . . . 1287
relevel . . . 1288
reorder.dendrogram . . . 1288
reorder.factor . . . 1289
replications . . . 1291
reshape . . . 1292
residuals . . . 1295
runmed . . . 1295
scatter.smooth . . . 1297
screeplot . . . 1299
sd . . . 1300
se.contrast . . . 1300
selfStart . . . 1302
setNames . . . 1304
shapiro.test . . . 1305
SignRank . . . 1306
simulate . . . 1307
smooth . . . 1308
smooth.spline . . . 1310
smoothEnds . . . 1313
sortedXyData . . . 1315
spec.ar . . . 1315
spec.pgram . . . 1317
spec.taper . . . 1319
spectrum . . . 1320
splinefun . . . 1321
SSasymp . . . 1324
SSasympOff . . . 1325
SSasympOrig . . . 1326
SSbiexp . . . 1327
SSD . . . 1328
SSfol . . . 1329
SSfpl . . . 1330
SSgompertz . . . 1331

SSlogis . . . 1332
SSmicmen . . . 1333
SSweibull . . . 1334
start . . . 1335
stat.anova . . . 1336
stats-deprecated . . . 1337
step . . . 1337
stepfun . . . 1339
stl . . . 1341
stlmethods . . . 1343
StructTS . . . 1344
summary.aov . . . 1346
summary.glm . . . 1348
summary.lm . . . 1350
summary.manova . . . 1352
summary.nls . . . 1353
summary.princomp . . . 1355
supsmu . . . 1356
symnum . . . 1357
t.test . . . 1359
TDist . . . 1361
termplot . . . 1363
terms . . . 1365
terms.formula . . . 1366
terms.object . . . 1367
time . . . 1368
toeplitz . . . 1369
ts . . . 1370
ts-methods . . . 1371
ts.plot . . . 1372
ts.union . . . 1373
tsdiag . . . 1374
tsp . . . 1375
tsSmooth . . . 1375
Tukey . . . 1376
TukeyHSD . . . 1377
Uniform . . . 1379
uniroot . . . 1380
update . . . 1381
update.formula . . . 1382
var.test . . . 1383
varimax . . . 1385
vcov . . . 1386
Weibull . . . 1386
weighted.mean . . . 1388
weighted.residuals . . . 1388
wilcox.test . . . 1389
Wilcoxon . . . 1392
window . . . 1394
xtabs . . . 1396

8 The `tools` package **1399**
tools-package . . . 1399
buildVignettes . . . 1399
charsets . . . 1400
checkFF . . . 1401
checkMD5sums . . . 1402
checkTnF . . . 1403
checkVignettes . . . 1404
codoc . . . 1405
delimMatch . . . 1406
encoded_text_to_latex . . . 1407
fileutils . . . 1408
getDepList . . . 1410
installFoundDepends . . . 1411
makeLazyLoading . . . 1412
md5sum . . . 1413
package.dependencies . . . 1413
QC . . . 1414
Rdindex . . . 1415
Rdutils . . . 1416
read.00Index . . . 1417
readNEWS . . . 1418
showNonASCII . . . 1419
texi2dvi . . . 1420
tools-deprecated . . . 1420
undoc . . . 1421
vignetteDepends . . . 1422
write_PACKAGES . . . 1423
xgettext . . . 1424

9 The `utils` package **1427**
utils-package . . . 1427
alarm . . . 1427
apropos . . . 1428
BATCH . . . 1429
browseEnv . . . 1430
browseURL . . . 1432
browseVignettes . . . 1433
bug.report . . . 1434
capture.output . . . 1436
chooseCRANmirror . . . 1437
citation . . . 1438
citEntry . . . 1439
close.socket . . . 1441
combn . . . 1441
compareVersion . . . 1443
COMPILE . . . 1443
count.fields . . . 1444
data . . . 1445
dataentry . . . 1447
debugger . . . 1449
demo . . . 1451
download.file . . . 1452

edit . . . 1454
edit.data.frame . . . 1455
example . . . 1457
file.edit . . . 1459
file_test . . . 1459
fix . . . 1460
flush.console . . . 1461
format . . . 1461
getAnywhere . . . 1462
getFromNamespace . . . 1463
getS3method . . . 1465
glob2rx . . . 1465
head . . . 1466
help . . . 1468
help.request . . . 1471
help.search . . . 1473
help.start . . . 1475
index.search . . . 1476
INSTALL . . . 1477
installed.packages . . . 1478
LINK . . . 1479
localeToCharset . . . 1480
ls.str . . . 1481
make.packages.html . . . 1482
make.socket . . . 1483
memory.size . . . 1484
menu . . . 1485
methods . . . 1486
mirrorAdmin . . . 1487
modifyList . . . 1488
normalizePath . . . 1488
nsl . . . 1489
object.size . . . 1490
package.skeleton . . . 1491
packageDescription . . . 1492
packageStatus . . . 1493
page . . . 1495
person . . . 1495
PkgUtils . . . 1496
prompt . . . 1497
promptData . . . 1499
promptPackage . . . 1500
rcompgen . . . 1501
read.DIF . . . 1506
read.fortran . . . 1508
read.fwf . . . 1509
read.socket . . . 1510
read.table . . . 1511
recover . . . 1515
relist . . . 1517
REMOVE . . . 1519
remove.packages . . . 1520

RHOME 1520
roman 1521
Rprof 1521
Rprofmem 1523
Rscript 1524
RShowDoc 1525
RSiteSearch 1526
Rtangle 1527
RweaveLatex 1528
savehistory 1530
select.list 1532
sessionInfo 1533
setRepositories 1533
SHLIB 1534
stack 1535
str 1536
summaryRprof 1539
Sweave 1540
SweaveSyntConv 1542
toLatex 1543
txtProgressBar 1544
type.convert 1545
update.packages 1546
url.show 1550
URLencode 1551
utils-deprecated 1552
View 1552
vignette 1553
withVisible 1554
write.table 1555
zip.file.extract 1558

Index **1559**

Chapter 1

The `base` package

`base-package` *The R Base Package*

Description

Base R functions

Details

This package contains the basic functions which let R function as a language: arithmetic, input/output, basic programming support, etc. Its contents are available through inheritance from any environment.

For a complete list of functions, use `library(help="base")`.

`.Device` *Lists of Open/Active Graphics Devices*

Description

A pairlist of the names of open graphics devices is stored in `.Devices`. The name of the active device (see `dev.cur`) is stored in `.Device`. Both are symbols and so appear in the base name space.

Value

`.Device` is a length-one character vector.

`.Devices` is a pairlist of length-one character vectors. The first entry is always `"null device"`, and there are as many entries as the maximal number of graphics devices which have been simultaneously active. If a device has been removed, its entry will be `""` until the device number is reused.

`.Machine` *Numerical Characteristics of the Machine*

Description

`.Machine` is a variable holding information on the numerical characteristics of the machine R is running on, such as the largest double or integer and the machine's precision.

Usage

```
.Machine
```

Details

The algorithm is based on Cody's (1988) subroutine MACHAR. As almost all current inplements of R use 32-bit integers and IEC 60059 floating-point (double precision) arithmetic, most of these values are the same for almost all R builds.

Note that on most platforms smaller positive values than `.Machine$double.xmin` can occur. On a typical R platform the smallest positive double is about `5e-324`.

Value

A list with components (for simplicity, the prefix 'double' is omitted in the explanations)

`double.eps` the smallest positive floating-point number `x` such that `1 + x != 1`. It equals `base^ulp.digits` if either `base` is 2 or `rounding` is 0; otherwise, it is `(base^ulp.digits) / 2`. Normally `2.220446e-16`.

`double.neg.eps` a small positive floating-point number `x` such that `1 - x != 1`. It equals `base^neg.ulp.digits` if `base` is 2 or `round` is 0; otherwise, it is `(base^neg.ulp.digits) / 2`. Normally `1.110223e-16`. As `neg.ulp.digits` is bounded below by `-(digits + 3)`, `neg.eps` may not be the smallest number that can alter 1 by subtraction.

`double.xmin` the smallest non-vanishing normalized floating-point power of the radix, i.e., `base^min.exp`. Normally `2.225074e-308`.

`double.xmax` the largest finite floating-point number. Typically, it is equal to `(1 - neg.eps) * base^max.exp`, but on some machines it is only the second, or perhaps third, largest number, being too small by 1 or 2 units in the last digit of the significand. Normally `1.797693e+308`.

`double.base` the radix for the floating-point representation: normally `2`.

`double.digits` the number of base digits in the floating-point significand: normally `53`.

`double.rounding` the rounding action.
0 if floating-point addition chops;
1 if floating-point addition rounds, but not in the IEEE style;
2 if floating-point addition rounds in the IEEE style;
3 if floating-point addition chops, and there is partial underflow;
4 if floating-point addition rounds, but not in the IEEE style, and there is partial underflow;

5 if floating-point addition rounds in the IEEE style, and there is partial underflow. Normally 5.

`double.guard` the number of guard digits for multiplication with truncating arithmetic. It is 1 if floating-point arithmetic truncates and more than `digits` base `base` digits participate in the post-normalization shift of the floating-point significand in multiplication, and 0 otherwise.

`double.ulp.digits`
the largest negative integer `i` such that `1 + base^i != 1`, except that it is bounded below by `-(digits + 3)`. Normally `-52`.

`double.neg.ulp.digits`
the largest negative integer `i` such that `1 - base^i != 1`, except that it is bounded below by `-(digits + 3)`. Normally `-53`.

`double.exponent`
the number of bits (decimal places if `base` is 10) reserved for the representation of the exponent (including the bias or sign) of a floating-point number. Normally `11`.

`double.min.exp`
the largest in magnitude negative integer `i` such that `base ^ i` is positive and normalized. Normally `-1022`.

`double.max.exp`
the smallest positive power of `base` that overflows. Normally `1024`.

`integer.max` the largest integer which can be represented. Always `2147483647`.

`sizeof.long` the number of bytes in a C `long` type: `4` or `8` (most 64-bit systems, but not Windows).

`sizeof.longlong`
the number of bytes in a C `long long` type. Will be zero if there is no such type, otherwise usually `8`.

`sizeof.longdouble`
the number of bytes in a C `long double` type. Will be zero if there is no such type, otherwise possibly `12` (Windows, 32-bit Linux/Solaris) or `16` (64-bit Linux/Solaris, Intel Mac OS X).

`sizeof.pointer`
the number of bytes in a C `SEXP` type. Will be `4` on 32-bit builds and `8` on 64-bit builds of R.

References

Cody, W. J. (1988) MACHAR: A subroutine to dynamically determine machine parameters. *Transactions on Mathematical Software*, **14**, 4, 303–311.

See Also

`.Platform` for details of the platform.

Examples

```
.Machine
## or for a neat printout
noquote(unlist(format(.Machine)))
```

`.Platform` *Platform Specific Variables*

Description

`.Platform` is a list with some details of the platform under which R was built. This provides means to write OS-portable R code.

Usage

```
.Platform
```

Value

A list with at least the following components:

`OS.type`	character string, giving the **O**perating **S**ystem (family) of the computer. One of `"unix"` or `"windows"`.
`file.sep`	character string, giving the **file sep**arator used on your platform: `"/"` on both Unix-alikes *and* on Windows (but not on the now abandoned port to Classic MacOS).
`dynlib.ext`	character string, giving the file name **ext**ension of **dyn**amically loadable **lib**raries, e.g., `".dll"` on Windows and `".so"` or `".sl"` on Unix-alikes. (Note for Mac OS X users: these are shared objects as loaded by `dyn.load` and not dylibs: see `dyn.load`.)
`GUI`	character string, giving the type of GUI in use, or `"unknown"` if no GUI can be assumed. Possible values are for Unix-alikes the values given via the '`-g`' command-line flag (`"X11"`, `"Tk"`, `"GNOME"`), `"AQUA"` (running under `R.app` on MacOS X), `"Rgui"` and `"RTerm"` (Windows) and perhaps others under alternative front-ends or embedded R.
`endian`	character string, `"big"` or `"little"`, giving the endianness of the processor in use. This is relevant when it is necessary to know the order to read/write bytes of e.g. an integer or double from/to a connection: see `readBin`.
`pkgType`	character string, the preferred setting for `options("pkgType")`. Values `"source"`, `"mac.binary"` and `"win.binary"` are currently in use.
`path.sep`	character string, giving the **path sep**arator, used on your platform, e.g., `":"` on Unix-alikes and `";"` on Windows. Used to separate paths in variables such as `PATH` and `TEXINPUTS`.
`r_arch`	character string, possibly `""`. The name of the architecture-specific directories used in this build of R.

See Also

`R.version` and `Sys.info` give more details about the OS. In particular, `R.version$platform` is the canonical name of the platform under which R was compiled.

`.Machine` for details of the arithmetic used, and `system` for invoking platform-specific system commands.

Examples

```
## Note: this can be done in a system-independent way
## by file.info()$isdir
if(.Platform$OS.type == "unix") {
   system.test <- function(...) { system(paste("test", ...)) == 0 }
   dir.exists <- function(dir)
       sapply(dir, function(d)system.test("-d", d))
   dir.exists(c(R.home(), "/tmp", "~", "/NO"))# > T T T F
}
```

.Script *Scripting Language Interface*

Description

Run a script through its interpreter with given arguments.

Usage

```
.Script(interpreter, script, args, ...)
```

Arguments

`interpreter`	a character string naming the interpreter for the script.
`script`	a character string with the base file name of the script, which must be located in the '`interpreter`' subdirectory of '*R_SHARE_DIR*' (normally '*R_HOME*/share').
`args`	a character string giving the arguments to pass to the script.
`...`	further arguments to be passed to `system` when invoking the interpreter on the script.

Note

This function is for R internal use only.

Examples

```
## not useful on Windows, where the help is zipped.
.Script("perl", "massage-Examples.pl",
        paste("tools", system.file("R-ex", package = "tools")))
```

abbreviate *Abbreviate Strings*

Description

Abbreviate strings to at least `minlength` characters, such that they remain *unique* (if they were), unless `strict=TRUE`.

Usage

```
abbreviate(names.arg, minlength = 4, use.classes = TRUE,
           dot = FALSE, strict = FALSE, method = c("left.kept", "both.sides"))
```

Arguments

`names.arg`	a character vector of names to be abbreviated, or an object to be coerced to a character vector by `as.character`.
`minlength`	the minimum length of the abbreviations.
`use.classes`	logical (currently ignored by R).
`dot`	logical: should a dot (`"."`) be appended?
`strict`	logical: should `minlength` be observed strictly? Note that setting `strict=TRUE` may return *non*-unique strings.
`method`	a string specifying the method used with default `"left.kept"`, see 'Details' below.

Details

The algorithm (`method = "left.kept"`) used is similar to that of S. For a single string it works as follows. First all spaces at the beginning of the string are stripped. Then (if necessary) any other spaces are stripped. Next, lower case vowels are removed (starting at the right) followed by lower case consonants. Finally if the abbreviation is still longer than `minlength` upper case letters are stripped.

Characters are always stripped from the end of the word first. If an element of `names.arg` contains more than one word (words are separated by space) then at least one letter from each word will be retained.

Missing (`NA`) values are unaltered.

If `use.classes` is `FALSE` then the only distinction is to be between letters and space. This has NOT been implemented.

Value

A character vector containing abbreviations for the strings in its first argument. Duplicates in the original `names.arg` will be given identical abbreviations. If any non-duplicated elements have the same `minlength` abbreviations then, if `method = "both.sides"` the basic internal `abbreviate()` algorithm is applied to the characterwise *reversed* strings; if there are still duplicated abbreviations and if `strict=FALSE` as by default, `minlength` is incremented by one and new abbreviations are found for those elements only. This process is repeated until all unique elements of `names.arg` have unique abbreviations.

The character version of `names.arg` is attached to the returned value as a names argument: no other attributes are retained.

Warning

This is really only suitable for English, and does not work correctly with non-ASCII characters in multibyte locales. It will warn if used with non-ASCII characters.

See Also

`substr`.

Examples

```
x <- c("abcd", "efgh", "abce")
abbreviate(x, 2)
abbreviate(x, 2, strict=TRUE)# >> 1st and 3rd are == "ab"

(st.abb <- abbreviate(state.name, 2))
table(nchar(st.abb))# out of 50, 3 need 4 letters :
as <- abbreviate(state.name, 3, strict=TRUE)
as[which(as == "Mss")]

## method="both.sides" helps:  no 4-letters, and only 4 3-letters:
st.ab2 <- abbreviate(state.name, 2, method="both")
table(nchar(st.ab2))
## Compare the two methods:
cbind(st.abb, st.ab2)
```

---

agrep *Approximate String Matching (Fuzzy Matching)*

Description

Searches for approximate matches to `pattern` (the first argument) within the string `x` (the second argument) using the Levenshtein edit distance.

Usage

```
agrep(pattern, x, ignore.case = FALSE, value = FALSE,
      max.distance = 0.1, useBytes = FALSE)
```

Arguments

`pattern`	a non-empty character string to be matched (*not* a regular expression!). Coerced by `as.character` to a string if possible.
`x`	character vector where matches are sought. Coerced by `as.character` to a character vector if possible.
`ignore.case`	if `FALSE`, the pattern matching is *case sensitive* and if `TRUE`, case is ignored during matching.
`value`	if `FALSE`, a vector containing the (integer) indices of the matches determined is returned and if `TRUE`, a vector containing the matching elements themselves is returned.

`max.distance` Maximum distance allowed for a match. Expressed either as integer, or as a fraction of the *pattern* length (will be replaced by the smallest integer not less than the corresponding fraction of the pattern length), or a list with possible components

`all`: maximal (overall) distance

`insertions`: maximum number/fraction of insertions

`deletions`: maximum number/fraction of deletions

`substitutions`: maximum number/fraction of substitutions

If `all` is missing, it is set to 10%, the other components default to `all`. The component names can be abbreviated.

`useBytes` logical. in a multibyte locale, should the comparison be character-by-chracter (the default) or byte-by-byte.

Details

The Levenshtein edit distance is used as measure of approximateness: it is the total number of insertions, deletions and substitutions required to transform one string into another.

The function is a simple interface to the `apse` library developed by Jarkko Hietaniemi (also used in the Perl String::Approx module), modified to work with multibyte character sets. To save space it only supports the first 65536 characters of UTF-8 (where all the characters for human languages lie). Note that it can be quite slow in UTF-8, and `useBytes = TRUE` will be much faster.

Value

Either a vector giving the indices of the elements that yielded a match, or, if `value` is `TRUE`, the matched elements (after coercion, preserving names but no other attributes).

Author(s)

Original version by David Meyer, based on C code by Jarkko Hietaniemi.

See Also

`grep`

Examples

```
agrep("lasy", "1 lazy 2")
agrep("lasy", c(" 1 lazy 2", "1 lasy 2"), max = list(sub = 0))
agrep("laysy", c("1 lazy", "1", "1 LAZY"), max = 2)
agrep("laysy", c("1 lazy", "1", "1 LAZY"), max = 2, value = TRUE)
agrep("laysy", c("1 lazy", "1", "1 LAZY"), max = 2, ignore.case = TRUE)
```

`all` *Are All Values True?*

Description

Given a set of logical vectors, are all of the values true?

Usage

```
all(..., na.rm = FALSE)
```

Arguments

`...`	zero or more logical vectors. Other objects of zero length are ignored, and the rest are coerced to logical ignoring any class.
`na.rm`	logical. If true `NA` values are removed before the result is computed.

Details

This is a generic function: methods can be defined for it directly or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

Coercion of types other than integer (raw, double, complex, character, list) gives a warning as from R 2.7.0, as this was often unintentional.

Value

The value is a logical vector of length one.

Let `x` denote the concatenation of all the logical vectors in `...` (after coercion), after removing `NA`s if requested by `na.rm = TRUE`.

The value returned is `TRUE` if all of the values in `x` are `TRUE` (including if there are no values), and `FALSE` if at least one of the values in `x` is `FALSE`. Otherwise the value is `NA` (which can only occur if `na.rm = FALSE` and `...` contains no `FALSE` values and at least one `NA` value).

S4 methods

This is part of the S4 `Summary` group generic. Methods for it must use the signature `x, ..., na.rm`.

Note

That `all(logical(0))` is true is a useful convention: it ensures that

```
all(all(x), all(y)) == all(x,y)
```

even if `x` has length zero.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

any, the 'complement' of all, and stopifnot(*) which is an all(*) 'insurance'.

Examples

```
range(x <- sort(round(stats::rnorm(10) - 1.2, 1)))
if(all(x < 0)) cat("all x values are negative\n")

all(logical(0))  # true, as all zero of the elements are true.
```

---

all.equal *Test if Two Objects are (Nearly) Equal*

Description

all.equal(x,y) is a utility to compare R objects x and y testing 'near equality'. If they are different, comparison is still made to some extent, and a report of the differences is returned. Don't use all.equal directly in if expressions—either use isTRUE(all.equal(....)) or identical if appropriate.

Usage

```
all.equal(target, current, ...)

## S3 method for class 'numeric':
all.equal(target, current,
          tolerance = .Machine$double.eps ^ 0.5,
          scale = NULL, check.attributes = TRUE, ...)

attr.all.equal(target, current,
               check.attributes = TRUE, check.names = TRUE, ...)
```

Arguments

target	R object.
current	other R object, to be compared with target.
...	Further arguments for different methods, notably the following two, for numerical comparison:
tolerance	numeric ≥ 0. Differences smaller than tolerance are not considered.
scale	numeric scalar > 0 (or NULL). See 'Details'.
check.attributes	logical indicating if the attributes(.) of target and current should be compared as well.
check.names	logical indicating if the names(.) of target and current should be compared as well (and separately from the attributes).

Details

There are several methods available, most of which are dispatched by the default method, see `methods("all.equal")`. `all.equal.list` and `all.equal.language` provide comparison of recursive objects.

Numerical comparisons for `scale = NULL` (the default) are done by first computing the mean absolute difference of the two numerical vectors. If this is smaller than `tolerance` or not finite, absolute differences are used, otherwise relative differences scaled by the mean absolute difference.

If `scale` is positive, absolute comparisons are made after scaling (dividing) by `scale`.

For complex arguments, the modulus `Mod` of the difference is used: `all.equal.numeric` is called so arguments `tolerance` and `scale` are available.

`attr.all.equal` is used for comparing `attributes`, returning `NULL` or a `character` vector.

Value

Either `TRUE` (`NULL` for `attr.all.equal`) or a vector of `mode` `"character"` describing the differences between `target` and `current`.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer (for =).

See Also

`identical`, `isTRUE`, `==`, and `all` for exact equality testing.

Examples

```
all.equal(pi, 355/113)
# not precise enough (default tol) > relative error

d45 <- pi*(1/4 + 1:10)
stopifnot(
all.equal(tan(d45), rep(1,10)))        # TRUE, but
all      (tan(d45) == rep(1,10))       # FALSE, since not exactly
all.equal(tan(d45), rep(1,10), tol=0)  # to see difference
```

---

all.names *Find All Names in an Expression*

Description

Return a character vector containing all the names which occur in an expression or call.

Usage

```
all.names(expr, functions = TRUE, max.names = -1L, unique = FALSE)

all.vars(expr, functions = FALSE, max.names = -1L, unique = TRUE)
```

Arguments

`expr`	an expression or call from which the names are to be extracted.
`functions`	a logical value indicating whether function names should be included in the result.
`max.names`	the maximum number of names to be returned. `-1` indicates no limit (other than vector size limits).
`unique`	a logical value which indicates whether duplicate names should be removed from the value.

Details

These functions differ only in the default values for their arguments.

Value

A character vector with the extracted names.

Examples

```
all.names(expression(sin(x+y)))
all.vars(expression(sin(x+y)))
```

---

any *Are Some Values True?*

Description

Given a set of logical vectors, is at least one of the values true?

Usage

```
any(..., na.rm = FALSE)
```

Arguments

`...`	zero or more logical vectors. Other objects of zero length are ignored, and the rest are coerced to logical ignoring any class.
`na.rm`	logical. If true `NA` values are removed before the result is computed.

Details

This is a generic function: methods can be defined for it directly or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

Coercion of types other than integer (raw, double, complex, character, list) gives a warning as from R 2.7.0, as this was often unintentional.

Value

The value is a logical vector of length one.

Let `x` denote the concatenation of all the logical vectors in `...` (after coercion), after removing `NA`s if requested by `na.rm = TRUE`.

The value returned is `TRUE` if at least one of the values in `x` is `TRUE`, and `FALSE` if all of the values in `x` are `FALSE` (including if there are no values). Otherwise the value is `NA` (which can only occur if `na.rm = FALSE` and `...` contains no `TRUE` values and at least one `NA` value).

S4 methods

This is part of the S4 `Summary` group generic. Methods for it must use the signature `x, ..., na.rm`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`all`, the 'complement' of `any`.

Examples

```
range(x <- sort(round(stats::rnorm(10) - 1.2,1)))
if(any(x < 0)) cat("x contains negative values\n")
```

---

aperm *Array Transposition*

Description

Transpose an array by permuting its dimensions and optionally resizing it.

Usage

```
aperm(a, perm, resize = TRUE)
```

Arguments

`a`	the array to be transposed.
`perm`	the subscript permutation vector, which must be a permutation of the integers `1:n`, where `n` is the number of dimensions of `a`. The default is to reverse the order of the dimensions.
`resize`	a flag indicating whether the vector should be resized as well as having its elements reordered (default `TRUE`).

Value

A transposed version of array `a`, with subscripts permuted as indicated by the array `perm`. If `resize` is `TRUE`, the array is reshaped as well as having its elements permuted, the `dimnames` are also permuted; if `resize = FALSE` then the returned object has the same dimensions as `a`, and the dimnames are dropped. In each case other attributes are copied from `a`.

The function `t` provides a faster and more convenient way of transposing matrices.

Author(s)

Jonathan Rougier, ⟨J.C.Rougier@durham.ac.uk⟩ did the faster C implementation.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`t`, to transpose matrices.

Examples

```
# interchange the first two subscripts on a 3-way array x
x  <- array(1:24, 2:4)
xt <- aperm(x, c(2,1,3))
stopifnot(t(xt[,,2]) == x[,,2],
          t(xt[,,3]) == x[,,3],
          t(xt[,,4]) == x[,,4])
```

---

append *Vector Merging*

Description

Add elements to a vector.

Usage

```
append(x, values, after = length(x))
```

Arguments

`x`	the vector to be modified.
`values`	to be included in the modified vector.
`after`	a subscript, after which the values are to be appended.

Value

A vector containing the values in `x` with the elements of `values` appended after the specified element of `x`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
append(1:5, 0:1, after=3)
```

---

apply *Apply Functions Over Array Margins*

Description

Returns a vector or array or list of values obtained by applying a function to margins of an array.

Usage

```
apply(X, MARGIN, FUN, ...)
```

Arguments

`X`	the array to be used.
`MARGIN`	a vector giving the subscripts which the function will be applied over. `1` indicates rows, `2` indicates columns, `c(1,2)` indicates rows and columns.
`FUN`	the function to be applied: see 'Details'. In the case of functions like `+`, `%*%`, etc., the function name must be backquoted or quoted.
`...`	optional arguments to `FUN`.

Details

If `X` is not an array but has a dimension attribute, `apply` attempts to coerce it to an array via `as.matrix` if it is two-dimensional (e.g., data frames) or via `as.array`.

`FUN` is found by a call to `match.fun` and typically is either a function or a symbol (e.g. a backquoted name) or a character string specifying a function to be searched for from the environment of the call to `apply`.

Value

If each call to `FUN` returns a vector of length `n`, then `apply` returns an array of dimension `c(n, dim(X)[MARGIN])` if `n > 1`. If `n` equals `1`, `apply` returns a vector if `MARGIN` has length 1 and an array of dimension `dim(X)[MARGIN]` otherwise. If `n` is `0`, the result has length 0 but not necessarily the 'correct' dimension.

If the calls to `FUN` return vectors of different lengths, `apply` returns a list of length `prod(dim(X)[MARGIN])` with `dim` set to `MARGIN` if this has length greater than one.

In all cases the result is coerced by `as.vector` to one of the basic vector types before the dimensions are set, so that (for example) factor results will be coerced to a character array.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

lapply, tapply, and convenience functions sweep and aggregate.

Examples

```
## Compute row and column sums for a matrix:
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
dimnames(x)[[1]] <- letters[1:8]
apply(x, 2, mean, trim = .2)
col.sums <- apply(x, 2, sum)
row.sums <- apply(x, 1, sum)
rbind(cbind(x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))

stopifnot( apply(x, 2, is.vector))

## Sort the columns of a matrix
apply(x, 2, sort)

##- function with extra args:
cave <- function(x, c1, c2) c(mean(x[c1]), mean(x[c2]))
apply(x,1, cave,  c1="x1", c2=c("x1","x2"))

ma <- matrix(c(1:4, 1, 6:8), nrow = 2)
ma
apply(ma, 1, table)  #--> a list of length 2
apply(ma, 1, stats::quantile)# 5 x n matrix with rownames

stopifnot(dim(ma) == dim(apply(ma, 1:2, sum)))

## Example with different lengths for each call
z <- array(1:24, dim=2:4)
zseq <- apply(z, 1:2, function(x) seq_len(max(x)))
zseq         ## a 2 x 3 matrix
typeof(zseq) ## list
dim(zseq) ## 2 3
zseq[1,]
apply(z, 3, function(x) seq_len(max(x)))
# a list without a dim attribute
```

---

args *Argument List of a Function*

Description

Displays the argument names and corresponding default values of a function or primitive.

Usage

```
args(name)
```

Arguments

name	a function (a closure or a primitive). If name is a character string then the function with that name is found and used.

Details

This function is mainly used interactively to print the argument list of a function. For programming, consider using `formals` instead.

Value

For a closure, a closure with identical formal argument list but an empty (`NULL`) body.

For a primitive, a closure with the documented usage and `NULL` body. Note that in almost all cases primitives do not make use of named arguments and match by position rather than name. (`rep` and `seq.int` are exceptions.)

`NULL` in case of a non-function.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`formals`, `help`.

Examples

```
args(c)
args(graphics::plot.default)
```

`Arithmetic` *Arithmetic Operators*

Description

These binary operators perform arithmetic on numeric or complex vectors (or objects which can be coerced to them).

Usage

```
x + y
x - y
x * y
x / y
x ^ y
x %% y
x %/% y
```

Arguments

`x, y`	numeric or complex vectors or objects which can be coerced to such, or other objects for which methods have been written.

Details

The binary arithmetic operators are generic functions: methods can be written for them individually or via the `Ops` group generic function. (See `Ops` for how dispatch is computed.)

If applied to arrays the result will be an array if this is sensible (for example it will not if the recycling rule has been invoked).

Logical vectors will be coerced to integer or numeric vectors, `FALSE` having value zero and `TRUE` having value one.

`1 ^ y` and `y ^ 0` are `1`, *always*. `x ^ y` should also give the proper limit result when either argument is infinite (i.e., `+-Inf`).

Objects such as arrays or time-series can be operated on this way provided they are conformable.

For real arguments, `%%` can be subject to catastrophic loss of accuracy if `x` is much larger than `y`, and a warning is given if this is detected.

Value

These operators return vectors containing the result of the element by element operations. The elements of shorter vectors are recycled as necessary (with a `warning` when they are recycled only *fractionally*). The operators are + for addition, − for subtraction, ⋆ for multiplication, / for division and ^ for exponentiation.

`%%` indicates `x mod y` and `%/%` indicates integer division. It is guaranteed that `x == (x %% y) + y * ( x %/% y )` (up to rounding error) unless `y == 0` where the result is `NA_integer_` or `NaN` (depending on the `typeof` of the arguments). See `http://en.wikipedia.org/wiki/Modulo_operation` for the rationale.

If either argument is complex the result will be complex, and if one or both arguments are numeric, the result will be numeric. If both arguments are integer, the result of / and ^ is numeric and of the other operators integer (with overflow returned as `NA` with a warning).

The rules for determining the attributes of the result are rather complicated. Most attributes are taken from the longer argument, the first if they are of the same length. Names will be copied from the first if it is the same length as the answer, otherwise from the second if that is. For time series, these operations are allowed only if the series are compatible, when the class and `tsp` attribute of whichever is a time series (the same, if both are) are used. For arrays (and an array result) the dimensions and dimnames are taken from first argument if it is an array, otherwise the second.

S4 methods

These operators are members of the S4 `Arith` group generic, and so methods can be written for them individually as well as for the group generic (or the `Ops` group generic), with arguments `c(e1, e2)`.

Note

⋆⋆ is translated in the parser to ^, but this was undocumented for many years. It appears as an index entry in Becker *et al* (1988), pointing to the help for `Deprecated` but is not actually mentioned on that page. Even though it has been deprecated in S for 20 years, it is still accepted.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`sqrt` for miscellaneous and `Special` for special mathematical functions.

`Syntax` for operator precedence.

`%*%` for matrix multiplication.

Examples

```
x <- -1:12
x + 1
2 * x + 3
x %% 2 #-- is periodic
x %/% 5
```

`array` *Multi-way Arrays*

Description

Creates or tests for arrays.

Usage

```
array(data = NA, dim = length(data), dimnames = NULL)
as.array(x, ...)
is.array(x)
```

Arguments

`data`	a vector (including a list) giving data to fill the array.
`dim`	the dim attribute for the array to be created, that is a vector of length one or more giving the maximal indices in each dimension.
`dimnames`	either `NULL` or the names for the dimensions. This is a list with one component for each dimension, either `NULL` or a character vector of the length given by `dim` for that dimension. The list can be named, and the list names will be used as names for the dimensions. If the list is shorter than the number of dimensions, it is extended by `NULL`s to the length required
`x`	an R object.
`...`	additional arguments to be passed to or from methods.

Value

`array` returns an array with the extents specified in `dim` and naming information in `dimnames`. The values in `data` are taken to be those in the array with the leftmost subscript moving fastest. If there are too few elements in `data` to fill the array, then the elements in `data` are recycled. If `data` has length zero, `NA` of an appropriate type is used for atomic vectors (`0` for raw vectors) and `NULL` for lists.

`as.array` is a generic function for coercing to arrays. The default method does so by attaching a `dim` attribute to it. It also attaches `dimnames` if `x` has `names`. The sole purpose of this is to make it possible to access the `dim`[names] attribute at a later time.

`is.array` returns `TRUE` or `FALSE` depending on whether its argument is an array (i.e., has a `dim` attribute of positive length) or not. It is generic: you can write methods to handle specific classes of objects, see InternalMethods.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`aperm`, `matrix`, `dim`, `dimnames`.

Examples

```
dim(as.array(letters))
array(1:3, c(2,4)) # recycle 1:3 "2 2/3 times"
#     [,1] [,2] [,3] [,4]
#[1,]    1    3    2    1
#[2,]    2    1    3    2
```

`as.data.frame` *Coerce to a Data Frame*

Description

Functions to check if an object is a data frame, or coerce it if possible.

Usage

```
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
## S3 method for class 'character':
as.data.frame(x, ...,
              stringsAsFactors = default.stringsAsFactors())
## S3 method for class 'matrix':
as.data.frame(x, row.names = NULL, optional = FALSE, ...,
              stringsAsFactors = default.stringsAsFactors())

is.data.frame(x)
```

Arguments

`x`	any R object.
`row.names`	`NULL` or a character vector giving the row names for the data frame. Missing values are not allowed.
`optional`	logical. If `TRUE`, setting row names and converting column names (to syntactic names: see `make.names`) is optional.
`...`	additional arguments to be passed to or from methods.
`stringsAsFactors`	logical: should the character vector be converted to a factor?

Details

as.data.frame is a generic function with many methods, and users and packages can supply further methods.

If a list is supplied, each element is converted to a column in the data frame. Similarly, each column of a matrix is converted separately. This can be overridden if the object has a class which has a method for as.data.frame: two examples are matrices of class "model.matrix" (which are included as a single column) and list objects of class "POSIXlt" which are coerced to class "POSIXct".

Arrays can be converted to data frames. One-dimensional arrays are treated like vectors and two-dimensional arrays like matrices. Arrays with more than two dimensions are converted to matrices by 'flattening' all dimensions after the first and creating suitable column labels.

Character variables are converted to factor columns unless protected by I.

If a data frame is supplied, all classes preceding "data.frame" are stripped, and the row names are changed if that argument is supplied.

If row.names = NULL, row names are constructed from the names or dimnames of x, otherwise are the integer sequence starting at one. Few of the methods check for duplicated row names. Names are removed from vector columns unless I.

Value

as.data.frame returns a data frame, normally with all row names "" if optional = TRUE.

is.data.frame returns TRUE if its argument is a data frame (that is, has "data.frame" amongst its classes) and FALSE otherwise.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

data.frame, as.data.frame.table for the table method (which has additional arguments if called directly).

---

as.environment *Coerce to an Environment Object*

Description

Converts a number or a character string to the corresponding environment on the search path.

Usage

```
as.environment(object)
```

Arguments

object	the object to convert. If it is already an environment, just return it. If it is a number, return the environment corresponding to that position on the search list. If it is a character string, match the string to the names on the search list.

Value

The corresponding environment object.

Author(s)

John Chambers

See Also

`environment` for creation and manipulation, `search`.

Examples

```
as.environment(1) ## the global environment
identical(globalenv(), as.environment(1)) ## is TRUE
try(as.environment("package:stats"))      ## stats need not be loaded
```

`as.function` *Convert Object to Function*

Description

`as.function` is a generic function which is used to convert objects to functions.

`as.function.default` works on a list `x`, which should contain the concatenation of a formal argument list and an expression or an object of mode `"call"` which will become the function body. The function will be defined in a specified environment, by default that of the caller.

Usage

```
as.function(x, ...)

## Default S3 method:
as.function(x, envir = parent.frame(), ...)
```

Arguments

`x`	object to convert, a list for the default method.
`...`	additional arguments, depending on object
`envir`	environment in which the function should be defined

Value

The desired function.

Note

For ancient historical reasons, `envir = NULL` uses the global environment rather than the base environment. Please use `envir = globalenv()` instead if this is what you want, as the special handling of `NULL` may change in a future release.

Author(s)

Peter Dalgaard

See Also

`function`; `alist` which is handy for the construction of argument lists, etc.

Examples

```
as.function(alist(a=,b=2,a+b))
as.function(alist(a=,b=2,a+b))(3)
```

---

as.POSIX* *Date-time Conversion Functions*

Description

Functions to manipulate objects of classes `"POSIXlt"` and `"POSIXct"` representing calendar dates and times.

Usage

```
as.POSIXct(x, tz = "", ...)
as.POSIXlt(x, tz = "", ...)

## S3 method for class 'character':
as.POSIXlt(x, tz = "", format, ...)

## S3 method for class 'numeric':
as.POSIXlt(x, tz = "", origin, ...)

## S3 method for class 'POSIXlt':
as.double(x, ...)
```

Arguments

`x`	An object to be converted.
`tz`	A timezone specification to be used for the conversion, *if one is required*. System-specific (see time zones), but `""` is the current timezone, and `"GMT"` is UTC (Universal Time, Coordinated).
`...`	further arguments to be passed to or from other methods.
`format`	character string giving a date-time format as used by `strptime`.
`origin`	a date-time object, or something which can be coerced by `as.POSIXct(tz="GMT")` to such an object.

Details

The as.POSIX* functions convert an object to one of the two classes used to represent date/times (calendar dates plus time to the nearest second). They can convert a wide variety of objects, including objects of the other class and of classes "Date", "date" (from package **date** or **survival**), "chron" and "dates" (from package **chron**) to these classes. Dates without times are treated as being at midnight UTC.

They can also convert character strings of the formats "2001-02-03" and "2001/02/03" optionally followed by white space and a time in the format "14:52" or "14:52:03". (Formats such as "01/02/03" are ambiguous but can be converted via a format specification by strptime.) Fractional seconds are allowed. Alternatively, format can be specified for character vectors or factors.

Logical NAs can be converted to either of the classes, but no other logical vectors can be.

The as.double method converts "POSIXlt" objects to "POSIXct".

If you are given a numeric time as the number of seconds since an epoch, see the examples.

Value

as.POSIXct and as.POSIXlt return an object of the appropriate class. If tz was specified, as.POSIXlt will give an appropriate "tzone" attribute.

Note

If you want to extract specific aspects of a time (such as the day of the week) just convert it to class "POSIXlt" and extract the relevant component(s) of the list, or if you want a character representation (such as a named day of the week) use format.POSIXlt or format.POSIXct.

If a timezone is needed and that specified is invalid on your system, what happens is system-specific but it will probably be ignored.

See Also

DateTimeClasses for details of the classes; strptime for conversion to and from character representations. Sys.timezone for details of the (system-specific)naming of time zones.

Examples

```
(z <- Sys.time())             # the current datetime, as class "POSIXct"
unclass(z)                    # a large integer
floor(unclass(z)/86400)       # the number of days since 1970-01-01
(z <- as.POSIXlt(Sys.time())) # the current datetime, as class "POSIXlt"
unlist(unclass(z))            # a list shown as a named vector

## suppose we have a time in seconds since 1960-01-01 00:00:00 GMT
z <- 1472562988
# ways to convert this
as.POSIXct(z, origin="1960-01-01")                # local
as.POSIXct(z, origin="1960-01-01", tz="GMT")      # in UTC
as.POSIXct(z, origin=ISOdatetime(1960,1,1,0,0,0)) # local
ISOdatetime(1960,1,1,0,0,0) + z                   # local

## SPSS dates (R-help 2006-02-17)
z <- c(10485849600, 10477641600, 10561104000, 10562745600)
as.Date(as.POSIXct(z, origin="1582-10-14", tz="GMT"))
```

```
as.POSIXlt(Sys.time(), "GMT") # the current time in UTC
## Not run:
## These may not be correct names on your system
as.POSIXlt(Sys.time(), "America/New_York")  # in New York
as.POSIXlt(Sys.time(), "EST5EDT")           # alternative.
as.POSIXlt(Sys.time(), "EST" )   # somewhere in Eastern Canada
as.POSIXlt(Sys.time(), "HST")    # in Hawaii
as.POSIXlt(Sys.time(), "Australia/Darwin")
## End(Not run)
```

AsIs *Inhibit Interpretation/Conversion of Objects*

Description

Change the class of an object to indicate that it should be treated 'as is'.

Usage

```
I(x)
```

Arguments

x	an object

Details

Function I has two main uses.

- In function data.frame. Protecting an object by enclosing it in I() in a call to data.frame inhibits the conversion of character vectors to factors and the dropping of names, and ensures that matrices are inserted as single columns. I can also be used to protect objects which are to be added to a data frame, or converted to a data frame *via* as.data.frame.

  It achieves this by prepending the class "AsIs" to the object's classes. Class "AsIs" has a few of its own methods, including for [, as.data.frame, print and format.
- In function formula. There it is used to inhibit the interpretation of operators such as "+", "-", "*" and "^" as formula operators, so they are used as arithmetical operators. This is interpreted as a symbol by terms.formula.

Value

A copy of the object with class "AsIs" prepended to the class(es).

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

data.frame, formula

`assign` *Assign a Value to a Name*

Description

Assign a value to a name in an environment.

Usage

```
assign(x, value, pos = -1, envir = as.environment(pos),
       inherits = FALSE, immediate = TRUE)
```

Arguments

`x`	a variable name, given as a character string. No coercion is done, and the first element of a character vector of length greater than one will be used, with a warning.
`value`	a value to be assigned to `x`.
`pos`	where to do the assignment. By default, assigns into the current environment. See the details for other possibilities.
`envir`	the `environment` to use. See the details section.
`inherits`	should the enclosing frames of the environment be inspected?
`immediate`	an ignored compatibility feature.

Details

There are no restrictions on `name`: it can be a non-syntactic name (see `make.names`).

The `pos` argument can specify the environment in which to assign the object in any of several ways: as an integer (the position in the `search` list); as the character string name of an element in the search list; or as an `environment` (including using `sys.frame` to access the currently active function calls). The `envir` argument is an alternative way to specify an environment, but is primarily there for back compatibility.

`assign` does not dispatch assignment methods, so it cannot be used to set elements of vectors, names, attributes, etc.

Note that assignment to an attached list or data frame changes the attached copy and not the original object: see `attach` and `with`.

Value

This function is invoked for its side effect, which is assigning `value` to the variable `x`. If no `envir` is specified, then the assignment takes place in the currently active environment.

If `inherits` is `TRUE`, enclosing environments of the supplied environment are searched until the variable `x` is encountered. The value is then assigned in the environment in which the variable is encountered (provided that the binding is not locked: see `lockBinding`: if it is, an error is signaled). If the symbol is not encountered then assignment takes place in the user's workspace (the global environment).

If `inherits` is `FALSE`, assignment takes place in the initial frame of `envir`, unless an existing binding is locked or there is no existing binding and the environment is locked.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`<-`, `get`, `exists`, `environment`.

Examples

```
for(i in 1:6) { #-- Create objects  'r.1', 'r.2', ... 'r.6' --
 nam <- paste("r",i, sep=".")
 assign(nam, 1:i)
}
ls(pattern = "^r..$")

##-- Global assignment within a function:
myf <- function(x) {
 innerf <- function(x) assign("Global.res", x^2, envir = .GlobalEnv)
 innerf(x+1)
}
myf(3)
Global.res # 16

a <- 1:4
assign("a[1]", 2)
a[1] == 2          #FALSE
get("a[1]") == 2   #TRUE
```

`assignOps` *Assignment Operators*

Description

Assign a value to a name.

Usage

```
x <- value
x <<- value
value -> x
value ->> x

x = value
```

Arguments

`x`	a variable name (possibly quoted).
`value`	a value to be assigned to `x`.

Details

There are three different assignment operators: two of them have leftwards and rightwards forms.

The operators <- and = assign into the environment in which they are evaluated. The operator <- can be used anywhere, whereas the operator = is only allowed at the top level (e.g., in the complete expression typed at the command prompt) or as one of the subexpressions in a braced list of expressions.

The operators <<- and ->> cause a search to made through the environment for an existing definition of the variable being assigned. If such a variable is found (and its binding is not locked) then its value is redefined, otherwise assignment takes place in the global environment. Note that their semantics differ from that in the S language, but are useful in conjunction with the scoping rules of R. See 'The R Language Definition' manual for further details and examples.

In all the assignment operator expressions, `x` can be a name or an expression defining a part of an object to be replaced (e.g., `z[[1]]`). A syntactic name does not need to be quoted, though it can be (preferably by backticks).

The leftwards forms of assignment `<- = <<-` group right to left, the other from left to right.

Value

`value`. Thus one can use `a <- b <- c <- 6`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chamber, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer (for =).

See Also

`assign`, `environment`.

`attach` *Attach Set of R Objects to Search Path*

Description

The database is attached to the R search path. This means that the database is searched by R when evaluating a variable, so objects in the database can be accessed by simply giving their names.

Usage

```
attach(what, pos = 2, name = deparse(substitute(what)),
       warn.conflicts = TRUE)
```

Arguments

what	'database'. This can be a data.frame or a list or a R data file created with save or NULL or an environment. See also 'Details'.
pos	integer specifying position in search() where to attach.
name	name to use for the attached database.
warn.conflicts	logical. If TRUE, warnings are printed about conflicts from attaching the database, unless that database contains an object .conflicts.OK. A conflict is a function masking a function, or a non-function masking a non-function.

Details

When evaluating a variable or function name R searches for that name in the databases listed by search. The first name of the appropriate type is used.

By attaching a data frame (or list) to the search path it is possible to refer to the variables in the data frame by their names alone, rather than as components of the data frame (e.g. in the example below, height rather than women$height).

By default the database is attached in position 2 in the search path, immediately after the user's workspace and before all previously loaded packages and previously attached databases. This can be altered to attach later in the search path with the pos option, but you cannot attach at pos=1.

The database is not actually attached. Rather, a new environment is created on the search path and the elements of a list (including columns of a data frame) or objects in a save file or an environment are *copied* into the new environment. If you use <<- or assign to assign to an attached database, you only alter the attached copy, not the original object. (Normal assignment will place a modified version in the user's workspace: see the examples.) For this reason attach can lead to confusion.

One useful 'trick' is to use what = NULL (or equivalently a length-zero list) to create a new environment on the search path into which objects can be assigned by assign or load or sys.source.

Names starting "package:" are reserved for library and should not be used by end users. The name given for the attached environment will be used by search and can be used as the argument to as.environment.

There are hooks to attach user-defined table objects of class "UserDefinedDatabase", supported by the Omegahat package **RObjectTables**. See http://www.omegahat.org/RObjectTables/.

Value

The environment is returned invisibly with a "name" attribute.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

library, detach, search, objects, environment, with.

Examples

```
require(utils)

summary(women$height)    # refers to variable 'height' in the data frame
attach(women)
summary(height)          # The same variable now available by name
height <- height*2.54    # Don't do this. It creates a new variable
                         # in the user's workspace
find("height")
summary(height)          # The new variable in the workspace
rm(height)
summary(height)          # The original variable.
height <<- height*25.4   # Change the copy in the attached environment
find("height")
summary(height)          # The changed copy
detach("women")
summary(women$height)    # unchanged

## Not run:
## create an environment on the search path and populate it
sys.source("myfuns.R", envir=attach(NULL, name="myfuns"))
## End(Not run)
```

`attr` *Object Attributes*

Description

Get or set specific attributes of an object.

Usage

```
attr(x, which, exact = FALSE)
attr(x, which) <- value
```

Arguments

`x`	an object whose attributes are to be accessed.
`which`	a non-empty character string specifying which attribute is to be accessed.
`exact`	logical: should `which` be matched exactly?
`value`	an object, the new value of the attribute, or `NULL` to remove the attribute.

Details

These functions provide access to a single attribute of an object. The replacement form causes the named attribute to take the value specified (or create a new attribute with the value given).

The extraction function first looks for an exact match to `which` amongst the attributes of `x`, then (unless `exact = TRUE`) a unique partial match. (Setting `options(warnPartialMatchAttr=TRUE)` causes partial matches to give warnings.)

The replacement function only uses exact matches.

Note that some attributes (namely `class`, `comment`, `dim`, `dimnames`, `names`, `row.names` and `tsp`) are treated specially and have restrictions on the values which can be set. (Note that this is not true of `levels` which should be set for factors via the `levels` replacement function.)

The extractor function allows (and does not match) empty and missing values of `which`: the replacement function does not.

Value

For the extractor, the value of the attribute matched, or `NULL` if no exact match is found and no or more than one partial match is found.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`attributes`

Examples

```
# create a 2 by 5 matrix
x <- 1:10
attr(x,"dim") <- c(2, 5)
```

`attributes` *Object Attribute Lists*

Description

These functions access an object's attributes. The first form below returns the object's attribute list. The replacement forms uses the list on the right-hand side of the assignment as the object's attributes (if appropriate).

Usage

```
attributes(obj)
attributes(obj) <- value
mostattributes(obj) <- value
```

Arguments

`obj`	an object
`value`	an appropriate named list of attributes, or `NULL`.

Details

Unlike attr it is possible to set attributes on a NULL object: it will first be coerced to an empty list.

Note that some attributes (namely class, comment, dim, dimnames, names, row.names and tsp) are treated specially and have restrictions on the values which can be set. (Note that this is not true of levels which should be set for factors via the levels replacement function.)

Attributes are not stored internally as a list and should be thought of as a set and not a vector. They must have unique names (and NA is taken as "NA", not a missing value).

Assigning attributes first removes all attributes, then sets any dim attribute and then the remaining attributes in the order given: this ensures that setting a dim attribute always precedes the dimnames attribute.

The mostattributes assignment takes special care for the dim, names and dimnames attributes, and assigns them only when valid whereas an attributes assignment would give an error if any are not.

The names of a pairlist are not stored as attributes, but are reported as if they were (and can be set by the replacement method for attributes).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

attr.

Examples

```
x <- cbind(a=1:3, pi=pi) # simple matrix w/ dimnames
attributes(x)

## strip an object's attributes:
attributes(x) <- NULL
x # now just a vector of length 6

mostattributes(x) <- list(mycomment = "really special", dim = 3:2,
   dimnames = list(LETTERS[1:3], letters[1:5]), names = paste(1:6))
x # dim(), but not {dim}names
```

---

autoload *On-demand Loading of Packages*

Description

autoload creates a promise-to-evaluate autoloader and stores it with name name in .AutoloadEnv environment. When R attempts to evaluate name, autoloader is run, the package is loaded and name is re-evaluated in the new package's environment. The result is that R behaves as if file was loaded but it does not occupy memory.

.Autoloaded contains the names of the packages for which autoloading has been promised.

Usage

```
autoload(name, package, reset = FALSE, ...)
autoloader(name, package, ...)

.AutoloadEnv
.Autoloaded
```

Arguments

`name`	string giving the name of an object.
`package`	string giving the name of a package containing the object.
`reset`	logical: for internal use by `autoloader`.
`...`	other arguments to `library`.

Value

This function is invoked for its side-effect. It has no return value.

See Also

`delayedAssign`, `library`

Examples

```
require(stats)
autoload("interpSpline", "splines")
search()
ls("Autoloads")
.Autoloaded

x <- sort(stats::rnorm(12))
y <- x^2
is <- interpSpline(x,y)
search() ## now has splines
detach("package:splines")
search()
is2 <- interpSpline(x,y+x)
search() ## and again
detach("package:splines")
```

`backsolve` *Solve an Upper or Lower Triangular System*

Description

Solves a system of linear equations where the coefficient matrix is upper or lower triangular.

Usage

```
   backsolve(r, x, k=ncol(r), upper.tri=TRUE, transpose=FALSE)
forwardsolve(l, x, k=ncol(l), upper.tri=FALSE, transpose=FALSE)
```

Arguments

r, l	an upper (or lower) triangular matrix giving the coefficients for the system to be solved. Values below (above) the diagonal are ignored.
x	a matrix whose columns give the right-hand sides for the equations.
k	The number of columns of r and rows of x to use.
upper.tri	logical; if TRUE (default), the *upper tri*angular part of r is used. Otherwise, the lower one.
transpose	logical; if TRUE, solve $r' * y = x$ for y, i.e., t(r) %*% y == x.

Value

The solution of the triangular system. The result will be a vector if x is a vector and a matrix if x is a matrix.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch,J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

See Also

chol, qr, solve.

Examples

```
## upper triangular matrix 'r':
r <- rbind(c(1,2,3),
           c(0,1,1),
           c(0,0,2))
( y <- backsolve(r, x <- c(8,4,2)) ) # -1 3 1
r %*% y # == x = (8,4,2)
backsolve(r, x, transpose = TRUE) # 8 -12 -5
```

---

base-deprecated *Deprecated Functions in Base package*

Description

These functions are provided for compatibility with older versions of R only, and may be defunct as soon as the next release.

Usage

```
Sys.putenv(...)
```

Arguments

...	named arguments with values coercible to a character string.

Details

The original help page for these functions is often available at `help("oldName-deprecated")` (note the quotes). Functions in packages other than the base package are listed in `help("pkg-deprecated")`.

`Sys.putenv` is a deprecated synonym for `Sys.setenv`.

See Also

`Deprecated`, `base-defunct`

---

basename *Manipulate File Paths*

Description

`basename` removes all of the path up to the last path separator (if any).

`dirname` returns the part of the `path` up to (but excluding) the last path separator, or `"."` if there is no path separator.

Usage

```
basename(path)
dirname(path)
```

Arguments

`path`	character vector, containing path names.

Details

For `dirname` tilde expansion is done: see the description of `path.expand`.

Trailing file separators are removed before dissecting the path, and for `dirname` any trailing file separators are removed from the result.

If an element of `path` is `NA`, so is the result.

Value

A character vector of the same length as `path`. A zero-length input will give a zero-length output with no error.

See Also

`file.path`, `path.expand`.

Examples

```
basename(file.path("","p1","p2","p3", c("file1", "file2")))
dirname(file.path("","p1","p2","p3","filename"))
```

`Bessel` *Bessel Functions*

Description

Bessel Functions of integer and fractional order, of first and second kind, J_ν and Y_ν, and Modified Bessel functions (of first and third kind), I_ν and K_ν.

Usage

```
besselI(x, nu, expon.scaled = FALSE)
besselK(x, nu, expon.scaled = FALSE)
besselJ(x, nu)
besselY(x, nu)
```

Arguments

`x`	numeric, ≥ 0.
`nu`	numeric; The *order* (maybe fractional!) of the corresponding Bessel function.
`expon.scaled`	logical; if `TRUE`, the results are exponentially scaled in order to avoid overflow (I_ν) or underflow (K_ν), respectively.

Details

If `expon.scaled = TRUE`, $e^{-x} I_\nu(x)$, or $e^x K_\nu(x)$ are returned.

For $\nu < 0$, formulae 9.1.2 and 9.6.2 from Abramowitz & Stegun are applied (which is probably suboptimal), except for `besselK` which is symmetric in `nu`.

Value

Numeric vector of the same length of `x` with the (scaled, if `expon.scale=TRUE`) values of the corresponding Bessel function.

Author(s)

Original Fortran code: W. J. Cody, Argonne National Laboratory
Translation to C and adaption to R: Martin Maechler ⟨maechler@stat.math.ethz.ch.⟩

Source

The C code is a translation of Fortran routines from `http://www.netlib.org/specfun/r[ijky]besl`.

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* Dover, New York; Chapter 9: Bessel Functions of Integer Order.

See Also

Other special mathematical functions, such as `gamma`, $\Gamma(x)$, and `beta`, $B(x)$.

Examples

```
require(graphics)

nus <- c(0:5, 10, 20)

x <- seq(0, 4, length.out = 501)
plot(x, x, ylim = c(0, 6), ylab = "", type = "n",
     main = "Bessel Functions  I_nu(x)")
for(nu in nus) lines(x, besselI(x, nu=nu), col = nu+2)
legend(0, 6, legend = paste("nu=", nus), col = nus+2, lwd = 1)

x <- seq(0, 40, length.out = 801); yl <- c(-.8, .8)
plot(x, x, ylim = yl, ylab = "", type = "n",
     main = "Bessel Functions  J_nu(x)")
for(nu in nus) lines(x, besselJ(x, nu=nu), col = nu+2)
legend(32,-.18, legend = paste("nu=", nus), col = nus+2, lwd = 1)

## Negative nu's :
xx <- 2:7
nu <- seq(-10, 9, length.out = 2001)
op <- par(lab = c(16, 5, 7))
matplot(nu, t(outer(xx, nu, besselI)), type = "l", ylim = c(-50, 200),
        main = expression(paste("Bessel ", I[nu](x), " for fixed ", x,
                                ",  as ", f(nu))),
        xlab = expression(nu))
abline(v=0, col = "light gray", lty = 3)
legend(5, 200, legend = paste("x=", xx), col=seq(xx), lty=seq(xx))
par(op)

x0 <- 2^(-20:10)
plot(x0, x0^-8, log="xy", ylab="",type="n",
     main = "Bessel Functions  J_nu(x)  near 0\n log - log  scale")
for(nu in sort(c(nus, nus+.5)))
    lines(x0, besselJ(x0, nu=nu), col = nu+2)
legend(3, 1e50, legend = paste("nu=", paste(nus, nus+.5, sep=",")),
       col = nus + 2, lwd = 1)

plot(x0, x0^-8, log="xy", ylab="", type="n",
     main = "Bessel Functions  K_nu(x)  near 0\n log - log  scale")
for(nu in sort(c(nus, nus+.5)))
    lines(x0, besselK(x0, nu=nu), col = nu+2)
legend(3, 1e50, legend = paste("nu=", paste(nus, nus+.5, sep=",")),
       col = nus + 2, lwd = 1)

x <- x[x > 0]
plot(x, x, ylim=c(1e-18, 1e11), log = "y", ylab = "", type = "n",
     main = "Bessel Functions  K_nu(x)")
for(nu in nus) lines(x, besselK(x, nu=nu), col = nu+2)
legend(0, 1e-5, legend=paste("nu=", nus), col = nus+2, lwd = 1)

yl <- c(-1.6, .6)
plot(x, x, ylim = yl, ylab = "", type = "n",
     main = "Bessel Functions  Y_nu(x)")
for(nu in nus){
    xx <- x[x > .6*nu]
    lines(xx, besselY(xx, nu=nu), col = nu+2)
```

```
}
legend(25, -.5, legend = paste("nu=", nus), col = nus+2, lwd = 1)

## negative nu in bessel_Y -- was bogous for a long time
curve(besselY(x, -0.1), 0, 10, ylim = c(-3,1), ylab = '')
for(nu in c(seq(-0.2, -2, by = -0.1)))
  curve(besselY(x, nu), add = TRUE)
title(expression(besselY(x, nu) * "    " *
                 {nu == list(-0.1, -0.2, ..., -2)}))
```

`bindenv` *Binding and Environment Adjustments*

Description

These functions represent an experimental interface for adjustments to environments and bindings within environments. They allow for locking environments as well as individual bindings, and for linking a variable to a function.

Usage

```
lockEnvironment(env, bindings = FALSE)
environmentIsLocked(env)
lockBinding(sym, env)
unlockBinding(sym, env)
bindingIsLocked(sym, env)
makeActiveBinding(sym, fun, env)
bindingIsActive(sym, env)
```

Arguments

`env`	an environment.
`bindings`	logical specifying whether bindings should be locked.
`sym`	a name object or character string
`fun`	a function taking zero or one arguments

Details

The function `lockEnvironment` locks its environment argument, which must be a normal environment (not base). (Locking the base environment and name space may be supported later.) Locking the environment prevents adding or removing variable bindings from the environment. Changing the value of a variable is still possible unless the binding has been locked. The name space environments of packages with name spaces are locked when loaded.

`lockBinding` locks individual bindings in the specified environment. The value of a locked binding cannot be changed. Locked bindings may be removed from an environment unless the environment is locked.

`makeActiveBinding` installs `fun` so that getting the value of `sym` calls `fun` with no arguments, and assigning to `sym` calls `fun` with one argument, the value to be assigned. This allows the implementation of things like C variables linked to R variables and variables linked to databases. It may also be useful for making thread-safe versions of some system globals.

Value

The `*isLocked` funtions return a length-one logical vector. The remaining functions return `NULL`, invisibly.

Author(s)

Luke Tierney

Examples

```
# locking environments
e <- new.env()
assign("x", 1, envir = e)
get("x", envir = e)
lockEnvironment(e)
get("x", envir = e)
assign("x", 2, envir = e)
try(assign("y", 2, envir = e)) # error

# locking bindings
e <- new.env()
assign("x", 1, envir = e)
get("x", envir = e)
lockBinding("x", e)
try(assign("x", 2, envir = e)) # error
unlockBinding("x", e)
assign("x", 2, envir = e)
get("x", envir = e)

# active bindings
f <- local( {
    x <- 1
    function(v) {
       if (missing(v))
           cat("get\n")
       else {
           cat("set\n")
           x <<- v
       }
       x
    }
})
makeActiveBinding("fred", f, .GlobalEnv)
bindingIsActive("fred", .GlobalEnv)
fred
fred <- 2
fred
```

`body` *Access to and Manipulation of the Body of a Function*

Description

Get or set the body of a function.

Usage

```
body(fun = sys.function(sys.parent()))
body(fun, envir = environment(fun)) <- value
```

Arguments

fun	a function object, or see 'Details'.
envir	environment in which the function should be defined.
value	an object, usually a language object: see section 'Value'.

Details

For the first form, fun can be a character string naming the function to be manipulated, which is searched for from the parent environment. If it is not specified, the function calling body is used.

To set the value to a list call the replacement function with value a length-one list containing the desired value.

Value

body returns the body of the function specified. This is normally a language object, most often a call to {, but it can also be an object (e.g. pi) to be the return value of the function.

The replacement form sets the body of a function to the object on the right hand side, and (potentially) resets the environment of the function. If value is of class "expression" or a length-one list, the first element is used as the body. For expressions (but not lists) any additional elements are ignored, with a warning.

See Also

alist, args, function.

Examples

```
body(body)
f <- function(x) x^5
body(f) <- expression(5^x)
## or equivalently  body(f) <- list(quote(5^x))
f(3) # = 125
body(f)
```

---

bquote *Partial substitution in expressions*

Description

An analogue of the LISP backquote macro. bquote quotes its argument except that terms wrapped in .() are evaluated in the specified where environment.

Usage

```
bquote(expr, where = parent.frame())
```

Arguments

`expr`	A language object.
`where`	An environment.

Value

A language object.

See Also

`quote`, `substitute`

Examples

```
require(graphics)

a <- 2

bquote(a == a)
quote(a == a)

bquote(a == .(a))
substitute(a == A, list(A = a))

plot(1:10, a*(1:10), main = bquote(a == .(a)))
```

`browser` *Environment Browser*

Description

Interrupt the execution of an expression and allow the inspection of the environment where `browser` was called from.

Usage

```
browser()
```

Details

A call to `browser` can be included in the body of a function. When reached, this causes a pause in the execution of the current expression and allows access to the R interpreter.

At the browser prompt the user can enter commands or R expressions. The commands are

`c` (or just return) exit the browser and continue execution at the next statement.

`cont` synonym for `c`.

`n` enter the step-through debugger. This changes the meaning of `c`: see the documentation for `debug`.

`where` print a stack trace of all active function calls.

`Q` exit the browser and the current evaluation and return to the top-level prompt.

(Leading and trailing whitespace is ignored, except for return).

Anything else entered at the browser prompt is interpreted as an R expression to be evaluated in the calling environment: in particular typing an object name will cause the object to be printed, and `ls()` lists the objects in the calling frame. (If you want to look at an object with a name such as `n`, print it explicitly.)

The number of lines printed for the deparsed call can be limited by setting `options(deparse.max.lines)`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`debug`, and `traceback` for the stack on error.

`builtins` *Returns the Names of All Built-in Objects*

Description

Return the names of all the built-in objects. These are fetched directly from the symbol table of the R interpreter.

Usage

```
builtins(internal = FALSE)
```

Arguments

`internal`	a logical indicating whether only 'internal' functions (which can be called via `.Internal`) should be returned.

Details

`builtins()` returns an unsorted list of the objects in the symbol table, that is all the objects in the base environment. These are the built-in objects plus any that have been added subsequently when the base package was loaded. It is less confusing to use `ls(baseenv(), all=TRUE)`.

`builtins(TRUE)` returns an unsorted list of the names of internal functions, that is those which can be accessed as `.Internal(`*foo*`(args ...))` for *foo* in the list.

Value

A character vector.

`by` *Apply a Function to a Data Frame split by Factors*

Description

Function `by` is an object-oriented wrapper for `tapply` applied to data frames.

Usage

```
by(data, INDICES, FUN, ..., simplify = TRUE)
```

Arguments

`data`	an R object, normally a data frame, possibly a matrix.
`INDICES`	a factor or a list of factors, each of length `nrow(data)`.
`FUN`	a function to be applied to data frame subsets of `data`.
`...`	further arguments to `FUN`.
`simplify`	logical: see `tapply`.

Details

A data frame is split by row into data frames subsetted by the values of one or more factors, and function `FUN` is applied to each subset in turn.

Object `data` will be coerced to a data frame by the default method, *but* if this results in a 1-column data frame, the objects passed to `FUN` are dropped to a subsets of that column. (This was the long-term behaviour, but only documented since R 2.7.0.)

Value

An object of class `"by"`, giving the results for each subset. This is always a list if `simplify` is false, otherwise a list or array (see `tapply`).

See Also

`tapply`

Examples

```
require(stats)
attach(warpbreaks)
by(warpbreaks[, 1:2], tension, summary)
by(warpbreaks[, 1], list(wool = wool, tension = tension), summary)
by(warpbreaks, tension, function(x) lm(breaks ~ wool, data = x))

## now suppose we want to extract the coefficients by group
tmp <- by(warpbreaks, tension, function(x) lm(breaks ~ wool, data = x))
sapply(tmp, coef)

detach("warpbreaks")
```

c *Combine Values into a Vector or List*

Description

This is a generic function which combines its arguments.

The default method combines its arguments to form a vector. All arguments are coerced to a common type which is the type of the returned value, and all attributes except names are removed.

Usage

```
c(..., recursive=FALSE)
```

Arguments

`...`	objects to be concatenated.
`recursive`	logical. If `recursive = TRUE`, the function recursively descends through lists (and pairlists) combining all their elements into a vector.

Details

The output type is determined from the highest type of the components in the hierarchy NULL < raw < logical < integer < real < complex < character < list < expression. Pairlists are treated as lists, but non-vector components (such names and calls) are treated as one-element lists which cannot be unlisted even if `recursive = TRUE`.

`c` is sometimes used for its side effect of removing attributes except names, for example to turn an array into a vector. `as.vector` is a more intuitive way to do this, but also drops names. Note too that methods other than the default are not required to do this (and they will almost certainly preserve a class attribute).

Value

`NULL` or an expression or a vector of an appropriate mode. (With no arguments the value is `NULL`.)

S4 methods

This function is S4 generic, but with argument list `(x, ..., recursive = FALSE)`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`unlist` and `as.vector` to produce attribute-free vectors.

Examples

```
c(1,7:9)
c(1:5, 10.5, "next")

## uses with a single argument to drop attributes
x <- 1:4
names(x) <- letters[1:4]
x
c(x)           # has names
as.vector(x)   # no names
dim(x) <- c(2,2)
x
c(x)
as.vector(x)

## append to a list:
ll <- list(A = 1, c="C")
## do *not* use
c(ll, d = 1:3) # which is == c(ll, as.list(c(d=1:3))
## but rather
c(ll, d = list(1:3))# c() combining two lists

c(list(A=c(B=1)), recursive=TRUE)

c(options(), recursive=TRUE)
c(list(A=c(B=1,C=2), B=c(E=7)), recursive=TRUE)
```

`call` *Function Calls*

Description

Create or test for objects of mode `"call"`.

Usage

```
call(name, ...)
is.call(x)
as.call(x)
```

Arguments

`name`	a non-empty character string naming the function to be called.
`...`	arguments to be part of the call.
`x`	an arbitrary R object.

Details

`call` returns an unevaluated function call, that is, an unevaluated expression which consists of the named function applied to the given arguments (`name` must be a quoted string which gives the name of a function to be called). Note that although the call is unevaluated, the arguments `...` are evaluated.

`call` is a primitive, so the first argument (named or not) is taken as `name` and the remaining arguments as arguments for the constructed call: `call(x="c", 1,3, name="foo")` is a call to `c` and not to `foo`.

`is.call` is used to determine whether `x` is a call (i.e., of mode `"call"`).

Objects of mode `"list"` can be coerced to mode `"call"`. The first element of the list becomes the function part of the call, so should be a function or the name of one (as a symbol; a quoted string will not do).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`do.call` for calling a function by name and argument list; `Recall` for recursive calling of functions; further `is.language`, `expression`, `function`.

Examples

```
is.call(call) #-> FALSE: Functions are NOT calls

## set up a function call to round with argument 10.5
cl <- call("round", 10.5)
is.call(cl)# TRUE
cl
## such a call can also be evaluated.
eval(cl)# [1] 10

A <- 10.5
call("round", A)        # round(10.5)
call("round", quote(A)) # round(A)
f <- "round"
call(f, quote(A))       # round(A)
## if we want to supply a function we need to use as.call or similar
f <- round
## Not run: call(f, quote(A))  # error: first arg must be character
(g <- as.call(list(f, quote(A))))
eval(g)
## alternatively but less transparently
g <- list(f, quote(A))
mode(g) <- "call"
g
eval(g)
## see also the examples in the help for do.call
```

---

callCC *Call With Current Continuation*

Description

A downward-only version of Scheme's call with current continuation.

Usage

```
callCC(fun)
```

Arguments

fun	function of one argument, the exit procedure.

Details

callCC provides a non-local exit mechanism that can be useful for early termination of a computation. callCC calls fun with one argument, an *exit function*. The exit function takes a single argument, the intended return value. If the body of fun calls the exit function then the call to callCC immediately returns, with the value supplied to the exit function as the value returned by callCC.

Author(s)

Luke Tierney

Examples

```
# The following all return the value 1
callCC(function(k) 1)
callCC(function(k) k(1))
callCC(function(k) {k(1); 2})
callCC(function(k) repeat k(1))
```

---

capabilities *Report Capabilities of this Build of R*

Description

Report on the optional features which have been compiled into this build of R.

Usage

```
capabilities(what = NULL)
```

Arguments

what	character vector or NULL, specifying required components. NULL implies that all are required.

Value

A named logical vector. Current components are

jpeg	Is the jpeg function operational?
png	Is the png function operational?
tiff	Is the tiff function operational?
tcltk	Is the **tcltk** package operational? Note that to make use of Tk you will almost always need to check that "X11" is also available.

X11	Are the X11 graphics device and the X11-based data editor available? This loads the X11 module if not already loaded, and checks that the default display can be contacted unless a X11 device has already been used.
aqua	Are the R.app GUI components and the quartz function operational? Only on some Mac OS X builds. Note that this is distinct from .Platform$GUI == "AQUA", which is true when using the Mac R.app console.
http/ftp	Are url and the internal method for download.file available?
sockets	Are make.socket and related functions available?
libxml	Is there support for integrating libxml with the R event loop?
fifo	are FIFO connections supported?
cledit	Is command-line editing available in the current R session? This is false in non-interactive sessions. It will be true for the command-line interface if readline support has been compiled in and '--no-readline' was *not* used when R was invoked.
iconv	is internationalization conversion via iconv supported?
NLS	is there Natural Language Support (for message translations)?
profmem	is there support for memory profiling?
cairo	is there support for type="Cairo" in X11, png, jpeg, tiff and bmp, and for the svg, cairo_pdf and cairo_ps devices?

Note to Mac OS X users

Capabilities "jpeg", "png" and "tiff" refer to the X11-based versions of these devices. If capabilities("aqua") is true, then these devices with type="quartz" will be available, and out-of-the-box will be the default type. Thus for example the tiff device will be available if capabilities("aqua") || capabilities("tiff") if the defaults are unchanged.

See Also

.Platform

Examples

```
capabilities()

if(!capabilities("http/ftp"))
   warning("internal download.file() is not available")

## See also the examples for 'connections'.
```

---

cat *Concatenate and Print*

Description

Outputs the objects, concatenating the representations. cat performs much less conversion than print.

Usage

```
cat(... , file = "", sep = " ", fill = FALSE, labels = NULL,
    append = FALSE)
```

Arguments

`...`	R objects (see 'Details' for the types of objects allowed).
`file`	A connection, or a character string naming the file to print to. If `""` (the default), `cat` prints to the standard output connection, the console unless redirected by `sink`. If it is `"|cmd"`, the output is piped to the command given by 'cmd', by opening a pipe connection.
`sep`	a character vector of strings to append after each element.
`fill`	a logical or (positive) numeric controlling how the output is broken into successive lines. If `FALSE` (default), only newlines created explicitly by '`"\n"`' are printed. Otherwise, the output is broken into lines with print width equal to the option `width` if `fill` is `TRUE`, or the value of `fill` if this is numeric. Non-positive `fill` values are ignored, with a warning.
`labels`	character vector of labels for the lines printed. Ignored if `fill` is `FALSE`.
`append`	logical. Only used if the argument `file` is the name of file (and not a connection or `"|cmd"`). If `TRUE` output will be appended to `file`; otherwise, it will overwrite the contents of `file`.

Details

`cat` is useful for producing output in user-defined functions. It converts its arguments to character vectors, concatenates them to a single character vector, appends the given `sep=` string(s) to each element and then outputs them.

No linefeeds are output unless explicitly requested by '`"\n"`' or if generated by filling (if argument `fill` is `TRUE` or numeric.)

If `file` is a connection and open for writing it is written from its current position. If it is not open, it is opened for the duration of the call in `"wt"` mode and then closed again.

Currently only atomic vectors (and so not lists) and names are handled. Character strings are output 'as is' (unlike `print.default` which escapes non-printable characters and backslash — use `encodeString` if you want to output encoded strings using `cat`). Other types of R object should be converted (e.g. by `as.character` or `format`) before being passed to `cat`.

`cat` converts numeric/complex elements in the same way as `print` (and not in the same way as `as.character` which is used by the S equivalent), so `options` `"digits"` and `"scipen"` are relevant. However, it uses the minimum field width necessary for each element, rather than the same field width for all elements.

Value

None (invisible `NULL`).

Note

Despite its name and earlier documentation, `sep` is a vector of terminators rather than separators, being output after every vector element (including the last). Entries are recycled as needed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`print`, `format`, and `paste` which concatenates into a string.

Examples

```
iter <- stats::rpois(1, lambda=10)
## print an informative message
cat("iteration = ", iter <- iter + 1, "\n")

## 'fill' and label lines:
cat(paste(letters, 100* 1:26), fill = TRUE,
    labels = paste("{",1:10,"}:",sep=""))
```

---

cbind — *Combine R Objects by Rows or Columns*

Description

Take a sequence of vector, matrix or data frames arguments and combine by *c*olumns or *r*ows, respectively. These are generic functions with methods for other R classes.

Usage

```
cbind(..., deparse.level = 1)
rbind(..., deparse.level = 1)
```

Arguments

`...`	vectors or matrices. These can be given as named arguments. Other R objects will be coerced as appropriate: see sections 'Details' and 'Value'. (For the `"data.frame"` method of `cbind` these can be further arguments to `data.frame` such as `stringsAsFactors`.)
`deparse.level`	integer controlling the construction of labels in the case of non-matrix-like arguments (for the default method): `deparse.level = 0` constructs no labels; the default, `deparse.level = 1 or 2` constructs labels from the argument names, see the 'Value' section below.

Details

The functions `cbind` and `rbind` are S3 generic, with methods for data frames. The data frame method will be used if at least one argument is a data frame and the rest are vectors or matrices. There can be other methods; in particular, there is one for time series objects. See the section on 'Dispatch' for how the method to be used is selected.

In the default method, all the vectors/matrices must be atomic (see `vector`) or lists. Expressions are not allowed. Language objects (such as formulae and calls) and pairlists will be coerced to lists:

other objects (such as names and external pointers) will be included as elements in a list result. Any classes the inputs might have are discarded (in particular, factors are replaced by their internal codes).

If there are several matrix arguments, they must all have the same number of columns (or rows) and this will be the number of columns (or rows) of the result. If all the arguments are vectors, the number of columns (rows) in the result is equal to the length of the longest vector. Values in shorter arguments are recycled to achieve this length (with a `warning` if they are recycled only *fractionally*).

When the arguments consist of a mix of matrices and vectors the number of columns (rows) of the result is determined by the number of columns (rows) of the matrix arguments. Any vectors have their values recycled or subsetted to achieve this length.

For `cbind` (`rbind`), vectors of zero length (including `NULL`) are ignored unless the result would have zero rows (columns), for S compatibility. (Zero-extent matrices do not occur in S3 and are not ignored in R.)

Value

For the default method, a matrix combining the `...` arguments column-wise or row-wise. (Exception: if there are no inputs or all the inputs are `NULL`, the value is `NULL`.)

The type of a matrix result determined from the highest type of any of the inputs in the hierarchy raw < logical < integer < real < complex < character < list .

For `cbind` (`rbind`) the column (row) names are taken from the `colnames` (`rownames`) of the arguments if these are matrix-like. Otherwise from the names of the arguments or where those are not supplied and `deparse.level > 0`, by deparsing the expressions given, for `deparse.level = 1` only if that gives a sensible name (a 'symbol', see `is.symbol`).

For `cbind` row names are taken from the first argument with appropriate names: rownames for a matrix, or names for a vector of length the number of rows of the result.

For `rbind` column names are taken from the first argument with appropriate names: colnames for a matrix, or names for a vector of length the number of columns of the result.

Data frame methods

The `cbind` data frame method is just a wrapper for `data.frame(..., check.names = FALSE)` . This means that it will split matrix columns in data frame arguments, and convert character columns to factors unless `stringsAsFactors = FALSE` is specified.

The `rbind` data frame method first drops all zero-column and zero-row arguments. (If that leaves none, it returns the first argument with columns otherwise a zero-column zero-row data frame.) It then takes the classes of the columns from the first data frame, and matches columns by name (rather than by position). Factors have their levels expanded as necessary (in the order of the levels of the levelsets of the factors encountered) and the result is an ordered factor if and only if all the components were ordered factors. (The last point differs from S-PLUS.) Old-style categories (integer vectors with levels) are promoted to factors.

Dispatch

The method dispatching is *not* done via `UseMethod()` , but by C-internal dispatching. Therefore there is no need for, e.g., `rbind.default`.

The dispatch algorithm is described in the source file ('.../src/main/bind.c') as

1. For each argument we get the list of possible class memberships from the class attribute.

2. We inspect each class in turn to see if there is an applicable method.
3. If we find an applicable method we make sure that it is identical to any method determined for prior arguments. If it is identical, we proceed, otherwise we immediately drop through to the default code.

If you want to combine other objects with data frames, it may be necessary to coerce them to data frames first. (Note that this algorithm can result in calling the data frame method if all the arguments are either data frames or vectors, and this will result in the coercion of character vectors to factors.)

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`c` to combine vectors (and lists) as vectors, `data.frame` to combine vectors and matrices as a data frame.

Examples

```
m <- cbind(1, 1:7) # the '1' (= shorter vector) is recycled
m
m <- cbind(m, 8:14)[, c(1, 3, 2)] # insert a column
m
cbind(1:7, diag(3))# vector is subset -> warning

cbind(0, rbind(1, 1:3))
cbind(I=0, X=rbind(a=1, b=1:3))  # use some names
xx <- data.frame(I=rep(0,2))
cbind(xx, X=rbind(a=1, b=1:3))   # named differently

cbind(0, matrix(1, nrow=0, ncol=4))#> Warning (making sense)
dim(cbind(0, matrix(1, nrow=2, ncol=0)))#-> 2 x 1

## deparse.level
dd <- 10
rbind(1:4, c=2, "a++" = 10, dd, deparse.level=0)# middle 2 rownames
rbind(1:4, c=2, "a++" = 10, dd, deparse.level=1)# 3 rownames (default)
rbind(1:4, c=2, "a++" = 10, dd, deparse.level=2)# 4 rownames
```

---

char.expand *Expand a String with Respect to a Target Table*

Description

Seeks a unique match of its first argument among the elements of its second. If successful, it returns this element; otherwise, it performs an action specified by the third argument.

Usage

```
char.expand(input, target, nomatch = stop("no match"))
```

Arguments

input	a character string to be expanded.
target	a character vector with the values to be matched against.
nomatch	an R expression to be evaluated in case expansion was not possible.

Details

This function is particularly useful when abbreviations are allowed in function arguments, and need to be uniquely expanded with respect to a target table of possible values.

Value

A length-one character vector, one of the elements of target (unless nomatch is changed to be a non-error, when it can be a zero-length character string).

See Also

charmatch and pmatch for performing partial string matching.

Examples

```
locPars <- c("mean", "median", "mode")
char.expand("me", locPars, warning("Could not expand!"))
char.expand("mo", locPars)
```

---

character *Character Vectors*

Description

Create or test for objects of type "character".

Usage

```
character(length = 0)
as.character(x, ...)
is.character(x)
```

Arguments

length	desired length.
x	object to be coerced or tested.
...	further arguments passed to or from other methods.

Details

as.character and is.character are generic: you can write methods to handle specific classes of objects, see InternalMethods. Further, for as.character the default method calls as.vector, so dispatch is first on methods for as.character and then for methods for as.vector.

as.character represents real and complex numbers to 15 significant digits (technically the compiler's setting of the ISO C constant DBL_DIG, which will be 15 on machines supporting IEC60559 arithmetic according to the C99 standard). This ensures that all the digits in the result will be reliable (and not the result of representation error), but does mean that conversion to character and back to numeric may change the number. If you want to convert numbers to character with the maximum possible precision, use format.

Value

character creates a character vector of the specified length. The elements of the vector are all equal to "".

as.character attempts to coerce its argument to character type; like as.vector it strips attributes including names. For lists it deparses the elements individually, except that it extracts the first element of length-one character vectors.

is.character returns TRUE or FALSE depending on whether its argument is of character type or not.

Note

as.character truncates components of language objects to 500 characters (was about 70 before 1.3.1).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

paste, substr and strsplit for character concatenation and splitting, chartr for character translation and casefolding (e.g., upper to lower case) and sub, grep etc for string matching and substitutions. Note that help.search(keyword = "character") gives even more links. deparse, which is normally preferable to as.character for language objects.

Examples

```
form <- y ~ a + b + c
as.character(form)  ## length 3
deparse(form)       ## like the input

a0 <- 11/999          # has a repeating decimal representation
(a1 <- as.character(a0))
format(a0, digits=16) # shows one more digit
a2 <- as.numeric(a1)
a2 - a0               # normally around -1e-17
as.character(a2)      # normally different from a1
print(c(a0, a2), digits = 16)
```

charmatch *Partial String Matching*

Description

charmatch seeks matches for the elements of its first argument among those of its second.

Usage

```
charmatch(x, table, nomatch = NA_integer_)
```

Arguments

x	the values to be matched: converted to a character vector by as.character.
table	the values to be matched against: converted to a character vector.
nomatch	the (integer) value to be returned at non-matching positions.

Details

Exact matches are preferred to partial matches (those where the value to be matched has an exact match to the initial part of the target, but the target is longer).

If there is a single exact match or no exact match and a unique partial match then the index of the matching value is returned; if multiple exact or multiple partial matches are found then 0 is returned and if no match is found then nomatch is returned.

NA values are treated as the string constant "NA".

Value

An integer vector of the same length as x, giving the indices of the elements in table which matched, or nomatch.

Author(s)

This function is based on a C function written by Terry Therneau.

See Also

pmatch, match.

grep or regexpr for more general (regexp) matching of strings.

Examples

```
charmatch("", "")                            # returns 1
charmatch("m",   c("mean", "median", "mode")) # returns 0
charmatch("med", c("mean", "median", "mode")) # returns 2
```

chartr *Character Translation and Casefolding*

Description

Translate characters in character vectors, in particular from upper to lower case or vice versa.

Usage

```
chartr(old, new, x)
tolower(x)
toupper(x)
casefold(x, upper = FALSE)
```

Arguments

x	a character vector, or an object that can be coerced to character by as.character.
old	a character string specifying the characters to be translated.
new	a character string specifying the translations.
upper	logical: translate to upper or lower case?.

Details

chartr translates each character in x that is specified in old to the corresponding character specified in new. Ranges are supported in the specifications, but character classes and repeated characters are not. If old contains more characters than new, an error is signaled; if it contains fewer characters, the extra characters at the end of new are ignored.

tolower and toupper convert upper-case characters in a character vector to lower-case, or vice versa. Non-alphabetic characters are left unchanged.

casefold is a wrapper for tolower and toupper provided for compatibility with S-PLUS.

Value

A character vector of the same length and with the same attributes as x (after possible coercion).

Elements of the result will be have the encoding declared as that of the current locale (see Encoding if the corresponding input had a declared encoding and the current locale is either Latin-1 or UTF-8. The result will be in the current locale's encoding unless the corresponding input was in UTF-8, when it will be in UTF-8 when the system has Unicode wide characters.

See Also

sub and gsub for other substitutions in strings.

Examples

```
x <- "MiXeD cAsE 123"
chartr("iXs", "why", x)
chartr("a-cX", "D-Fw", x)
tolower(x)
toupper(x)

## "Mixed Case" Capitalizing - toupper( every first letter of a word ) :

.simpleCap <- function(x) {
    s <- strsplit(x, " ")[[1]]
    paste(toupper(substring(s, 1,1)), substring(s, 2),
          sep="", collapse=" ")
}
.simpleCap("the quick red fox jumps over the lazy brown dog")
## ->  [1] "The Quick Red Fox Jumps Over The Lazy Brown Dog"

## and the better, more sophisticated version:
capwords <- function(s, strict = FALSE) {
    cap <- function(s) paste(toupper(substring(s,1,1)),
                  {s <- substring(s,2); if(strict) tolower(s) else s},
                             sep = "", collapse = " " )
    sapply(strsplit(s, split = " "), cap, USE.NAMES = !is.null(names(s)))
}
capwords(c("using AIC for model selection"))
## ->  [1] "Using AIC For Model Selection"
capwords(c("using AIC", "for MODEL selection"), strict=TRUE)
## ->  [1] "Using Aic"  "For Model Selection"
##                ^^^        ^^^^^
##               'bad'       'good'

## -- Very simple insecure crypto --
rot <- function(ch, k = 13) {
   p0 <- function(...) paste(c(...), collapse="")
   A <- c(letters, LETTERS, " '")
   I <- seq_len(k); chartr(p0(A), p0(c(A[-I], A[I])), ch)
}

pw <- "my secret pass phrase"
(crypw <- rot(pw, 13)) #-> you can send this off

## now ``decrypt'' :
rot(crypw, 54 - 13)# -> the original:
stopifnot(identical(pw, rot(crypw, 54 - 13)))
```

---

chol *The Choleski Decomposition*

Description

Compute the Choleski factorization of a real symmetric positive-definite square matrix.

Usage

```
chol(x, ...)

## Default S3 method:
chol(x, pivot = FALSE,  LINPACK = pivot, ...)
```

Arguments

x	an object for which a method exists. The default method applies to real symmetric, positive-definite matrices.
...	arguments to be based to or from methods.
pivot	Should pivoting be used?
LINPACK	logical. Should LINPACK be used in the non-pivoting case (for compatibility with R < 1.7.0)?

Details

`chol` is generic: the description here applies to the default method.

This is an interface to the LAPACK routine DPOTRF and the LINPACK routines DPOFA and DCHDC.

Note that only the upper triangular part of `x` is used, so that $R'R = x$ when `x` is symmetric.

If `pivot = FALSE` and `x` is not non-negative definite an error occurs. If `x` is positive semi-definite (i.e., some zero eigenvalues) an error will also occur, as a numerical tolerance is used.

If `pivot = TRUE`, then the Choleski decomposition of a positive semi-definite `x` can be computed. The rank of `x` is returned as `attr(Q, "rank")`, subject to numerical errors. The pivot is returned as `attr(Q, "pivot")`. It is no longer the case that `t(Q) %*% Q` equals `x`. However, setting `pivot <- attr(Q, "pivot")` and `oo <- order(pivot)`, it is true that `t(Q[, oo]) %*% Q[, oo]` equals `x`, or, alternatively, `t(Q) %*% Q` equals `x[pivot, pivot]`. See the examples.

Value

The upper triangular factor of the Choleski decomposition, i.e., the matrix R such that $R'R = x$ (see example).

If pivoting is used, then two additional attributes `"pivot"` and `"rank"` are also returned.

Warning

The code does not check for symmetry.

If `pivot = TRUE` and `x` is not non-negative definite then there will be a warning message but a meaningless result will occur. So only use `pivot = TRUE` when `x` is non-negative definite by construction.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide.* Philadelphia: SIAM Publications.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at http://www.netlib.org/lapack/lug/lapack_lug.html.

See Also

chol2inv for its *inverse* (without pivoting), backsolve for solving linear systems with upper triangular left sides.

qr, svd for related matrix factorizations.

Examples

```
( m <- matrix(c(5,1,1,3),2,2) )
( cm <- chol(m) )
t(cm) %*% cm  #-- = 'm'
crossprod(cm)  #-- = 'm'

# now for something positive semi-definite
x <- matrix(c(1:5, (1:5)^2), 5, 2)
x <- cbind(x, x[, 1] + 3*x[, 2])
m <- crossprod(x)
qr(m)$rank # is 2, as it should be

# chol() may fail, depending on numerical rounding:
# chol() unlike qr() does not use a tolerance.
try(chol(m))

(Q <- chol(m, pivot = TRUE)) # NB wrong rank here - see Warning section.
## we can use this by
pivot <- attr(Q, "pivot")
oo <- order(pivot)
t(Q[, oo]) %*% Q[, oo] # recover m

## now for a non-positive-definite matrix
( m <- matrix(c(5,-5,-5,3),2,2) )
try(chol(m))  # fails
try(chol(m, LINPACK=TRUE))   # fails
(Q <- chol(m, pivot = TRUE)) # warning
crossprod(Q)  # not equal to m
```

---

chol2inv *Inverse from Choleski (or QR) Decomposition*

Description

Invert a symmetric, positive definite square matrix from its Choleski decomposition. Equivalently, compute $(X'X)^{-1}$ from the (R part) of the QR decomposition of X.

Usage

```
chol2inv(x, size = NCOL(x), LINPACK = FALSE)
```

Arguments

x	a matrix. The first size columns of the upper triangle contain the Choleski decomposition of the matrix to be inverted.
size	the number of columns of x containing the Choleski decomposition.
LINPACK	logical. Should LINPACK be used (for compatibility with R < 1.7.0)?

Details

This is an interface to the LAPACK routine DPOTRI and the LINPACK routine DPODI.

Value

The inverse of the matrix whose Choleski decomposition was given.

References

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM. Available on-line at http://www.netlib.org/lapack/lug/lapack_lug.html.

See Also

chol, solve.

Examples

```
cma <- chol(ma  <- cbind(1, 1:3, c(1,3,7)))
ma %*% chol2inv(cma)
```

---

class *Object Classes*

Description

R possesses a simple generic function mechanism which can be used for an object-oriented style of programming. Method dispatch takes place based on the class of the first argument to the generic function.

Usage

```
class(x)
class(x) <- value
unclass(x)
inherits(x, what, which = FALSE)

oldClass(x)
oldClass(x) <- value
```

Arguments

x	a R object
what, value	a character vector naming classes. value can also be NULL.
which	logical affecting return value: see 'Details'.

Details

Many R objects have a `class` attribute, a character vector giving the names of the classes from which the object *inherits*. If the object does not have a class attribute, it has an implicit class, `"matrix"`, `"array"` or the result of `mode(x)` (except that integer vectors have implicit class `"integer"`). (Functions `oldClass` and `oldClass<-` get and set the attribute, which can also be done directly.)

When a generic function `fun` is applied to an object with class attribute `c("first", "second")`, the system searches for a function called `fun.first` and, if it finds it, applies it to the object. If no such function is found, a function called `fun.second` is tried. If no class name produces a suitable function, the function `fun.default` is used (if it exists). If there is no class attribute, the implicit class is tried, then the default method.

The function `class` prints the vector of names of classes an object inherits from. Correspondingly, `class<-` sets the classes an object inherits from. Assigning a zero-length vector or `NULL` removes the class attribute.

`unclass` returns (a copy of) its argument with its class attribute removed. (It is not allowed for objects which cannot be copied, namely environments and external pointers.)

`inherits` indicates whether its first argument inherits from any of the classes specified in the `what` argument. If `which` is `TRUE` then an integer vector of the same length as `what` is returned. Each element indicates the position in the `class(x)` matched by the element of `what`; zero indicates no match. If `which` is `FALSE` then `TRUE` is returned by `inherits` if any of the names in `what` match with any `class`.

Formal classes

An additional mechanism of *formal* classes is available in packages **methods** which is attached by default. For objects which have a formal class, its name is returned by `class` as a character vector of length one.

The replacement version of the function sets the class to the value provided. For classes that have a formal definition, directly replacing the class this way is strongly deprecated. The expression `as(object, value)` is the way to coerce an object to a particular class.

The analogue of `inherits` for formal classes is `is`.

Note

Functions `oldClass` and `oldClass<-` behave in the same way as functions of those names in S-PLUS 5/6, *but* in R `UseMethod` dispatches on the class as returned by `class` (with some interpolated classes: see the link) rather than `oldClass`. *However*, group generics dispatch on the `oldClass` for efficiency, and internal generics only dispatch on objects for which `is.object` is true.

See Also

`UseMethod`, `NextMethod`, 'group generic', 'internal generic'

Examples

```
x <- 10
class(x) # "numeric"
oldClass(x) # NULL
inherits(x, "a") #FALSE
class(x) <- c("a", "b")
```

```
inherits(x,"a") #TRUE
inherits(x, "a", TRUE) # 1
inherits(x, c("a", "b", "c"), TRUE) # 1 2 0
```

`col` *Column Indexes*

Description

Returns a matrix of integers indicating their column number in a matrix-like object, or a factor of column labels.

Usage

```
col(x, as.factor = FALSE)
```

Arguments

`x`	a matrix-like object, that is one with a two-dimensional `dim`.
`as.factor`	a logical value indicating whether the value should be returned as a factor of column labels (created if necessary) rather than as numbers.

Value

An integer (or factor) matrix with the same dimensions as `x` and whose `ij`-th element is equal to `j` (or the `j`-th column label).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`row` to get rows.

Examples

```
# extract an off-diagonal of a matrix
ma <- matrix(1:12, 3, 4)
ma[row(ma) == col(ma) + 1]

# create an identity 5-by-5 matrix
x <- matrix(0, nrow = 5, ncol = 5)
x[row(x) == col(x)] <- 1
```

`Colon` *Colon Operator*

Description

Generate regular sequences.

Usage

```
from:to
   a:b
```

Arguments

`from`	starting value of sequence.
`to`	(maximal) end value of the sequence.
`a, b`	`factor`s of same length.

Details

The binary operator `:` has two meanings: for factors `a:b` is equivalent to `interaction(a, b)` (but the levels are ordered and labelled differently).

For numeric arguments `from:to` is equivalent to `seq(from, to)`, and generates a sequence from `from` to `to` in steps of `1` or `1-`. Value `to` will be included if it differs from `from` by an integer up to a numeric fuzz of about `1e-7`.

Value

For numeric arguments, a numeric vector. This will be of type `integer` if `from` and `to` are both integers and representable in the integer type, otherwise of type `numeric`.

For factors, an unordered factor with levels labelled as `la:lb` and ordered lexicographically (that is, `lb` varies fastest).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.
(for numeric arguments: S does not have `:` for factors.)

See Also

`seq`.

As an alternative to using `:` for factors, `interaction`.

For `:` used in the formal representation of an interaction, see `formula`.

Examples

```
1:4
pi:6 # real
6:pi # integer

f1 <- gl(2,3); f1
f2 <- gl(3,2); f2
f1:f2 # a factor, the "cross"  f1 x f2
```

---

colSums *Form Row and Column Sums and Means*

Description

Form row and column sums and means for numeric arrays.

Usage

```
colSums (x, na.rm = FALSE, dims = 1)
rowSums (x, na.rm = FALSE, dims = 1)
colMeans(x, na.rm = FALSE, dims = 1)
rowMeans(x, na.rm = FALSE, dims = 1)
```

Arguments

x	an array of two or more dimensions, containing numeric, complex, integer or logical values, or a numeric data frame.
na.rm	logical. Should missing values (including NaN) be omitted from the calculations?
dims	Which dimensions are regarded as 'rows' or 'columns' to sum over. For row*, the sum or mean is over dimensions dims+1, ...; for col* it is over dimensions 1:dims.

Details

These functions are equivalent to use of apply with FUN = mean or FUN = sum with appropriate margins, but are a lot faster. As they are written for speed, they blur over some of the subtleties of NaN and NA. If na.rm = FALSE and either NaN or NA appears in a sum, the result will be one of NaN or NA, but which might be platform-dependent.

Value

A numeric or complex array of suitable size, or a vector if the result is one-dimensional. The dimnames (or names for a vector result) are taken from the original array.

If there are no values in a range to be summed over (after removing missing values with na.rm = TRUE), that component of the output is set to 0 (*Sums) or NA (*Means), consistent with sum and mean.

See Also

apply, rowsum

Examples

```
## Compute row and column sums for a matrix:
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
rowSums(x); colSums(x)
dimnames(x)[[1]] <- letters[1:8]
rowSums(x); colSums(x); rowMeans(x); colMeans(x)
x[] <- as.integer(x)
rowSums(x); colSums(x)
x[] <- x < 3
rowSums(x); colSums(x)
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
x[3, ] <- NA; x[4, 2] <- NA
rowSums(x); colSums(x); rowMeans(x); colMeans(x)
rowSums(x, na.rm = TRUE); colSums(x, na.rm = TRUE)
rowMeans(x, na.rm = TRUE); colMeans(x, na.rm = TRUE)

## an array
dim(UCBAdmissions)
rowSums(UCBAdmissions); rowSums(UCBAdmissions, dims = 2)
colSums(UCBAdmissions); colSums(UCBAdmissions, dims = 2)

## complex case
x <- cbind(x1 = 3 + 2i, x2 = c(4:1, 2:5) - 5i)
x[3, ] <- NA; x[4, 2] <- NA
rowSums(x); colSums(x); rowMeans(x); colMeans(x)
rowSums(x, na.rm = TRUE); colSums(x, na.rm = TRUE)
rowMeans(x, na.rm = TRUE); colMeans(x, na.rm = TRUE)
```

---

commandArgs *Extract Command Line Arguments*

Description

Provides access to a copy of the command line arguments supplied when this R session was invoked.

Usage

```
commandArgs(trailingOnly = FALSE)
```

Arguments

`trailingOnly` logical. Should only arguments after `--args` be returned?

Details

These arguments are captured before the standard R command line processing takes place. This means that they are the unmodified values. This is especially useful with the `--args` command-line flag to R, as all of the command line after that flag is skipped.

Value

A character vector containing the name of the executable and the user-supplied command line arguments. The first element is the name of the executable by which R was invoked. The exact form of this element is platform dependent: it may be the fully qualified name, or simply the last component (or basename) of the application, or for an embedded R it can be anything the programmer supplied.

If `trailingOnly = TRUE`, a character vector of those arguments (if any) supplied after `--args`.

See Also

`Startup` `BATCH`

Examples

```
commandArgs()
## Spawn a copy of this application as it was invoked,
## subject to shell quoting issues
## system(paste(commandArgs(), collapse=" "))
```

---

comment *Query or Set a 'Comment' Attribute*

Description

These functions set and query a *comment* attribute for any R objects. This is typically useful for `data.frame`s or model fits.

Contrary to other `attributes`, the `comment` is not printed (by `print` or `print.default`).

Assigning `NULL` or a zero-length character vector removes the comment.

Usage

```
comment(x)
comment(x) <- value
```

Arguments

`x`	any R object
`value`	a `character` vector, or `NULL`.

See Also

`attributes` and `attr` for other attributes.

Examples

```
x <- matrix(1:12, 3,4)
comment(x) <- c("This is my very important data from experiment #0234",
                "Jun 5, 1998")
x
comment(x)
```

`Comparison` *Relational Operators*

Description

Binary operators which allow the comparison of values in atomic vectors.

Usage

```
x < y
x > y
x <= y
x >= y
x == y
x != y
```

Arguments

`x, y`	atomic vectors, symbols, calls, or other objects for which methods have been written.

Details

The binary comparison operators are generic functions: methods can be written for them individually or via the `Ops`) group generic function. (See `Ops` for how dispatch is computed.)

Comparison of strings in character vectors is lexicographic within the strings using the collating sequence of the locale in use: see `locales`. The collating sequence of locales such as '`en_US`' is normally different from '`C`' (which should use ASCII) and can be surprising. Beware of making *any* assumptions about the collation order: e.g. in Estonian `Z` comes between `S` and `T`, and collation is not necessarily character-by-character – in Danish `aa` sorts as a single letter, after `z`. In Welsh `ng` may or may not be a single sorting unit: if it is it follows `g`. Some platforms may not respect the locale and always sort in numerical order of the bytes in an 8-bit locale, or in Unicode point order for a UTF-8 locale (and may not sort in the same order for the same language in different character sets). Collation of non-letters (spaces, punctuation signs, hyphens, fractions and so on) is even more problematic.

At least one of `x` and `y` must be an atomic vector, but if the other is a list R attempts to coerce it to the type of the atomic vector: this will succeed if the list is made up of elements of length one that can be coerced to the correct type.

If the two arguments are atomic vectors of different types, one is coerced to the type of the other, the (decreasing) order of precedence being character, complex, numeric, integer, logical and raw.

Missing values (`NA`) and `NaN` values are regarded as non-comparable even to themselves, so comparisons involving them will always result in `NA`. Missing values can also result when character strings are compared and one is not valid in the current collation locale.

Language objects such as symbols and calls are deparsed to character strings before comparison.

Value

A logical vector indicating the result of the element by element comparison. The elements of shorter vectors are recycled as necessary.

Objects such as arrays or time-series can be compared this way provided they are conformable.

S4 methods

These operators are members of the S4 `Compare` group generic, and so methods can be written for them individually as well as for the group generic (or the `Ops` group generic), with arguments `c(e1, e2)`.

Note

Do not use == and != for tests, such as in `if` expressions, where you must get a single `TRUE` or `FALSE`. Unless you are absolutely sure that nothing unusual can happen, you should use the `identical` function instead.

For numerical and complex values, remember == and != do not allow for the finite representation of fractions, nor for rounding error. Using `all.equal` with `identical` is almost always preferable. See the examples.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Collation of character strings is a complex topic. For an introduction see http://en.wikipedia.org/wiki/Collating_sequence. The *Unicode Collation Algorithm* (http://unicode.org/reports/tr10/) is likely to be increasing influential.

See Also

`factor` for the behaviour with factor arguments.

`Syntax` for operator precedence.

Examples

```
x <- stats::rnorm(20)
x < 1
x[x > 0]

x1 <- 0.5 - 0.3
x2 <- 0.3 - 0.1
x1 == x2                           # FALSE on most machines
identical(all.equal(x1, x2), TRUE) # TRUE everywhere


z <- c(32:126, 160:255) # range of most 8-bit charsets, Latin-1 in Unicode
x <- if(l10n_info()$MBCS) {
    intToUtf8(z, multiple = TRUE)
} else rawToChar(as.raw(z), multiple= TRUE)
## by number
writeLines(strwrap(paste(x, collapse=" "), width = 60))
## by locale collation
writeLines(strwrap(paste(sort(x), collapse=" "), width = 60))
```

`complex` *Complex Vectors*

Description

Basic functions which support complex arithmetic in R.

Usage

```
complex(length.out = 0, real = numeric(), imaginary = numeric(),
        modulus = 1, argument = 0)
as.complex(x, ...)
is.complex(x)

Re(z)
Im(z)
Mod(z)
Arg(z)
Conj(z)
```

Arguments

`length.out`	numeric. Desired length of the output vector, inputs being recycled as needed.
`real`	numeric vector.
`imaginary`	numeric vector.
`modulus`	numeric vector.
`argument`	numeric vector.
`x`	an object, probably of mode `complex`.
`z`	an object of mode `complex`, or one of a class for which a methods has been defined.
`...`	further arguments passed to or from other methods.

Details

Complex vectors can be created with `complex`. The vector can be specified either by giving its length, its real and imaginary parts, or modulus and argument. (Giving just the length generates a vector of complex zeroes.)

`as.complex` attempts to coerce its argument to be of complex type: like `as.vector` it strips attributes including names. All forms of `NA` and `NaN` are coerced to a complex `NA`, for which both the real and imaginary parts are `NA`.

Note that `is.complex` and `is.numeric` are never both `TRUE`.

The functions `Re`, `Im`, `Mod`, `Arg` and `Conj` have their usual interpretation as returning the real part, imaginary part, modulus, argument and complex conjugate for complex values. Modulus and argument are also called the *polar coordinates*. If $z = x + iy$ with real x and y, for $r = \texttt{Mod}(z) = \sqrt{x^2 + y^2}$, and $\phi = \texttt{Arg}(z)$, $x = r * \cos(\phi)$ and $y = r * \sin(\phi)$. They are all generic functions: methods can be defined for them individually or via the `Complex` group generic.

In addition, the elementary trigonometric, logarithmic and exponential functions are available for complex values.

S4 methods

`as.complex` is primitive and can have S4 methods set.

`Re`, `Im`, `Mod`, `Arg` and `Conj` constitute the S4 group generic `Complex` and so S4 methods can be set for them individually or via the group generic.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
require(graphics)

0i ^ (-3:3)

matrix(1i^ (-6:5), nrow=4) #- all columns are the same
0 ^ 1i # a complex NaN

## create a complex normal vector
z <- complex(real = stats::rnorm(100), imaginary = stats::rnorm(100))
## or also (less efficiently):
z2 <- 1:2 + 1i*(8:9)

## The Arg(.) is an angle:
zz <- (rep(1:4,len=9) + 1i*(9:1))/10
zz.shift <- complex(modulus = Mod(zz), argument= Arg(zz) + pi)
plot(zz, xlim=c(-1,1), ylim=c(-1,1), col="red", asp = 1,
     main = expression(paste("Rotation by ","  ", pi == 180^o)))
abline(h=0,v=0, col="blue", lty=3)
points(zz.shift, col="orange")
```

---

conditions *Condition Handling and Recovery*

Description

These functions provide a mechanism for handling unusual conditions, including errors and warnings.

Usage

```
tryCatch(expr, ..., finally)
withCallingHandlers(expr, ...)

signalCondition(cond)

simpleCondition(message, call = NULL)
simpleError    (message, call = NULL)
simpleWarning  (message, call = NULL)
simpleMessage  (message, call = NULL)
```

```
## S3 method for class 'condition':
as.character(x, ...)
## S3 method for class 'error':
as.character(x, ...)
## S3 method for class 'condition':
print(x, ...)
## S3 method for class 'restart':
print(x, ...)

conditionCall(c)
## S3 method for class 'condition':
conditionCall(c)
conditionMessage(c)
## S3 method for class 'condition':
conditionMessage(c)

withRestarts(expr, ...)

computeRestarts(cond = NULL)
findRestart(name, cond = NULL)
invokeRestart(r, ...)
invokeRestartInteractively(r)

isRestart(x)
restartDescription(r)
restartFormals(r)

.signalSimpleWarning(msg, call)
.handleSimpleError(h, msg, call)
```

Arguments

`c`	a condition object.
`call`	call expression.
`cond`	a condition object.
`expr`	expression to be evaluated.
`finally`	expression to be evaluated before returning or exiting.
`h`	function.
`message`	character string.
`msg`	character string.
`name`	character string naming a restart.
`r`	restart object.
`x`	object.
`...`	additional arguments; see details below.

Details

The condition system provides a mechanism for signaling and handling unusual conditions, including errors and warnings. Conditions are represented as objects that contain information about the

condition that occurred, such as a message and the call in which the condition occurred. Currently conditions are S3-style objects, though this may eventually change.

Conditions are objects inheriting from the abstract class `condition`. Errors and warnings are objects inheriting from the abstract subclasses `error` and `warning`. The class `simpleError` is the class used by `stop` and all internal error signals. Similarly, `simpleWarning` is used by `warning`, and `simpleMessage` is used by `message`. The constructors by the same names take a string describing the condition as argument and an optional call. The functions `conditionMessage` and `conditionCall` are generic functions that return the message and call of a condition.

Conditions are signaled by `signalCondition`. In addition, the `stop` and `warning` functions have been modified to also accept condition arguments.

The function `tryCatch` evaluates its expression argument in a context where the handlers provided in the `...` argument are available. The `finally` expression is then evaluated in the context in which `tryCatch` was called; that is, the handlers supplied to the current `tryCatch` call are not active when the `finally` expression is evaluated.

Handlers provided in the `...` argument to `tryCatch` are established for the duration of the evaluation of `expr`. If no condition is signaled when evaluating `expr` then `tryCatch` returns the value of the expression.

If a condition is signaled while evaluating `expr` then established handlers are checked, starting with the most recently established ones, for one matching the class of the condition. When several handlers are supplied in a single `tryCatch` then the first one is considered more recent than the second. If a handler is found then control is transferred to the `tryCatch` call that established the handler, the handler found and all more recent handlers are disestablished, the handler is called with the condition as its argument, and the result returned by the handler is returned as the value of the `tryCatch` call.

Calling handlers are established by `withCallingHandlers`. If a condition is signaled and the applicable handler is a calling handler, then the handler is called by `signalCondition` in the context where the condition was signaled but with the available handlers restricted to those below the handler called in the handler stack. If the handler returns, then the next handler is tried; once the last handler has been tried, `signalCondition` returns `NULL`.

User interrupts signal a condition of class `interrupt` that inherits directly from class `condition` before executing the default interrupt action.

Restarts are used for establishing recovery protocols. They can be established using `withRestarts`. One pre-established restart is an `abort` restart that represents a jump to top level.

`findRestart` and `computeRestarts` find the available restarts. `findRestart` returns the most recently established restart of the specified name. `computeRestarts` returns a list of all restarts. Both can be given a condition argument and will then ignore restarts that do not apply to the condition.

`invokeRestart` transfers control to the point where the specified restart was established and calls the restart's handler with the arguments, if any, given as additional arguments to `invokeRestart`. The restart argument to `invokeRestart` can be a character string, in which case `findRestart` is used to find the restart.

New restarts for `withRestarts` can be specified in several ways. The simplest is in `name=function` form where the function is the handler to call when the restart is invoked. Another simple variant is as `name=string` where the string is stored in the `description` field of the restart object returned by `findRestart`; in this case the handler ignores its arguments and returns `NULL`. The most flexible form of a restart specification is as a list that can include several fields, including `handler`, `description`, and `test`. The `test` field should contain a function

of one argument, a condition, that returns TRUE if the restart applies to the condition and FALSE if it does not; the default function returns TRUE for all conditions.

One additional field that can be specified for a restart is interactive. This should be a function of no arguments that returns a list of arguments to pass to the restart handler. The list could be obtained by interacting with the user if necessary. The function invokeRestartInteractively calls this function to obtain the arguments to use when invoking the restart. The default interactive method queries the user for values for the formal arguments of the handler function.

.signalSimpleWarning and .handleSimpleError are used internally and should not be called directly.

References

The tryCatch mechanism is similar to Java error handling. Calling handlers are based on Common Lisp and Dylan. Restarts are based on the Common Lisp restart mechanism.

See Also

stop and warning signal conditions, and try is essentially a simplified version of tryCatch.

Examples

```
tryCatch(1, finally=print("Hello"))
e <- simpleError("test error")
## Not run:
 stop(e)
 tryCatch(stop(e), finally=print("Hello"))
 tryCatch(stop("fred"), finally=print("Hello"))
## End(Not run)
tryCatch(stop(e), error = function(e) e, finally=print("Hello"))
tryCatch(stop("fred"),  error = function(e) e, finally=print("Hello"))
withCallingHandlers({ warning("A"); 1+2 }, warning = function(w) {})
## Not run:
 { withRestarts(stop("A"), abort = function() {}); 1 }
## End(Not run)
withRestarts(invokeRestart("foo", 1, 2), foo = function(x, y) {x + y})
```

---

conflicts *Search for Masked Objects on the Search Path*

Description

conflicts reports on objects that exist with the same name in two or more places on the search path, usually because an object in the user's workspace or a package is masking a system object of the same name. This helps discover unintentional masking.

Usage

```
conflicts(where = search(), detail = FALSE)
```

Arguments

where	A subset of the search path, by default the whole search path.
detail	If TRUE, give the masked or masking functions for all members of the search path.

Value

If detail=FALSE, a character vector of masked objects. If detail=TRUE, a list of character vectors giving the masked or masking objects in that member of the search path. Empty vectors are omitted.

Examples

```
lm <- 1:3
conflicts(, TRUE)
## gives something like
# $.GlobalEnv
# [1] "lm"
#
# $package:base
# [1] "lm"

## Remove things from your "workspace" that mask others:
remove(list = conflicts(detail=TRUE)$.GlobalEnv)
```

---

connections *Functions to Manipulate Connections*

Description

Functions to create, open and close connections.

Usage

```
file(description = "", open = "", blocking = TRUE,
     encoding = getOption("encoding"))

url(description, open = "", blocking = TRUE,
    encoding = getOption("encoding"))

gzfile(description, open = "", encoding = getOption("encoding"),
       compression = 6)

bzfile(description, open = "", encoding = getOption("encoding"))

unz(description, filename, open = "",
    encoding = getOption("encoding"))

pipe(description, open = "", encoding = getOption("encoding"))

fifo(description, open = "", blocking = FALSE,
```

```
     encoding = getOption("encoding"))

socketConnection(host = "localhost", port, server = FALSE,
                 blocking = FALSE, open = "a+",
                 encoding = getOption("encoding"))

open(con, ...)
## S3 method for class 'connection':
open(con, open = "r", blocking = TRUE, ...)

close(con, ...)
## S3 method for class 'connection':
close(con, type = "rw", ...)

flush(con)

isOpen(con, rw = "")
isIncomplete(con)
```

Arguments

`description`	character string. A description of the connection: see 'Details'.
`open`	character. A description of how to open the connection (if at all). See 'Details' for possible values.
`blocking`	logical. See the 'Blocking' section below.
`encoding`	The name of the encoding to be used. See the 'Encoding' section below.
`compression`	integer in 0–9. The amount of compression to be applied when writing, from none to maximal. The default is a good space/time compromise.
`filename`	a filename within a zip file.
`host`	character. Host name for port.
`port`	integer. The TCP port number.
`server`	logical. Should the socket be a client or a server?
`con`	a connection.
`type`	character. Currently ignored.
`rw`	character. Empty or `"read"` or `"write"`, partial matches allowed.
`...`	arguments passed to or from other methods.

Details

The first eight functions create connections. By default the connection is not opened (except for `socketConnection`), but may be opened by setting a non-empty value of argument `open`.

For `file` the description is either a path to the file to be opened or a complete URL, or `""` (the default) or `"stdin"` or `"clipboard"` (see below).

For `url` the description is a complete URL, including scheme (such as `http://`, `ftp://` or `file://`).

For `gzfile` the description is the path to a file that is compressed by `gzip`: it can also opened uncompressed files.

For `bzfile` the description is the path to a file that is compressed by `bzip2`.

unz reads (only) single files within zip files, in binary mode. The description is the full path to the zip file, with '.zip' extension if required.

For pipe the description is the command line to be piped to or from.

For fifo the description is the path of the fifo.

file allows description="stdin" to refer to the C-level stdin of the process (which need not be connected to anything in a console or embedded version of R), provided the C99 function fdopen is supported on the platform.

gzfile and bzfile open the actual file in binary mode and so no translations are done if the original file was a text file. (See gzcon for a way to add compression to non-file connections such as URLs.)

All platforms support file, gzfile, bzfile, unz and url("file://") connections. The other types may be partially implemented or not implemented at all. (They do work on most Unix platforms, and all but fifo on Windows.)

Proxies can be specified for url connections: see download.file.

open, close and seek are generic functions: the following applies to the methods relevant to connections.

open opens a connection. In general functions using connections will open them if they are not open, but then close them again, so to leave a connection open call open explicitly.

Possible values for the mode open to open a connection are

"r" or "rt" Open for reading in text mode.

"w" or "wt" Open for writing in text mode.

"a" or "at" Open for appending in text mode.

"rb" Open for reading in binary mode.

"wb" Open for writing in binary mode.

"ab" Open for appending in binary mode.

"r+", "r+b" Open for reading and writing.

"w+", "w+b" Open for reading and writing, truncating file initially.

"a+", "a+b" Open for reading and appending.

Not all modes are applicable to all connections: for example URLs can only be opened for reading. Only file and socket connections can be opened for reading and writing/appending. For many connections there is little or no difference between text and binary modes, but there is for file-like connections on Windows, and pushBack is text-oriented and is only allowed on connections open for reading in text mode. If a file or fifo is created on a Unix-alike, its permissions will be the maximal allowed by the current setting of umask (see Sys.umask).

gzfile connections are an exception, as the file always has to be opened in binary mode. Thus modes such as r are binary, and rt is needed to have a text-mode connection.

close closes and destroys a connection. Note that this will happen automatically in due course if there is no R object referring to the connection.

A maximum of 128 connections can be allocated (not necessarily open) at any one time. Three of these are pre-allocated (see stdout). The OS will impose limits on the numbers of connections of various types, but these are usually larger than 125.

flush flushes the output stream of a connection open for write/append (where implemented).

If for a file or fifo connection the description is "", the file/fifo is immediately opened (in "w+" mode unless open="w+b" is specified) and unlinked from the file system. This provides a temporary file/fifo to write to and then read from.

A note on `file://` URLs. The most general form (from RFC1738) is `file://host/path/to/file`, but R only accepts the form with an empty `host` field referring to the local machine. This is then `file:///path/to/file`, where `path/to/file` is relative to `/`. So although the third slash is strictly part of the specification not part of the path, this can be regarded as a way to specify the file '/path/to/file'. It is not possible to specify a relative path using a file URL. Also, no attempt is made to decode an encoded URL: call `URLdecode` if necessary.

Note that `https://` connections are not supported.

Value

`file`, `pipe`, `fifo`, `url`, `gzfile`, `bzfile`, `unz` and `socketConnection` return a connection object which inherits from class `"connection"` and has a first more specific class.

`isOpen` returns a logical value, whether the connection is currently open.

`isIncomplete` returns a logical value, whether last read attempt was blocked, or for an output text connection whether there is unflushed output.

Encoding

The encoding of the input/output stream of a connection in *text* mode can be specified by name, in the same way as it would be given to `iconv`: see that help page for how to find out what names are recognized on your platform. Additionally, `""` and `"native.enc"` both mean the 'native' encoding, that is the internal encoding of the current locale and hence no translation is done. Not all builds of R support this, and if yours does not, specifying a non-default encoding will give an error when the connection is opened.

Re-encoding only works for connections in text mode.

The encoding `"UCS-2LE"` is treated specially, as it is the appropriate value for Windows 'Unicode' text files. If the first two bytes are the Byte Order Mark `0xFFFE` then these are removed as most implementations of `iconv` do not accept BOMs. Note that some implementations will handle BOMs using encoding `"UCS-2"` but many will not.

Exactly what happens when the requested translation cannot be done is in general undocumented. Requesting a conversion that is not supported is an error, reported when the connection is opened. On output the result is likely to be that up to the error, with a warning. On input, it will most likely be all or some of the input up to the error.

Blocking

The default condition for all but fifo and socket connections is to be in blocking mode. In that mode, functions do not return to the R evaluator until they are complete. In non-blocking mode, operations return as soon as possible, so on input they will return with whatever input is available (possibly none) and for output they will return whether or not the write succeeded.

The function `readLines` behaves differently in respect of incomplete last lines in the two modes: see its help page.

Even when a connection is in blocking mode, attempts are made to ensure that it does not block the event loop and hence the operation of GUI parts of R. These do not always succeed, and the whole process will be blocked during a DNS lookup on Unix, for example.

Most blocking operations on URLs and sockets are subject to the timeout set by `options("timeout")`. Note that this is a timeout for no response at all, not for the whole operation. The timeout is set at the time the connection is opened (more precisely, when the last connection of that type – `http:`, `ftp:` or socket – was opened).

Fifos

Fifos default to non-blocking. That follows S version 4 and is probably most natural, but it does have some implications. In particular, opening a non-blocking fifo connection for writing (only) will fail unless some other process is reading on the fifo.

Opening a fifo for both reading and writing (in any mode: one can only append to fifos) connects both sides of the fifo to the R process, and provides an similar facility to `file()`.

Clipboard

`file` can also be used with `description = "clipboard"` in mode `"r"` only. This reads the X11 primary selection (see http://standards.freedesktop.org/clipboards-spec/clipboards-latest.txt), which can also be specified as `"X11_primary"` and the secondary selection as `"X11_secondary"`. On most systems the clipboard selection (that used by 'Copy' from an 'Edit' menu) can be specified as `"X11_clipboard"`.

When a clipboard is opened for reading, the contents are immediately copied to internal storage in the connection.

Unix users wishing to *write* to one of the selections may be able to do so via `xclip` (http://people.debian.org/~kims/xclip/), for example by `pipe("xclip -i", "w")` for the primary selection.

MacOS X users can use `pipe("pbpaste")` and `pipe("pbcopy", "w")` to read from and write to that system's clipboard.

Note

R's connections are modelled on those in S version 4 (see Chambers, 1998). However R goes well beyond the S model, for example in output text connections and URL, `gzfile`, `bzfile` and socket connections.

The default mode in R is `"r"` except for socket connections. This differs from S, where it is the equivalent of `"r+"`, known as `"*"`.

On (rare) platforms where `vsnprintf` does not return the needed length of output there is a 100,000 character output limit on the length of line for `fifo`, `gzfile` and `bzfile` connections: longer lines will be truncated with a warning.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language.* Springer.

See Also

`textConnection`, `seek`, `showConnections`, `pushBack`.

Functions making direct use of connections are `readLines`, `readBin`, `readChar`, `writeLines`, `writeBin`, `writeChar`, `cat`, `sink`, `scan`, `parse`, `read.dcf`, `load`, `save`, `dput` and `dump`.

`capabilities` to see if `url`, `fifo` and `socketConnection` are supported by this build of R.

`gzcon` to wrap gzip (de)compression around a connection.

Examples

```
zz <- file("ex.data", "w")  # open an output file connection
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
cat("One more line\n", file = zz)
close(zz)
readLines("ex.data")
unlink("ex.data")

zz <- gzfile("ex.gz", "w")  # compressed file
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
readLines(zz <- gzfile("ex.gz"))
close(zz)
unlink("ex.gz")

zz <- bzfile("ex.bz2", "w")  # bzip2-ed file
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
print(readLines(zz <- bzfile("ex.bz2")))
close(zz)
unlink("ex.bz2")

## An example of a file open for reading and writing
Tfile <- file("test1", "w+")
c(isOpen(Tfile, "r"), isOpen(Tfile, "w")) # both TRUE
cat("abc\ndef\n", file=Tfile)
readLines(Tfile)
seek(Tfile, 0, rw="r") # reset to beginning
readLines(Tfile)
cat("ghi\n", file=Tfile)
readLines(Tfile)
close(Tfile)
unlink("test1")

## We can do the same thing with an anonymous file.
Tfile <- file()
cat("abc\ndef\n", file=Tfile)
readLines(Tfile)
close(Tfile)

## fifo example -- may fail, e.g. on Cygwin, even with OS support for fifos
if(capabilities("fifo")) {
  zz <- fifo("foo-fifo", "w+")
  writeLines("abc", zz)
  print(readLines(zz))
  close(zz)
  unlink("foo-fifo")
}


## Unix examples of use of pipes

# read listing of current directory
readLines(pipe("ls -1"))

# remove trailing commas. Suppose
```

```
## Not run:
% cat data2
450, 390, 467, 654,  30, 542, 334, 432, 421,
357, 497, 493, 550, 549, 467, 575, 578, 342,
446, 547, 534, 495, 979, 479
## End(Not run)
# Then read this by
scan(pipe("sed -e s/,$// data2_"), sep=",")


# convert decimal point to comma in output: see also write.table
# both R strings and (probably) the shell need \ doubled
zz <- pipe(paste("sed s/\\\\./,/ >", "outfile"), "w")
cat(format(round(stats::rnorm(48), 4)), fill=70, file = zz)
close(zz)
file.show("outfile", delete.file=TRUE)

## example for a machine running a finger daemon

con <- socketConnection(port = 79, blocking = TRUE)
writeLines(paste(system("whoami", intern=TRUE), "\r", sep=""), con)
gsub(" *$", "", readLines(con))
close(con)


## Not run:
## two R processes communicating via non-blocking sockets
# R process 1
con1 <- socketConnection(port = 6011, server=TRUE)
writeLines(LETTERS, con1)
close(con1)

# R process 2
con2 <- socketConnection(Sys.info()["nodename"], port = 6011)
# as non-blocking, may need to loop for input
readLines(con2)
while(isIncomplete(con2)) {Sys.sleep(1); readLines(con2)}
close(con2)

## examples of use of encodings
cat(x, file = (con <- file("foo", "w", encoding="UTF-8"))); close(con)
# read a 'Windows Unicode' file
A <- read.table(con <- file("students", encoding="UCS-2LE")); close(con)
## End(Not run)
```

`Constants` *Built-in Constants*

Description

Constants built into R.

Usage

```
LETTERS
letters
month.abb
month.name
pi
```

Details

R has a small number of built-in constants (there is also a rather larger library of data sets which can be loaded with the function `data`).

The following constants are available:

- `LETTERS`: the 26 upper-case letters of the Roman alphabet;
- `letters`: the 26 lower-case letters of the Roman alphabet;
- `month.abb`: the three-letter abbreviations for the English month names;
- `month.name`: the English names for the months of the year;
- `pi`: the ratio of the circumference of a circle to its diameter.

These are implemented as variables in the base name space taking appropriate values.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`data`, `DateTimeClasses`.

`Quotes` for the parsing of character constants, `NumericConstants` for numeric constants.

Examples

```
## John Machin (ca 1706) computed pi to over 100 decimal places
## using the Taylor series expansion of the second term of
pi - 4*(4*atan(1/5) - atan(1/239))

## months in English
month.name
## months in your current locale
format(ISOdate(2000, 1:12, 1), "%B")
format(ISOdate(2000, 1:12, 1), "%b")
```

`contributors` *R Project Contributors*

Description

The R Who-is-who, describing who made significant contributions to the development of R.

Usage

```
contributors()
```

`Control` *Control Flow*

Description

These are the basic control-flow constructs of the R language. They function in much the same way as control statements in any Algol-like language. They are all reserved words.

Usage

```
if(cond) expr
if(cond) cons.expr  else  alt.expr

for(var in seq) expr
while(cond) expr
repeat expr
break
next
```

Arguments

`cond`	A length-one logical vector that is not `NA`. Conditions of length greater than one are accepted with a warning, but only the first element is used. Other types are coerced to logical if possible, ignoring any class.
`var`	A syntactical name for a variable.
`seq`	An expression evaluating to a vector (including a list and an expression) or to a pairlist or `NULL`.
`expr, cons.expr, alt.expr`	An *expression* in a formal sense. This is either a simple expression or a so called *compound expression*, usually of the form `{ expr1 ; expr2 }`.

Details

`break` breaks out of a `for`, `while` or `repeat` loop; control is transferred to the first statement outside the inner-most loop. `next` halts the processing of the current iteration and advances the looping index. Both `break` and `next` apply only to the innermost of nested loops.

Note that it is a common mistake to forget to put braces (`{ .. }`) around your statements, e.g., after `if(..)` or `for(....)`. In particular, you should not have a newline between `}` and `else` to avoid a syntax error in entering a `if ... else` construct at the keyboard or via `source`. For that reason, one (somewhat extreme) attitude of defensive programming is to always use braces, e.g., for `if` clauses.

The index `seq` in a `for` loop is evaluated at the start of the loop; changing it subsequently does not affect the loop. The variable `var` has the same type as `seq`, and is read-only: assigning to it does not alter `seq`. If `seq` is a factor (which is not strictly allowed) then its internal codes are used: the effect is that of `as.integer` not `as.vector`.

Value

`if` returns the value of the expression evaluated, or `NULL` if none was (which may happen if there is no `else`).

`for`, `while` and `repeat` return the value of the last expression evaluated (or `NULL` if none was), invisibly. `for` sets `var` to the last used element of `seq`, or to `NULL` if it was of length zero.

`break` and `next` have value `NULL`, although it would be strange to look for a return value.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`Syntax` for the basic R syntax and operators, `Paren` for parentheses and braces; further, `ifelse`, `switch`.

Examples

```
for(i in 1:5) print(1:i)
for(n in c(2,5,10,20,50)) {
   x <- stats::rnorm(n)
   cat(n,":", sum(x^2),"\n")
}
```

---

copyright *Copyrights of Files Used to Build R*

Description

R is released under the 'GNU Public License': see `license` for details. The license describes your right to use R. Copyright is concerned with ownership of intellectual rights, and some of the software used has conditions that the copyright must be explicitly stated: see the 'Details' section. We are grateful to these people and other contributors (see `contributors`) for the ability to use their work.

Details

The file '*R_HOME*/COPYRIGHTS' lists the copyrights in full detail.

`crossprod` *Matrix Crossproduct*

Description

Given matrices `x` and `y` as arguments, return a matrix cross-product. This is formally equivalent to (but usually slightly faster than) the call `t(x) %*% y` (`crossprod`) or `x %*% t(y)` (`tcrossprod`).

Usage

```
crossprod(x, y = NULL)

tcrossprod(x, y = NULL)
```

Arguments

`x, y`	numeric or complex matrices: `y = NULL` is taken to be the same matrix as `x`. Vectors are promoted to single-column matrices.

Value

A double or complex matrix, with appropriate `dimnames` taken from `x` and `y`.

Note

When `x` or `y` are not matrices, they are treated as column or row matrices, but their `names` are usually **not** promoted to `dimnames`. Hence, currently, the last example has empty dimnames.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`%*%` and outer product `%o%`.

Examples

```
(z <- crossprod(1:4))     # = sum(1 + 2^2 + 3^2 + 4^2)
drop(z)                   # scalar
x <- 1:4; names(x) <- letters[1:4]; x
tcrossprod(as.matrix(x)) # is
identical(tcrossprod(as.matrix(x)),
          crossprod(t(x)))
tcrossprod(x)             # no dimnames
```

Cstack_info *Report Information on C Stack Size and Usage*

Description

Report information on the C stack size and usage (if available).

Usage

```
Cstack_info()
```

Details

On most platforms, C stack information is recorded when R is initialized and used for stack-checking. If this information is unavailable, the `size` will be returned as `NA`, and stack-checking is not performed.

The information on the stack base address is thought to be accurate on Windows, Linux and FreeBSD (including MacOS X), but a heuristic is used on other platforms. Because this might be slightly inaccurate, the current usage could be estimated as negative. (The heuristic is not used on embedded uses of R on platforms where the stack base is not thought to be accurate.)

Value

An integer vector. This has named elements

`size`	The size of the stack (in bytes), or `NA` if unknown.
`current`	The estimated current usage (in bytes), possibly `NA`.
`direction`	`1` (stack grows down, the usual case) or `-1` (stack grows up).
`eval_depth`	The current evaluation depth (including two calls for the call to `Cstack_info`).

Examples

```
Cstack_info()
```

cumsum *Cumulative Sums, Products, and Extremes*

Description

Returns a vector whose elements are the cumulative sums, products, minima or maxima of the elements of the argument.

Usage

```
cumsum(x)
cumprod(x)
cummax(x)
cummin(x)
```

Arguments

x	a numeric or complex (not cummin or cummax) object, or an object that can be coerced to one of these.

Details

These are generic functions: methods can be defined for them individually or via the Math group generic.

Value

A vector of the same length and type as x (after coercion), except that cumprod returns a numeric vector for integer input (for consistency with *). Names are preserved.

An NA value in x causes the corresponding and following elements of the return value to be NA, as does integer overflow in cumsum (with a warning).

S4 methods

cumsum and cumprod are S4 generic functions: methods can be defined for them individually or via the Math group generic. cummax and cummin are individually S4 generic functions.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (cumsum only.)

Examples

```
cumsum(1:10)
cumprod(1:10)
cummin(c(3:1, 2:0, 4:2))
cummax(c(3:1, 2:0, 4:2))
```

---

cut *Convert Numeric to Factor*

Description

cut divides the range of x into intervals and codes the values in x according to which interval they fall. The leftmost interval corresponds to level one, the next leftmost to level two and so on.

Usage

```
cut(x, ...)

## Default S3 method:
cut(x, breaks, labels = NULL,
    include.lowest = FALSE, right = TRUE, dig.lab = 3,
    ordered_result = FALSE, ...)
```

Arguments

`x`	a numeric vector which is to be converted to a factor by cutting.
`breaks`	either a numeric vector of two or more cut points or a single number (greater than or equal to 2) giving the number of intervals into which `x` is to be cut.
`labels`	labels for the levels of the resulting category. By default, labels are constructed using `"(a,b]"` interval notation. If `labels = FALSE`, simple integer codes are returned instead of a factor.
`include.lowest`	logical, indicating if an 'x[i]' equal to the lowest (or highest, for `right = FALSE`) 'breaks' value should be included.
`right`	logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.
`dig.lab`	integer which is used when labels are not given. It determines the number of digits used in formatting the break numbers.
`ordered_result`	logical: should the result be an ordered factor?
`...`	further arguments passed to or from other methods.

Details

When `breaks` is specified as a single number, the range of the data is divided into `breaks` pieces of equal length, and then the outer limits are moved away by 0.1% of the range to ensure that the extreme values both fall within the break intervals. (If `x` is a constant vector, equal-length intervals are created that cover the single value.)

If a `labels` parameter is specified, its values are used to name the factor levels. If none is specified, the factor level labels are constructed as `"(b1, b2]"`, `"(b2, b3]"` etc. for `right = TRUE` and as `"[b1, b2)"`, ...if `right = FALSE`. In this case, `dig.lab` indicates the minimum number of digits should be used in formatting the numbers `b1`, `b2`, ... A larger value (up to 12) will be used if needed to distinguish between any pair of endpoints: if this fails labels such as `"Range3"` will be used.

Value

A `factor` is returned, unless `labels = FALSE` which results in the mere integer level codes.

Note

Instead of `table(cut(x, br))`, `hist(x, br, plot = FALSE)` is more efficient and less memory hungry. Instead of `cut(*, labels = FALSE)`, `findInterval()` is more efficient.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`split` for splitting a variable according to a group factor; `factor`, `tabulate`, `table`, `findInterval()`.

`quantile` for ways of choosing breaks of roughly equal content (rather than length), `cut2` in package **Hmisc** for a canned way to form quantile groups.

Examples

```
Z <- stats::rnorm(10000)
table(cut(Z, breaks = -6:6))
sum(table(cut(Z, breaks = -6:6, labels=FALSE)))
sum(graphics::hist(Z, breaks = -6:6, plot=FALSE)$counts)

cut(rep(1,5),4)#-- dummy
tx0 <- c(9, 4, 6, 5, 3, 10, 5, 3, 5)
x <- rep(0:8, tx0)
stopifnot(table(x) == tx0)

table( cut(x, b = 8))
table( cut(x, breaks = 3*(-2:5)))
table( cut(x, breaks = 3*(-2:5), right = FALSE))

##--- some values OUTSIDE the breaks :
table(cx  <- cut(x, breaks = 2*(0:4)))
table(cxl <- cut(x, breaks = 2*(0:4), right = FALSE))
which(is.na(cx));  x[is.na(cx)]  #-- the first 9  values  0
which(is.na(cxl)); x[is.na(cxl)] #-- the last  5  values  8

## Label construction:
y <- stats::rnorm(100)
table(cut(y, breaks = pi/3*(-3:3)))
table(cut(y, breaks = pi/3*(-3:3), dig.lab=4))

table(cut(y, breaks =  1*(-3:3), dig.lab=4))
# extra digits don't "harm" here
table(cut(y, breaks =  1*(-3:3), right = FALSE))
#- the same, since no exact INT!

## sometimes the default dig.lab is not enough to be avoid confusion:
aaa <- c(1,2,3,4,5,2,3,4,5,6,7)
cut(aaa, 3)
cut(aaa, 3, dig.lab=4, ordered = TRUE)

## one way to extract the breakpoints
labs <- levels(cut(aaa, 3))
cbind(lower = as.numeric( sub("\\((.+),.*", "\\1", labs) ),
      upper = as.numeric( sub("[^,]*,([^]]*)\\]", "\\1", labs) ))
```

`cut.POSIXt` *Convert a Date or Date-Time Object to a Factor*

Description

Method for `cut` applied to date-time objects.

Usage

```
## S3 method for class 'POSIXt':
cut(x, breaks, labels = NULL, start.on.monday = TRUE,
    right = FALSE, ...)
```

```
## S3 method for class 'Date':
cut(x, breaks, labels = NULL, start.on.monday = TRUE,
    right = FALSE, ...)
```

Arguments

x	an object inheriting from class "POSIXt" or "Date".
breaks	a vector of cut points *or* number giving the number of intervals which x is to be cut into *or* an interval specification, one of "sec", "min", "hour", "day", "DSTday", "week", "month", "quarter" or "year", optionally preceded by an integer and a space, or followed by "s". For "Date" objects only "day", "week", "month", "quarter" and "year" are allowed.
labels	labels for the levels of the resulting category. By default, labels are constructed from the left-hand end of the intervals (which are include for the default value of right). If labels = FALSE, simple integer codes are returned instead of a factor.
start.on.monday	logical. If breaks = "weeks", should the week start on Mondays or Sundays?
right, ...	arguments to be passed to or from other methods.

Details

Using both right = TRUE and include.lowest = TRUE will include both ends of the range of dates.

Using breaks = "quarter" will create intervals of 3 calendar months, with the intervals beginning on January 1, April 1, July 1 or October 1, based upon min(x) as appropriate.

Value

A factor is returned, unless labels = FALSE which returns the integer level codes.

See Also

seq.POSIXt, seq.Date, cut

Examples

```
## random dates in a 10-week period
cut(ISOdate(2001, 1, 1) + 70*86400*stats::runif(100), "weeks")
cut(as.Date("2001/1/1") + 70*stats::runif(100), "weeks")
```

---

data.class *Object Classes*

Description

Determine the class of an arbitrary R object.

Usage

```
data.class(x)
```

Arguments

x	an R object.

Value

character string giving the *class* of x.

The class is the (first element) of the class attribute if this is non-NULL, or inferred from the object's dim attribute if this is non-NULL, or mode(x).

Simply speaking, data.class(x) returns what is typically useful for method dispatching. (Or, what the basic creator functions already and maybe eventually all will attach as a class attribute.)

Note

For compatibility reasons, there is one exception to the rule above: When x is integer, the result of data.class(x) is "numeric" even when x is classed.

See Also

class

Examples

```
x <- LETTERS
data.class(factor(x))                 # has a class attribute
data.class(matrix(x, ncol = 13))      # has a dim attribute
data.class(list(x))                   # the same as mode(x)
data.class(x)                         # the same as mode(x)

stopifnot(data.class(1:2) == "numeric")# compatibility "rule"
```

---

data.frame *Data Frames*

Description

This function creates data frames, tightly coupled collections of variables which share many of the properties of matrices and of lists, used as the fundamental data structure by most of R's modeling software.

Usage

```
data.frame(..., row.names = NULL, check.rows = FALSE,
           check.names = TRUE,
           stringsAsFactors = default.stringsAsFactors())

default.stringsAsFactors()
```

Arguments

`...`	these arguments are of either the form `value` or `tag = value`. Component names are created based on the tag (if present) or the deparsed argument itself.
`row.names`	`NULL` or a single integer or character string specifying a column to be used as row names, or a character or integer vector giving the row names for the data frame.
`check.rows`	if `TRUE` then the rows are checked for consistency of length and names.
`check.names`	logical. If `TRUE` then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names and are not duplicated. If necessary they are adjusted (by `make.names`) so that they are.
`stringsAsFactors`	logical: should character vectors be converted to factors? The 'factory-fresh' default is `TRUE`, but this can be changed by setting `options(stringsAsFactors = FALSE)`.

Details

A data frame is a list of variables of the same length with unique row names, given class `"data.frame"`. If there are zero variables, the row names determine the number of rows.

Duplicate column names are allowed, but you need to use `check.names=FALSE` for `data.frame` to generate such a data frame. However, not all operations on data frames will preserve duplicated column names: for example matrix-like subsetting will force column names in the result to be unique.

`data.frame` converts each of its arguments to a data frame by calling `as.data.frame(optional=TRUE)`. As that is a generic function, methods can be written to change the behaviour of arguments according to their classes: R comes with many such methods. Character variables passed to `data.frame` are converted to factor columns unless protected by `I`. If a list or data frame or matrix is passed to `data.frame` it is as if each component or column had been passed as a separate argument (except for matrices of class `"model.matrix"` and those protected by `I`).

Objects passed to `data.frame` should have the same number of rows, but atomic vectors, factors and character vectors protected by `I` will be recycled a whole number of times if necessary.

If row names are not supplied in the call to `data.frame`, the row names are taken from the first component that has suitable names, for example a named vector or a matrix with rownames or a data frame. (If that component is subsequently recycled, the names are discarded with a warning.) If `row.names` was supplied as `NULL` or no suitable component was found the row names are the integer sequence starting at one (and such row names are considered to be 'automatic', and not preserved by `as.matrix`).

If row names are supplied of length one and the data frame has a single row, the `row.names` is taken to specify the row names and not a column (by name or number).

Names are removed from vector inputs not protected by `I`.

`default.stringsAsFactors` is a utility that takes `getOption("stringsAsFactors")` and ensures the result is `TRUE` or `FALSE` (or throws an error if the value is not `NULL`).

Value

A data frame, a matrix-like structure whose columns may be of differing types (numeric, logical, factor and character and so on).

How the names of the data frame are created is complex, and the rest of this paragraph is only the basic story. If the arguments are all named and simple objects (not lists, matrices of data frames) then the argument names give the column names. For an unnamed simple argument, a deparsed version of the argument is used as the name (with an enclosing `I(...)` removed). For a named matrix/list/data frame argument with ore than one named column, the names of the columns are the name of the argument followed by a dot and the column name inside the argument: if the argument is unnamed, the argument's column names are used. For a named or unnamed matrix/list/data frame argument that contains a single column, the column name in the result is the column name in the argument. Finally, the names are adjusted to be unique and syntactically valid unless `check.names = FALSE`.

Note

In versions of R prior to 2.4.0 `row.names` had to be character: to ensure compatibility with earlier versions of R, supply a character vector as the `row.names` argument.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`I`, `plot.data.frame`, `print.data.frame`, `row.names`, `names` (for the column names), `[.data.frame` for subsetting methods, `Math.data.frame` etc, about *Group* methods for `data.frames`; `read.table`, `make.names`.

Examples

```
L3 <- LETTERS[1:3]
(d <- data.frame(cbind(x=1, y=1:10), fac=sample(L3, 10, replace=TRUE)))

## The same with automatic column names:
data.frame(cbind(  1,   1:10),     sample(L3, 10, replace=TRUE))

is.data.frame(d)

## do not convert to factor, using I() :
(dd <- cbind(d, char = I(letters[1:10])))
rbind(class=sapply(dd, class), mode=sapply(dd, mode))

stopifnot(1:10 == row.names(d))# {coercion}

(d0  <- d[, FALSE]) # NULL data frame with 10 rows
(d.0 <- d[FALSE, ]) # <0 rows> data frame  (3 cols)
(d00 <- d0[FALSE,])  # NULL data frame with 0 rows
```

`data.matrix` *Convert a Data Frame to a Numeric Matrix*

Description

Return the matrix obtained by converting all the variables in a data frame to numeric mode and then binding them together as the columns of a matrix. Factors and ordered factors are replaced by their internal codes.

Usage

```
data.matrix(frame, rownames.force = NA)
```

Arguments

`frame`	a data frame whose components are logical vectors, factors or numeric vectors.
`rownames.force`	logical indicating if the resulting matrix should have character (rather than `NULL`) `rownames`. The default, `NA`, uses `NULL` rownames if the data frame has 'automatic' row.names or for a zero-row data frame.

Details

Logical and factor columns are converted to integers. Any other column which is not numeric (according to `is.numeric`) is converted by `as.numeric` or, for S4 objects, `as(, "numeric")`. If all columns are integer (after conversion) the result is an integer natrix, otherwise a numeric (double) matrix.

Value

If `frame` inherits from class `"data.frame"`, an integer or numeric matrix of the same dimensions as `frame`, with dimnames taken from the `row.names` (or `NULL`, depending on `rownames.force`) and `names`.

Otherwise, the result of `as.matrix`.

Note

The default behaviour for data frames differs from R < 2.5.0 which always gave the result character rownames.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`as.matrix`, `data.frame`, `matrix`.

Examples

```
DF <- data.frame(a=1:3, b=letters[10:12],
                 c=seq(as.Date("2004-01-01"), by = "week", len = 3),
                 stringsAsFactors = TRUE)
data.matrix(DF[1:2])
data.matrix(DF)
```

date *System Date and Time*

Description

Returns a character string of the current system date and time.

Usage

```
date()
```

Value

The string has the form `"Fri Aug 20 11:11:00 1999"`, i.e., length 24, since it relies on POSIX's `ctime` ensuring the above fixed format. Timezone and Daylight Saving Time are taken account of, but *not* indicated in the result.

The day and month abbreviations are always in English, irrespective of locale.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`Sys.Date` and `Sys.time`; `Date` and `DateTimeClasses` for objects representing date and time.

Examples

```
(d <- date())
nchar(d) == 24

## something similar in the current locale
format(Sys.time(), "%a %b %d %H:%M:%S %Y")
```

Dates *Date Class*

Description

Description of the class `"Date"` representing calendar dates.

Usage

```
## S3 method for class 'Date':
summary(object, digits = 12, ...)
```

Arguments

object	An object summarized.
digits	Number of significant digits for the computations.
...	Further arguments to be passed from or to other methods.

Details

Dates are represented as the number of days since 1970-01-01, with negative values for earlier dates. They are always printed following the rules of the current Gregorian calendar, even though that calendar was not in use long ago (it was adopted in 1752 in Great Britain and its colonies).

It is intended that the date should be an integer, but this is not enforced in the internal representation. Fractional days will be ignored when printing. It is possible to produce fractional days via the mean method or by adding or subtracting (see Ops.Date).

See Also

Sys.Date for the current date.

Ops.Date for operators on "Date" objects.

format.Date for conversion to and from character strings.

plot.Date and hist.Date for plotting.

weekdays for convenience extraction functions.

seq.Date, cut.Date, round.Date for utility operations.

DateTimeClasses for date-time classes.

Examples

```
(today <- Sys.Date())
format(today, "%d %b %Y")  # with month as a word
(tenweeks <- seq(today, length.out=10, by="1 week")) # next ten weeks
weekdays(today)
months(tenweeks)
as.Date(.leap.seconds)
```

DateTimeClasses *Date-Time Classes*

Description

Description of the classes "POSIXlt" and "POSIXct" representing calendar dates and times (to the nearest second).

Usage

```
## S3 method for class 'POSIXct':
print(x, ...)

## S3 method for class 'POSIXct':
summary(object, digits = 15, ...)
```

```
time + z
time - z
time1 lop time2
```

Arguments

x, object	An object to be printed or summarized from one of the date-time classes.
digits	Number of significant digits for the computations: should be high enough to represent the least important time unit exactly.
...	Further arguments to be passed from or to other methods.
time	date-time objects
time1, time2	date-time objects or character vectors. (Character vectors are converted by as.POSIXct.)
z	a numeric vector (in seconds)
lop	One of ==, !=, <, <=, > or >=.

Details

There are two basic classes of date/times. Class "POSIXct" represents the (signed) number of seconds since the beginning of 1970 as a numeric vector. Class "POSIXlt" is a named list of vectors representing

sec 0–61: seconds

min 0–59: minutes

hour 0–23: hours

mday 1–31: day of the month

mon 0–11: months after the first of the year.

year Years since 1900.

wday 0–6 day of the week, starting on Sunday.

yday 0–365: day of the year.

isdst Daylight savings time flag. Positive if in force, zero if not, negative if unknown.

The classes correspond to the POSIX/C99 constructs of 'calendar time' (the time_t data type) and 'local time' (or broken-down time, the struct tm data type), from which they also inherit their names.

"POSIXct" is more convenient for including in data frames, and "POSIXlt" is closer to human-readable forms. A virtual class "POSIXt" inherits from both of the classes: it is used to allow operations such as subtraction to mix the two classes.

Components wday and yday of "POSIXlt" are for information, and are not used in the conversion to calendar time. However, isdst is needed to distinguish times at the end of DST: typically 1am to 2am occurs twice, first in DST and then in standard time. At all other times isdst can be deduced from the first six values, but the behaviour if it is set incorrectly is platform-dependent.

Logical comparisons and limited arithmetic are available for both classes. One can add or subtract a number of seconds from a date-time object, but not add two date-time objects. Subtraction of two date-time objects is equivalent to using difftime. Be aware that "POSIXlt" objects will be interpreted as being in the current timezone for these operations, unless a timezone has been specified.

"POSIXlt" objects will often have an attribute "tzone", a character vector of length 3 giving the timezone name from the TZ environment variable and the names of the base timezone and the alternate (daylight-saving) timezone. Sometimes this may just be of length one, giving the timezone name.

"POSIXct" objects may also have an attribute "tzone", a character vector of length one. If set, it will determine how the object is converted to class "POSIXlt" and in particular how it is printed. This is usually desirable, but if you want to specify an object in a particular timezone but to be printed in the current timezone you may want to remove the "tzone" attribute (e.g. by c(x)).

Unfortunately, the conversion is complicated by the operation of time zones and leap seconds (23 days have been 86401 seconds long so far: the times of the extra seconds are in the object .leap.seconds). The details of this are entrusted to the OS services where possible. This always covers the period 1970–2037, and on most machines back to 1902 (when time zones were in their infancy). Outside the platform limits we use our own C code. This uses the offset from GMT in use either for 1902 (when there was no DST) or that predicted for one of 2030 to 2037 (chosen so that the likely DST transition days are Sundays), and uses the alternate (daylight-saving) timezone only if isdst is positive or (if −1) if DST was predicted to be in operation in the 2030s on that day. (There is no reason to suppose that the DST rules will remain the same in the future, and indeed the US legislated in 2005 to change its rules as from 2007, with a possible future reversion.)

It seems that some rare systems use leap seconds, but most ignore them (as required by POSIX). This is detected and corrected for at build time, so all "POSIXct" times used by R do not include leap seconds. (Conceivably this could be wrong if the system has changed since build time, just possibly by changing locales or the 'zoneinfo' database.)

Using c on "POSIXlt" objects converts them to the current time zone.

A few times have specific issues. First, the leapseconds are ignored, and real times such as "2005-12-31 23:59:60" are (probably) treated as the next second. However, they will never be generated by R, and are unlikely to arise as input. Second, on some OSes there is a problem in the POSIX/C99 standard with "1969-12-31 23:59:59", which is −1 in calendar time and is used as an error code. Thus as.POSIXct("1969-12-31 23:59:59", format="%Y-%m-%d %H:%M:%S", tz="UTC") may give NA, and hence as.POSIXct("1969-12-31 23:59:59", tz="UTC") will give "1969-12-31 23:59:50". Other OSes report errors separately and so are able to handle that time as valid.

Sub-second Accuracy

Classes "POSIXct" and "POSIXlt" are able to express fractions of a second. (Conversion of fractions between the two forms may not be exact, but will have better than microsecond accuracy.)

Fractional seconds are printed only if options("digits.secs") is set: see strftime.

Warning

Some Unix-like systems (especially Linux ones) do not have "TZ" set, yet have internal code that expects it (as does POSIX). We have tried to work around this, but if you get unexpected results try setting "TZ". See as.POSIXlt for valid settings.

See Also

Dates for dates without times.

as.POSIXct and as.POSIXlt for conversion between the classes.

strptime for conversion to and from character representations.

Sys.time for clock time as a "POSIXct" object.

difftime for time intervals.

cut.POSIXt, seq.POSIXt, round.POSIXt and trunc.POSIXt for methods for these classes.

weekdays.POSIXt for convenience extraction functions.

Examples

```
(z <- Sys.time())             # the current date, as class "POSIXct"

Sys.time() - 3600             # an hour ago

as.POSIXlt(Sys.time(), "GMT") # the current time in GMT
format(.leap.seconds)         # all 23 leapseconds in your timezone
print(.leap.seconds, tz="PST8PDT")  # and in Seattle's
```

---

dcf *Read and Write Data in DCF Format*

Description

Reads or writes an R object from/to a file in Debian Control File format.

Usage

```
read.dcf(file, fields = NULL, all = FALSE)

write.dcf(x, file = "", append = FALSE,
          indent = 0.1 * getOption("width"),
          width = 0.9 * getOption("width"))
```

Arguments

file	either a character string naming a file or a connection. "" indicates output to the console. For read.dcf this can name a gzip-compressed file.
fields	Fields to read from the DCF file. Default is to read all fields.
all	a logical indicating whether in case of multiple occurrences of a field in a record, all these should be gathered. If all is false (default), only the last such occurrence is used.
x	the object to be written, typically a data frame. If not, it is attempted to coerce x to a data frame.
append	logical. If TRUE, the output is appended to the file. If FALSE, any existing file of the name is destroyed.
indent	a positive integer specifying the indentation for continuation lines in output entries.
width	a positive integer giving the target column for wrapping lines in the output.

Details

DCF is a simple format for storing databases in plain text files that can easily be directly read and written by humans. DCF is used in various places to store R system information, like descriptions and contents of packages.

The DCF rules as implemented in R are:

1. A database consists of one or more records, each with one or more named fields. Not every record must contain each field. Fields may appear more than once in a record.
2. Regular lines start with a non-whitespace character.
3. Regular lines are of form `tag:value`, i.e., have a name tag and a value for the field, separated by `:` (only the first `:` counts). The value can be empty (=whitespace only).
4. Lines starting with whitespace are continuation lines (to the preceding field) if at least one character in the line is non-whitespace. Continuation lines where the only non-whitespace character is a '`.`' are taken as blank lines (allowing for multi-paragraph field values).
5. Records are separated by one or more empty (=whitespace only) lines.

By default, `read.dcf` returns a character matrix with one row per record and one column per field. Leading and trailing whitespace of field values is ignored. If a tag name is specified, but the corresponding value is empty, then an empty string is returned. If the tag name of a field is never used in a record, then `NA` is returned. If fields are repeated within a record, the last one encountered is returned. Malformed lines lead to an error. If `all` is true, a data frame is returned, again with one row per record and one column per field, and columns lists of character vectors for fields with multiple occurrences, and character vectors otherwise.

`write.dcf` does not write `NA` fields.

See Also

`write.table`.

Examples

```
## Create a reduced version of the 'CONTENTS' file in package 'splines'
x <- read.dcf(file = system.file("CONTENTS", package = "splines"),
              fields = c("Entry", "Description"))
write.dcf(x)
```

---

debug *Debug a Function*

Description

Set or unset the debugging flag on a function.

Usage

```
debug(fun)
undebug(fun)
```

Arguments

`fun`	any interpreted R function.

Details

When a function flagged for debugging is entered, normal execution is suspended and the body of function is executed one statement at a time. A new browser context is initiated for each step (and the previous one destroyed).

At the debug prompt the user can enter commands or R expressions. The commands are

n (or just return). Advance to the next step.

c continue to the end of the current context: e.g. to the end of the loop if within a loop or to the end of the function.

cont synonym for `c`.

where print a stack trace of all active function calls.

Q exit the browser and the current evaluation and return to the top-level prompt.

(Leading and trailing whitespace is ignored, except for return).

Anything else entered at the debug prompt is interpreted as an R expression to be evaluated in the calling environment: in particular typing an object name will cause the object to be printed, and `ls()` lists the objects in the calling frame. (If you want to look at an object with a name such as `n`, print it explicitly.)

If a function is defined inside a function, single-step though to the end of its definition, and then call `debug` on its name.

In order to debug S4 methods (see `Methods`), you need to use `trace`, typically calling `browser`, e.g., as

```
trace("plot", browser, exit=browser, signature = c("track",
"missing"))
```

See Also

`browser`, `trace`; `traceback` to see the stack after an `Error: ... ` message; `recover` for another debugging approach.

`Defunct` *Marking Objects as Defunct*

Description

When a function is removed from R it should be replaced by a function which calls `.Defunct`.

Usage

```
.Defunct(new, package = NULL, msg)
```

Arguments

`new`	character string: A suggestion for a replacement function.
`package`	character string: The package to be used when suggesting where the defunct function might be listed.
`msg`	character string: A message to be printed, if missing a default message is used.

Details

`.Defunct` is called from defunct functions. Functions should be listed in `help("pkg-defunct")` for an appropriate `pkg`, including `base`.

See Also

`Deprecated`.

`base-defunct` and so on which list the defunct functions in the packages.

`delayedAssign` *Delay Evaluation*

Description

`delayedAssign` creates a *promise* to evaluate the given expression if its value is requested. This provides direct access to the *lazy evaluation* mechanism used by R for the evaluation of (interpreted) functions.

Usage

```
delayedAssign(x, value, eval.env = parent.frame(1),
              assign.env = parent.frame(1))
```

Arguments

`x`	a variable name (given as a quoted string in the function call)
`value`	an expression to be assigned to `x`
`eval.env`	an environment in which to evaluate `value`
`assign.env`	an environment in which to assign `x`

Details

Both `eval.env` and `assign.env` default to the currently active environment.

The expression assigned to a promise by `delayedAssign` will not be evaluated until it is eventually 'forced'. This happens when the variable is first accessed.

When the promise is eventually forced, it is evaluated within the environment specified by `eval.env` (whose contents may have changed in the meantime). After that, the value is fixed and the expression will not be evaluated again.

Value

This function is invoked for its side effect, which is assigning a promise to evaluate `value` to the variable `x`.

See Also

`substitute`, to see the expression associated with a promise.

Examples

```
msg <- "old"
delayedAssign("x", msg)
msg <- "new!"
x #- new!
substitute(x) #- msg

delayedAssign("x", {
    for(i in 1:3)
        cat("yippee!\n")
    10
})

x^2 #- yippee
x^2 #- simple number

e <- (function(x, y = 1, z) environment())(1+2, "y", {cat(" HO! "); pi+2})
(le <- as.list(e)) # evaluates the promises
```

---

deparse *Expression Deparsing*

Description

Turn unevaluated expressions into character strings.

Usage

```
deparse(expr, width.cutoff = 60,
        backtick = mode(expr) %in% c("call", "expression", "(", "function"),
        control = c("keepInteger", "showAttributes", "keepNA"),
        nlines = -1)
```

Arguments

`expr`	any R expression.
`width.cutoff`	integer in $[20, 500]$ determining the cutoff at which line-breaking is tried.
`backtick`	logical indicating whether symbolic names should be enclosed in backticks if they do not follow the standard syntax.
`control`	character vector of deparsing options. See `.deparseOpts`.
`nlines`	the maximum number of lines to produce. Negative values indicate no limit.

Details

This function turns unevaluated expressions (where 'expression' is taken in a wider sense than the strict concept of a vector of mode `"expression"` used in `expression`) into character strings (a kind of inverse to `parse`).

A typical use of this is to create informative labels for data sets and plots. The example shows a simple use of this facility. It uses the functions `deparse` and `substitute` to create labels for a plot which are character string versions of the actual arguments to the function `myplot`.

The default for the backtick option is not to quote single symbols but only composite expressions. This is a compromise to avoid breaking existing code.

Using control = "all" comes closest to making deparse() an inverse of parse(). However, not all objects are deparse-able even with this option and a warning will be issued if the function recognizes that it is being asked to do the impossible.

Numeric and complex vectors are converted using 15 significant digits: see as.character for more details.

width.cutoff is a lower bound for the line lengths: deparsing a line proceeds until at least width.cutoff *bytes* have been output and e.g. arg = value expressions will not be split across lines.

Note

To avoid the risk of a source attribute out of sync with the actual function definition, the source attribute of a function will never be deparsed as an attribute.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

substitute, parse, expression.

Quotes for quoting conventions, including backticks.

Examples

```
require(stats); require(graphics)

deparse(args(lm))
deparse(args(lm), width = 500)
myplot <-
function(x, y) {
    plot(x, y, xlab=deparse(substitute(x)),
        ylab=deparse(substitute(y)))
}
e <- quote(`foo bar`)
deparse(e)
deparse(e, backtick=TRUE)
e <- quote(`foo bar`+1)
deparse(e)
deparse(e, control = "all")
```

---

deparseOpts *Options for Expression Deparsing*

Description

Process the deparsing options for deparse, dput and dump.

Usage

```
.deparseOpts(control)
```

Arguments

control	character vector of deparsing options.

Details

This is called by deparse, dput and dump to process their control argument.

The control argument is a vector containing zero or more of the following strings. Partial string matching is used.

keepInteger Either surround integer vectors by as.integer() or use suffix L, so they are not converted to type double when parsed. This includes making sure that integer NAs are preserved (via NA_integer_ if there are no non-NA values in the vector, unless "S_compatible" is set).

quoteExpressions Surround expressions with quote(), so they are not evaluated when re-parsed.

showAttributes If the object has attributes (other than a source attribute), use structure() to display them as well as the object value. This is the default for deparse and dput.

useSource If the object has a source attribute, display that instead of deparsing the object. Currently only applies to function definitions.

warnIncomplete Some exotic objects such as environments, external pointers, etc. can not be deparsed properly. This option causes a warning to be issued if any of those may give problems.

Also, the parser in R < 2.7.0 would only accept strings of up to 8192 bytes, and this option gives a warning for longer strings.

keepNA Integer, real and character NAs are surrounded by coercion where necessary to ensure that they are parsed to the same type.

all An abbreviated way to specify all of the options listed above. This is the default for dump, and the options used by edit (which are fixed).

delayPromises Deparse promises in the form <promise: expression> rather than evaluating them. The value and the environment of the promise will not be shown and the deparsed code cannot be sourced.

S_compatible Make deparsing as far as possible compatible with S and R < 2.5.0. For compatibility with S, integer values of double vectors are deparsed with a trailing decimal point.

For the most readable (but perhaps incomplete) display, use control = NULL. This displays the object's value, but not its attributes. The default in deparse is to display the attributes as well, but not to use any of the other options to make the result parseable. (dput and dump do use more default options, and printing of functions without sources uses c("keepInteger", "keepNA").)

Using control = "all" comes closest to making deparse() an inverse of parse(). However, not all objects are deparse-able even with this option. A warning will be issued if the function recognizes that it is being asked to do the impossible.

Value

A numerical value corresponding to the options selected.

Deprecated — *Marking Objects as Deprecated*

Description

When an object is about removed from R it is first deprecated and should include a call to `.Deprecated`.

Usage

```
.Deprecated(new, package=NULL, msg)
```

Arguments

`new`	character string: A suggestion for a replacement function.
`package`	character string: The package to be used when suggesting where the deprecated function might be listed.
`msg`	character string: A message to be printed, if missing a default message is used.

Details

`.Deprecated("<new name>")` is called from deprecated functions. The original help page for these functions is often available at `help("oldName-deprecated")` (note the quotes). Functions should be listed in `help("pkg-deprecated")` for an appropriate `pkg`, including `base`.

See Also

`Defunct`

`base-deprecated` and so on which list the deprecated functions in the packages.

det — *Calculate the Determinant of a Matrix*

Description

`det` calculates the determinant of a matrix. `determinant` is a generic function that returns separately the modulus of the determinant, optionally on the logarithm scale, and the sign of the determinant.

Usage

```
det(x, ...)
determinant(x, logarithm = TRUE, ...)
```

Arguments

`x`	numeric matrix.
`logarithm`	logical; if `TRUE` (default) return the logarithm of the modulus of the determinant.
`...`	Optional arguments. At present none are used. Previous versions of `det` allowed an optional `method` argument. This argument will be ignored but will not produce an error.

Details

The `determinant` function uses an LU decomposition and the `det` function is simply a wrapper around a call to `determinant`.

Often, computing the determinant is *not* what you should be doing to solve a given problem.

Value

For `det`, the determinant of `x`. For `determinant`, a list with components

`modulus`	a numeric value. The modulus (absolute value) of the determinant if `logarithm` is `FALSE`; otherwise the logarithm of the modulus.
`sign`	integer; either $+1$ or -1 according to whether the determinant is positive or negative.

Examples

```
(x <- matrix(1:4, ncol=2))
unlist(determinant(x))
det(x)

det(print(cbind(1,1:3,c(2,0,1))))
```

---

detach *Detach Objects from the Search Path*

Description

Detach a database, i.e., remove it from the `search()` path of available R objects. Usually, this is either a `data.frame` which has been `attach`ed or a package which was required previously.

Usage

```
detach(name, pos = 2, version, unload = FALSE)
```

Arguments

`name`	The object to detach. Defaults to `search()[pos]`. This can be an unquoted name or a character string but *not* a character vector. If a number is supplied this is taken as `pos`.
`pos`	Index position in `search()` of database to detach. When `name` is a number, `pos = name` is used.

`version` A character string denoting a version number of the package to be removed. This should be used only with a versioned installation of the package: see `library`.

`unload` A logical value indicating whether or not to attempt to unload the namespace and S4 methods when a package is being detached. If the package has a namespace and `unload` is `TRUE`, then `detach` will attempt to unload the namespace and remove any S4 methods defined by the package. If the namespace is in use or `unload` is `FALSE`, no unloading will occur.

Details

This most commonly used with a single number argument referring to a position on the search list, and can also be used with a unquoted or quoted name of an item on the search list such as `package:tools`.

When a package have been loaded with an explicit version number it can be detached using the name shown by `search` or by supplying `name` and `version`: see the examples.

If a package has a namespace, detaching it does not by default unload the namespace (and may not even with `unload=TRUE`), and detaching will not in general unload any dynamically loaded compiled code (DLLs). Further, registered S3 methods from the namespace will not be removed. If you use `library` on a package whose name space is loaded, it attaches the exports of the loaded name space. So detaching and re-attaching a package may not refresh some or all components of the package, and is inadvisable.

Value

The attached database is returned invisibly, either as `data.frame` or as `list`.

Note

You cannot detach either the workspace (position 1) or the **base** package (the last item in the search list).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`attach`, `library`, `search`, `objects`, `unloadNamespace`, `library.dynam.unload`.

Examples

```
require(splines) # package
detach(package:splines)
## could equally well use detach("package:splines")
## but NOT  pkg <- "package:splines"; detach(pkg)
## Instead, use
library(splines)
pkg <- "package:splines"
detach(pos = match(pkg, search()))

## careful: do not do this unless 'splines' is not already loaded.
library(splines)
detach(2) # 'pos' used for 'name'
```

```
## an example of the name argument to attach
## and of detaching a database named by a character vector
attach_and_detach <- function(db, pos=2)
{
   name <- deparse(substitute(db))
   attach(db, pos=pos, name=name)
   print(search()[pos])
   eval(substitute(detach(n), list(n=name)))
}
attach_and_detach(women, pos=3)

## Not run:
## Using a versioned install
library(ash, version="1.0-9")  # or perhaps just library(ash)
# then one of
detach("package:ash", version="1.0-9")
# or
detach("package:ash_1.0-9")
## End(Not run)
```

`diag` *Matrix Diagonals*

Description

Extract or replace the diagonal of a matrix, or construct a diagonal matrix.

Usage

```
diag(x = 1, nrow, ncol)
diag(x) <- value
```

Arguments

`x`	a matrix, vector or 1D array.
`nrow, ncol`	Optional dimensions for the result.
`value`	either a single value or a vector of length equal to that of the current diagonal. Should be of a mode which can be coerced to that of `x`.

Value

If `x` is a matrix then `diag(x)` returns the diagonal of `x`. The resulting vector will have `names` if the matrix `x` has matching column and row names.

If `x` is a vector (or 1D array) of length two or more, then `diag(x)` returns a diagonal matrix whose diagonal is `x`.

If `x` is a vector of length one then `diag(x)` returns an identity matrix of order the nearest integer to `x`. The dimension of the returned matrix can be specified by `nrow` and `ncol` (the default is square).

The replacement form sets the diagonal of the matrix `x` to the given value(s).

Note

Using `diag(x)` can have unexpected effects if `x` is a vector that could be of length one. Use `diag(x, nrow = length(x))` for consistent behaviour.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`upper.tri`, `lower.tri`, `matrix`.

Examples

```
require(stats)
dim(diag(3))
diag(10,3,4) # guess what?
all(diag(1:3) == {m <- matrix(0,3,3); diag(m) <- 1:3; m})

diag(var(M <- cbind(X = 1:5, Y = stats::rnorm(5))))
#-> vector with names "X" and "Y"

rownames(M) <- c(colnames(M),rep("",3));
M; diag(M) #  named as well
```

`diff` *Lagged Differences*

Description

Returns suitably lagged and iterated differences.

Usage

```
diff(x, ...)

## Default S3 method:
diff(x, lag = 1, differences = 1, ...)

## S3 method for class 'POSIXt':
diff(x, lag = 1, differences = 1, ...)

## S3 method for class 'Date':
diff(x, lag = 1, differences = 1, ...)
```

Arguments

`x`	a numeric vector or matrix containing the values to be differenced.
`lag`	an integer indicating which lag to use.
`differences`	an integer indicating the order of the difference.
`...`	further arguments to be passed to or from methods.

Details

`diff` is a generic function with a default method and ones for classes `"ts"`, `"POSIXt"` and `"Date"`.

`NA`'s propagate.

Value

If `x` is a vector of length `n` and `differences=1`, then the computed result is equal to the successive differences `x[(1+lag):n] - x[1:(n-lag)]`.

If `difference` is larger than one this algorithm is applied recursively to `x`. Note that the returned value is a vector which is shorter than `x`.

If `x` is a matrix then the difference operations are carried out on each column separately.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`diff.ts`, `diffinv`.

Examples

```
diff(1:10, 2)
diff(1:10, 2, 2)
x <- cumsum(cumsum(1:10))
diff(x, lag = 2)
diff(x, differences = 2)

diff(.leap.seconds)
```

`difftime` *Time Intervals*

Description

Create, print and round time intervals.

Usage

```
time1 - time2

difftime(time1, time2, tz = "",
         units = c("auto", "secs", "mins", "hours",
                   "days", "weeks"))

as.difftime(tim, format = "%X", units="auto")

## S3 method for class 'difftime':
round(x, digits = 0, ...)
```

```
## S3 method for class 'difftime':
format(x, ...)
## S3 method for class 'difftime':
units(x)
## S3 replacement method for class 'difftime':
units(x) <- value
## S3 method for class 'difftime':
as.double(x, units="auto", ...)
```

Arguments

time1, time2	date-time or date objects.
tz	a timezone specification to be used for the conversion. System-specific, but "" is the current time zone, and "GMT" is UTC.
units	character. Units in which the results are desired. Can be abbreviated.
value	character. Like units above, except that abbreviations are not allowed.
tim	character string or numeric value specifying a time interval.
format	character specifying the format of tim: see strptime. The default is a locale-specific time format.
x	an object inheriting from class "difftime".
digits	integer. Number of significant digits to retain.
...	arguments to be passed to or from other methods.

Details

Function difftime calculates a difference of two date/time objects and returns an object of class "difftime" with an attribute indicating the units. There is a round method for objects of this class, as well as methods for the group-generic (see Ops) logical and arithmetic operations.

If units = "auto", a suitable set of units is chosen, the largest possible (excluding "weeks") in which all the absolute differences are greater than one.

Subtraction of date-time objects gives an object of this class, by calling difftime with units="auto". Alternatively, as.difftime() works on character-coded or numeric time intervals; in the latter case, units must be specified, and format has no effect.

Limited arithmetic is available on "difftime" objects: they can be added or subtracted, and multiplied or divided by a numeric vector. In addition, adding or subtracting a numeric vector implicitly converts the numeric vector to a "difftime" object with the same units as the "difftime" object.

The units of a "difftime" object can be extracted by the units function, which also has an replacement form. If the units are changed, the numerical value is scaled accordingly.

The as.double method returns the numeric value expressed in the specified units. Using units="auto" means the units of the object.

The format method simply formats the numeric value and appends the units as a text string.

See Also

DateTimeClasses.

Examples

```
(z <- Sys.time() - 3600)
Sys.time() - z                # just over 3600 seconds.

## time interval between releases of R 1.2.2 and 1.2.3.
ISOdate(2001, 4, 26) - ISOdate(2001, 2, 26)

as.difftime(c("0:3:20", "11:23:15"))
as.difftime(c("3:20", "23:15", "2:"), format= "%H:%M")# 3rd gives NA
(z <- as.difftime(c(0,30,60), units="mins"))
as.numeric(z, units="secs")
as.numeric(z, units="hours")
format(z)
```

---

dim *Dimensions of an Object*

Description

Retrieve or set the dimension of an object.

Usage

```
dim(x)
dim(x) <- value
```

Arguments

x	an R object, for example a matrix, array or data frame.
value	For the default method, either NULL or a numeric vector, which is coerced to integer (by truncation).

Details

The functions dim and dim<- are generic.

dim has a method for data.frames, which returns the lengths of the row.names attribute of x and of x (as the numbers of rows and columns respectitvely).

Value

For an array (and hence in particular, for a matrix) dim retrieves the dim attribute of the object. It is NULL or a vector of mode integer.

The replacement method changes the "dim" attribute (provided the new value is compatible) and removes any "dimnames" *and* "names" attributes.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

ncol, nrow and dimnames.

Examples

```
x <- 1:12 ; dim(x) <- c(3,4)
x

# simple versions of nrow and ncol could be defined as follows
nrow0 <- function(x) dim(x)[1]
ncol0 <- function(x) dim(x)[2]
```

---

dimnames *Dimnames of an Object*

Description

Retrieve or set the dimnames of an object.

Usage

```
dimnames(x)
dimnames(x) <- value
```

Arguments

x	an R object, for example a matrix, array or data frame.
value	a possible value for dimnames(x): see the 'Value' section.

Details

The functions dimnames and dimnames<- are generic.

For an array (and hence in particular, for a matrix), they retrieve or set the dimnames attribute (see attributes) of the object. A list value can have names, and these will be used to label the dimensions of the array where appropriate.

The replacement method for arrays/matrices coerces vector and factor elements of value to character, but does not dispatch methods for as.character. It coerces zero-length elements to NULL, and a zero-length list to NULL. If value is a list shorter than the number of dimensions, as from R 2.8.0 it is extended with NULLs to the needed length.

Both have methods for data frames. The dimnames of a data frame are its row.names and its names. For the replacement method each component of value will be coerced by as.character.

For a 1D matrix the names are the same thing as the (only) component of the dimnames.

Value

The dimnames of a matrix or array can be NULL or a list of the same length as dim(x). If a list, its components are either NULL or a character vector with positive length of the appropriate dimension of x. The list can be named.

For the "data.frame" method both dimnames are character vectors, and the rownames must contain no duplicates nor missing values.

Note

Setting components of the dimnames, e.g. `dimnames(A)[[1]] <- value` is a common paradigm, but note that it will not work if the value assigned is `NULL`. Use `rownames` instead, or (as it does) manipulate the whole dimnames list.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`rownames`, `colnames`; `array`, `matrix`, `data.frame`.

Examples

```
## simple versions of rownames and colnames
## could be defined as follows
rownames0 <- function(x) dimnames(x)[[1]]
colnames0 <- function(x) dimnames(x)[[2]]
```

---

do.call *Execute a Function Call*

Description

`do.call` constructs and executes a function call from a name or a function and a list of arguments to be passed to it.

Usage

```
do.call(what, args, quote = FALSE, envir = parent.frame())
```

Arguments

`what`	either a function or a non-empty character string naming the function to be called.
`args`	a *list* of arguments to the function call. The `names` attribute of `args` gives the argument names.
`quote`	a logical value indicating whether to quote the arguments.
`envir`	an environment within which to evaluate the call. This will be most useful if `what` is a character string and the arguments are symbols or quoted expressions.

Details

If `quote` is `FALSE`, the default, then the arguments are evaluated (in the calling environment, not `envir`.). If `quote` is `TRUE` then each argument is quoted (see `quote`) so that the effect of argument evaluation is to remove the quotes – leaving the original arguments unevaluated when the call is constructed.

The behavior of some functions, such as `substitute`, will not be the same for functions evaluated using `do.call` as if they were evaluated from the interpreter. The precise semantics are currently undefined and subject to change.

Value

The result of the (evaluated) function call.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`call` which creates an unevaluated call.

Examples

```
do.call("complex", list(imag = 1:3))

## if we already have a list (e.g. a data frame)
## we need c() to add further arguments
tmp <- expand.grid(letters[1:2], 1:3, c("+", "-"))
do.call("paste", c(tmp, sep=""))

do.call(paste, list(as.name("A"), as.name("B")), quote=TRUE)

## examples of where objects will be found.
A <- 2
f <- function(x) print(x^2)
env <- new.env()
assign("A", 10, envir = env)
assign("f", f, envir = env)
f <- function(x) print(x)
f(A)                                    # 2
do.call("f", list(A))                   # 2
do.call("f", list(A), envir=env)        # 4
do.call(f, list(A), envir=env)          # 2
do.call("f", list(quote(A)), envir=env) # 100
do.call(f, list(quote(A)), envir=env)   # 10
do.call("f", list(as.name("A")), envir=env) # 100

eval(call("f", A))                      # 2
eval(call("f", quote(A)))               # 2
eval(call("f", A), envir=env)           # 4
eval(call("f", quote(A)), envir=env)    # 100
```

`double` *Double-Precision Vectors*

Description

Create, coerce to or test for a double-precision vector.

Usage

```
double(length = 0)
as.double(x, ...)
is.double(x)

single(length = 0)
as.single(x, ...)
```

Arguments

length	desired length.
x	object to be coerced or tested.
...	further arguments passed to or from other methods.

Details

double creates a double-precision vector of the specified length. The elements of the vector are all equal to 0. It is identical to numeric (and real).

as.double is a generic function. It is identical to as.numeric (and as.real). Methods should return an object of base type "double".

is.double is a test of double type.

R *has no single precision data type. All real numbers are stored in double precision format.* The functions as.single and single are identical to as.double and double except they set the attribute Csingle that is used in the .C and .Fortran interface, and they are intended only to be used in that context.

Value

double creates a double-precision vector of the specified length. The elements of the vector are all equal to 0.

as.double attempts to coerce its argument to be of double type: like as.vector it strips attributes including names. (To ensure that an object is of double type without stripping attributes, use storage.mode.) Character strings containing either a decimal representation or a hexadecimal representation (starting with 0x or 0X) can be converted. as.double for factors yields the codes underlying the factor levels, not the numeric representation of the labels, see also factor.

is.double returns TRUE or FALSE depending on whether its argument is of double type or not.

Note on names

It is a historical anomaly that R has three names for its floating-point vectors, double, numeric and real.

double is the name of the type. numeric is the name of the mode and also of the implicit class. As an S4 formal class, use "numeric" (there was a formal class "double" prior to R 2.7.0).

real is deprecated and should not be used in new code.

The potential confusion is that R has used *mode* "numeric" to mean 'double or integer', which conflicts with the S4 usage. Thus is.numeric tests the mode, not the class, but as.numeric (which is identical to as.double) coerces to the class.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`integer`, `numeric`, `storage.mode`.

Examples

```
is.double(1)
all(double(3) == 0)
```

`dput` *Write an Object to a File or Recreate it*

Description

Writes an ASCII text representation of an R object to a file or connection, or uses one to recreate the object.

Usage

```
dput(x, file = "",
     control = c("keepNA", "keepInteger", "showAttributes"))

dget(file)
```

Arguments

`x`	an object.
`file`	either a character string naming a file or a connection. `""` indicates output to the console.
`control`	character vector indicating deparsing options. See `.deparseOpts` for their description.

Details

`dput` opens `file` and deparses the object `x` into that file. The object name is not written (unlike `dump`). If `x` is a function the associated environment is stripped. Hence scoping information can be lost.

Deparsing an object is difficult, and not always possible. With the default `control`, `dput()` attempts to deparse in a way that is readable, but for more complex or unusual objects (see `dump`, not likely to be parsed as identical to the original. Use `control = "all"` for the most complete deparsing; use `control = NULL` for the simplest deparsing, not even including attributes.

`dput` will warn if fewer characters were written to a file than expected, which may indicate a full or corrupt file system.

To display saved source rather than deparsing the internal representation include `"useSource"` in `control`. R currently saves source only for function definitions.

Value

For `dput`, the first argument invisibly.

For `dget`, the object created.

Note

To avoid the risk of a source attribute out of sync with the actual function definition, the source attribute of a function will never be written as an attribute.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`deparse`, `dump`, `write`.

Examples

```
## Write an ASCII version of mean to the file "foo"
dput(mean, "foo")
## And read it back into 'bar'
bar <- dget("foo")
unlink("foo")
## Create a function with comments
baz <- function(x) {
  # Subtract from one
  1-x
}
## and display it
dput(baz)
## and now display the saved source
dput(baz, control = "useSource")
```

`drop` *Drop Redundant Extent Information*

Description

Delete the dimensions of an array which have only one level.

Usage

```
drop(x)
```

Arguments

`x`	an array (including a matrix).

Value

If `x` is an object with a `dim` attribute (e.g., a matrix or `array`), then `drop` returns an object like `x`, but with any extents of length one removed. Any accompanying `dimnames` attribute is adjusted and returned with `x`: if the result is a vector the `names` are taken from the `dimnames` (if any). If the result is a length-one vector, the names are taken from the first dimension with a dimname.

Array subsetting (`[`) performs this reduction unless used with `drop = FALSE`, but sometimes it is useful to invoke `drop` directly.

See Also

`drop1` which is used for dropping terms in models.

Examples

```
dim(drop(array(1:12, dim=c(1,3,1,1,2,1,2))))# = 3 2 2
drop(1:3 %*% 2:4)# scalar product
```

`dump` *Text Representations of R Objects*

Description

This function takes a vector of names of R objects and produces text representations of the objects on a file or connection. A `dump` file can usually be `source`d into another R (or S) session.

Usage

```
dump(list, file = "dumpdata.R", append = FALSE,
     control = "all", envir = parent.frame(), evaluate = TRUE)
```

Arguments

`list`	character. The names of one or more R objects to be dumped.
`file`	either a character string naming a file or a connection. `""` indicates output to the console.
`append`	if `TRUE` and `file` is a character string, output will be appended to `file`; otherwise, it will overwrite the contents of `file`.
`control`	character vector indicating deparsing options. See `.deparseOpts` for their description.
`envir`	the environment to search for objects.
`evaluate`	logical. Should promises be evaluated?

Details

If some of the objects named do not exist (in scope), they are omitted, with a warning. If `file` is a file and no objects exist then no file is created.

`source`ing may not produce an identical copy of `dump`ed objects. A warning is issued if it is likely that problems will arise, for example when dumping exotic or complex objects (see the Note).

`dump` will also warn if fewer characters were written to a file than expected, which may indicate a full or corrupt file system.

A `dump` file can be `source`d into another R (or perhaps S) session, but the function `save` is designed to be used for transporting R data, and will work with R objects that `dump` does not handle.

To produce a more readable representation of an object, use `control = NULL`. This will skip attributes, and will make other simplifications that make `source` less likely to produce an identical copy. See `deparse` for details.

To deparse the internal representation of a function rather than displaying the saved source, use `control = c("keepInteger", "warnIncomplete", "keepNA")`. This will lose all formatting and comments, but may be useful in those cases where the saved source is no longer correct.

Promises will normally only be encountered by users as a result of lazy-loading (when the default `evaluate = TRUE` is essential) and after the use of `delayedAssign`, when `evaluate = FALSE` might be intended.

Value

An invisible character vector containing the names of the objects which were dumped.

Note

As `dump` is defined in the base name space, the **base** package will be searched *before* the global environment unless `dump` is called from the top level prompt or the `envir` argument is given explicitly.

To avoid the risk of a source attribute becoming out of sync with the actual function definition, the source attribute of a function will never be dumped as an attribute.

Currently environments, external pointers, weak references and objects of type `S4` are not deparsed in a way that can be `source`d. In addition, language objects are deparsed in a simple way whatever the value of `control`, and this includes not dumping their attributes (which will result in a warning).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`dput`, `dget`, `write`.
`save` for a more reliable way to save R objects.

Examples

```
x <- 1; y <- 1:10
dump(ls(pattern = '^[xyz]'), "xyz.Rdmped")
print(.Last.value)
unlink("xyz.Rdmped")
```

---

duplicated *Determine Duplicate Elements*

Description

Determines which elements of a vector or data frame are duplicates of elements with smaller subscripts, and returns a logical vector indicating which elements (rows) are duplicates.

Usage

```
duplicated(x, incomparables = FALSE, ...)

## Default S3 method:
duplicated(x, incomparables = FALSE,
           fromLast = FALSE, ...)

## S3 method for class 'array':
duplicated(x, incomparables = FALSE, MARGIN = 1,
           fromLast = FALSE, ...)
```

Arguments

`x`	a vector or a data frame or an array or `NULL`.
`incomparables`	a vector of values that cannot be compared. `FALSE` is a special value, meaning that all values can be compared, and may be the only value accepted for methods other than the default. It will be coerced internally to the same type as `x`.
`fromLast`	logical indicating if duplication should be considered from the reverse side, i.e., the last (or rightmost) of identical elements would correspond to `duplicated=FALSE`.
`...`	arguments for particular methods.
`MARGIN`	the array margin to be held fixed: see `apply`.

Details

This is a generic function with methods for vectors (including lists), data frames and arrays (including matrices).

`duplicated(x, fromLast=TRUE)` is equivalent to but faster than `rev(duplicated(rev(x)))`.

The data frame method works by pasting together a character representation of the rows separated by `\r`, so may be imperfect if the data frame has characters with embedded carriage returns or columns which do not reliably map to characters.

The array method calculates for each element of the sub-array specified by `MARGIN` if the remaining dimensions are identical to those for an earlier (or later, when `fromLast=TRUE`) element (in row-major order). This would most commonly be used to find duplicated rows (the default) or columns (with `MARGIN = 2`).

Missing values are regarded as equal, but `NaN` is not equal to `NA_real_`.

Values in `incomparables` will never be marked as duplicated. This is intended to be used for a fairly small set of values and will not be efficient for a very large set.

Value

For a vector input, a logical vector of the same length as `x`. For a data frame, a logical vector with one element for each row. For a matrix or array, a logical array with the same dimensions and dimnames.

Warning

Using this for lists is potentially slow, especially if the elements are not atomic vectors (see `vector`) or differ only in their attributes. In the worst case it is $O(n^2)$.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`unique`.

Examples

```
x <- c(9:20, 1:5, 3:7, 0:8)
## extract unique elements
(xu <- x[!duplicated(x)])
## similar, but not the same:
(xu2 <- x[!duplicated(x, fromLast = TRUE)])

## xu == unique(x) but unique(x) is more efficient
stopifnot(identical(xu,  unique(x)),
          identical(xu2, unique(x, fromLast = TRUE)))

duplicated(iris)[140:143]

duplicated(iris3, MARGIN = c(1, 3))
```

---

dyn.load *Foreign Function Interface*

Description

Load or unload DLLs (also known as shared objects), and test whether a C function or Fortran subroutine is available.

Usage

```
dyn.load(x, local = TRUE, now = TRUE, ...)
dyn.unload(x)

is.loaded(symbol, PACKAGE = "", type = "")
```

Arguments

x	a character string giving the pathname to a DLL, also known as a dynamic shared object. (See 'Details' for what these terms mean.)
local	a logical value controlling whether the symbols in the DLL are stored in their own local table and not shared across DLLs, or added to the global symbol table. Whether this has any effect is system-dependent.
now	a logical controlling whether all symbols are resolved (and relocated) immediately the library is loaded or deferred until they are used. This control is useful for developers testing whether a library is complete and has all the necessary symbols, and for users to ignore missing symbols. Whether this has any effect is system-dependent.
...	other arguments for future expansion.
symbol	a character string giving a symbol name.
PACKAGE	if supplied, confine the search for the name to the DLL given by this argument (plus the conventional extension, '.so', '.sl', '.dll', ...). This is intended to add safety for packages, which can ensure by using this argument that no other package can override their external symbols. Use PACKAGE="base" for symbols linked in to R. This is used in the same way as in .C, .Call, .Fortran and .External functions
type	The type of symbol to look for: can be any ("", the default), "Fortran", "Call" or "External".

Details

The objects dyn.load loads are called 'dynamically loadable libraries' (abbreviated to 'DLL' on all platforms except Mac OS X, which unfortunately uses the term for a different sort of sobject. On Unix-alikes they are also called 'dynamic shared objects' ('DSO'), or 'shared objects' for short. (The POSIX standards use 'executable object file', but no one else does.)

See 'See Also' and the *Writing R Extensions* and *R Installation and Administration* manuals for how to create and install a suitable DLL. Note that unlike some versions of S-PLUS, dyn.load does not load an object ('.o') file but a shared object or DLL.

Unfortunately a very few platforms (e.g. Compaq Tru64) do not handle the PACKAGE argument correctly, and may incorrectly find symbols linked into R.

The additional arguments to dyn.load mirror the different aspects of the mode argument to the dlopen() routine on POSIX systems. They are available so that users can exercise greater control over the loading process for an individual library. In general, the defaults values are appropriate and you should override them only if there is good reason and you understand the implications.

The local argument allows one to control whether the symbols in the DLL being attached are visible to other DLLs. While maintaining the symbols in their own name space is good practice, the ability to share symbols across related 'chapters' is useful in many cases. Additionally, on certain platforms and versions of an operating system, certain libraries must have their symbols loaded globally to successfully resolve all symbols.

One should be careful of the potential side-effect of using lazy loading via the `now` argument as `FALSE`. If a routine is called that has a missing symbol, the process will terminate immediately and unsaved session variables will be lost. The intended use is for library developers to call specify a value `TRUE` to check that all symbols are actually resolved and for regular users to all with `FALSE` so that missing symbols can be ignored and the available ones can be called.

The initial motivation for adding these was to avoid such termination in the `_init()` routines of the Java virtual machine library. However, symbols loaded locally may not be (read probably) available to other DLLs. Those added to the global table are available to all other elements of the application and so can be shared across two different DLLs.

Some systems do not provide (explicit) support for local/global and lazy/eager symbol resolution. This can be the source of subtle bugs. One can arrange to have warning messages emitted when unsupported options are used. This is done by setting either of the options `verbose` or `warn` to be non-zero via the `options` function. Currently, we know of only 2 platforms that do not provide a value for local load (RTLD_LOCAL). These are IRIX6.4 and unpatched versions of Solaris 2.5.1.

There is a short discussion of these additional arguments with some example code available at http://cm.bell-labs.com/stat/duncan/R/dynload.

Value

The function `dyn.load` is used for its side effect which links the specified DLL to the executing R image. Calls to `.C`, `.Call`, `.Fortran` and `.External` can then be used to execute compiled C functions or Fortran subroutines contained in the library. The return value of `dyn.load` is an object of class `DLLInfo`. See `getLoadedDLLs` for information about this class.

The function `dyn.unload` unlinks the DLL.

`is.loaded` checks if the symbol name is loaded and hence available for use in `.C` or `.Fortran` or `.Call` or `.External`. It will succeed if any one of the four calling functions would succeed in using the entry point unless `type` is specified. (See `.Fortran` for how Fortran symbols are mapped.)

Warning

Do not use `dyn.unload` on a DLL loaded by `library.dynam`: use `library.dynam.unload`. This is needed for system houskeeping.

Note

`is.loaded` requires the name you would give to `.C` etc and **not** (as in S) that remapped by defunct functions `symbol.C` or `symbol.For`.

The creation of DLLs and the runtime linking of them into executing programs is very platform dependent. In recent years there has been some simplification in the process because the C subroutine call `dlopen` has become the POSIX standard for doing this. Under Unix-alikes `dyn.load` uses the `dlopen` mechanism and should work on all platforms which support it. On Windows it uses the standard mechanism (`LoadLibrary`) for loading DLLs.

The original code for loading DLLs in Unix-alikes was provided by Heiner Schwarte. The compatibility code for HP-UX was provided by Luke Tierney.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`library.dynam` to be used inside a package's `.First.lib` initialization.

`SHLIB` for how to create suitable DLLs.

`.C`, `.Fortran`, `.External`, `.Call`.

Examples

```
is.loaded("hcass2") #-> probably TRUE, as stats is loaded
is.loaded("supsmu") # Fortran entry point in stats
is.loaded("supsmu", "stats", "Fortran")
is.loaded("PDF", type = "External")
```

`eapply` *Apply a Function over values in an environment*

Description

`eapply` applies `FUN` to the named values from an environment and returns the results as a list. The user can request that all named objects are used (normally names that begin with a dot are not). The output is not sorted and no parent environments are searched.

Usage

```
eapply(env, FUN, ..., all.names = FALSE)
```

Arguments

`env`	environment to be used.
`FUN`	the function to be applied, found *via* `match.fun`. In the case of functions like `+`, `%*%`, etc., the function name must be backquoted or quoted.
`...`	optional arguments to `FUN`.
`all.names`	a logical indicating whether to apply the function to all values

See Also

`lapply`.

Examples

```
require(utils); require(stats)

env <- new.env()
env$a <- 1:10
env$beta <- exp(-3:3)
env$logic <- c(TRUE,FALSE,FALSE,TRUE)
# what have we there?
eapply(env, str)

# compute the mean for each list element
eapply(env, mean)
# median and quartiles for each list element
eapply(env, quantile, probs = 1:3/4)
eapply(env, quantile)
```

`eigen` *Spectral Decomposition of a Matrix*

Description

Computes eigenvalues and eigenvectors of real or complex matrices.

Usage

```
eigen(x, symmetric, only.values = FALSE, EISPACK = FALSE)
```

Arguments

`x`	a matrix whose spectral decomposition is to be computed.
`symmetric`	if `TRUE`, the matrix is assumed to be symmetric (or Hermitian if complex) and only its lower triangle (diagonal included) is used. If `symmetric` is not specified, the matrix is inspected for symmetry.
`only.values`	if `TRUE`, only the eigenvalues are computed and returned, otherwise both eigenvalues and eigenvectors are returned.
`EISPACK`	logical. Should EISPACK be used (for compatibility with R < 1.7.0)?

Details

By default `eigen` uses the LAPACK routines DSYEVR, DGEEV, ZHEEV and ZGEEV whereas `eigen(EISPACK=TRUE)` provides an interface to the EISPACK routines `RS`, `RG`, `CH` and `CG`.

If `symmetric` is unspecified, the code attempts to determine if the matrix is symmetric up to plausible numerical inaccuracies. It is faster and surer to set the value yourself.

`eigen` is preferred to `eigen(EISPACK = TRUE)` for new projects, but its eigenvectors may differ in sign and (in the asymmetric case) in normalization. (They may also differ between methods and between platforms.)

Computing the eigenvectors is the slow part for large matrices.

Computing the eigendecomposition of a matrix is subject to errors on a real-world computer: the definitive analysis is Wilkinson (1965). All you can hope for is a solution to a problem suitably close to `x`. So even though a real asymmetric `x` may have an algebraic solution with repeated real eigenvalues, the computed solution may be of a similar matrix with complex conjugate pairs of eigenvalues.

Value

The spectral decomposition of `x` is returned as components of a list with components

`values`	a vector containing the p eigenvalues of `x`, sorted in *decreasing* order, according to `Mod(values)` in the asymmetric case when they might be complex (even for real matrices). For real asymmetric matrices the vector will be complex only if complex conjugate pairs of eigenvalues are detected.
`vectors`	either a $p \times p$ matrix whose columns contain the eigenvectors of `x`, or `NULL` if `only.values` is `TRUE`.

For `eigen(, symmetric = FALSE, EISPACK =TRUE)` the choice of length of the eigenvectors is not defined by EISPACK. In all other cases the vectors are normalized to unit length.

Recall that the eigenvectors are only defined up to a constant: even when the length is specified they are still only defined up to a scalar of modulus one (the sign for real matrices).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Smith, B. T, Boyle, J. M., Dongarra, J. J., Garbow, B. S., Ikebe,Y., Klema, V., and Moler, C. B. (1976). *Matrix Eigensystems Routines – EISPACK Guide*. Springer-Verlag Lecture Notes in Computer Science.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at http://www.netlib.org/lapack/lug/lapack_lug.html.

Wilkinson, J. H. (1965) *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford.

See Also

`svd`, a generalization of `eigen`; `qr`, and `chol` for related decompositions.

To compute the determinant of a matrix, the `qr` decomposition is much more efficient: `det`.

Examples

```
eigen(cbind(c(1,-1),c(-1,1)))
eigen(cbind(c(1,-1),c(-1,1)), symmetric = FALSE)
# same (different algorithm).

eigen(cbind(1,c(1,-1)), only.values = TRUE)
eigen(cbind(-1,2:1)) # complex values
eigen(print(cbind(c(0,1i), c(-1i,0))))# Hermite ==> real Eigen values
## 3 x 3:
eigen(cbind( 1,3:1,1:3))
eigen(cbind(-1,c(1:2,0),0:2)) # complex values
```

---

encodeString *Encode Character Vector as for Printing*

Description

`encodeString` escapes the strings in a character vector in the same way `print.default` does, and optionally fits the encoded strings within a field width.

Usage

```
encodeString(x, width = 0, quote = "", na.encode = TRUE,
             justify = c("left", "right", "centre", "none"))
```

Arguments

x	A character vector, or an object that can be coerced to one by as.character.
width	integer: the minimum field width. If NULL or NA, this is taken to be the largest field width needed for any element of x.
quote	character: quoting character, if any.
na.encode	logical: should NA strings be encoded?
justify	character: partial matches are allowed. If padding to the minimum field width is needed, how should spaces be inserted? justify == "none" is equivalent to width = 0, for consistency with format.default.

Details

This escapes backslash and the control characters \a (bell), \b (backspace), \f (formfeed), \n (line feed), \r (carriage return), \t (tab) and \v (vertical tab) as well as any non-printable characters in a single-byte locale, which are printed in octal notation (\xyz with leading zeroes). (Which characters are non-printable depends on the current locale.) See print.default for how non-printable characters are handled in multi-byte locales.

If quote is a single or double quote any embedded quote of the same type is escaped. Note that justification is of the quoted string, hence spaces are added outside the quotes.

Value

A character vector of the same length as x, with the same attributes (including names and dimensions) but with no class set.

Note

The default for width is different from format.default, which does similar things for character vectors but without encoding using escapes.

See Also

print.default

Examples

```
x <- "ab\bc\ndef"
print(x)
cat(x) # interprets escapes
cat(encodeString(x), "\n", sep="") # similar to print()

factor(x) # makes use of this to print the levels

x <- c("a", "ab", "abcde")
encodeString(x, width = NA) # left justification
encodeString(x, width = NA, justify = "c")
encodeString(x, width = NA, justify = "r")
encodeString(x, width = NA, quote = "'", justify = "r")
```

`Encoding` *Read or Set the Declared Encodings for a Character Vector*

Description

Read or set the declared encodings for a character vector.

Usage

```
Encoding(x)

Encoding(x) <- value
```

Arguments

`x`	A character vector.
`value`	A character vector of positive length.

Details

Character strings in R can be declared to be in `"latin1"` or `"UTF-8"`. These declarations can be read by `Encoding`, which will return a character vector of values `"latin1"`, `"UTF-8"` or `"unknown"`, or set, when `value` is recycled as needed and other values are silently treated as `"unknown"`. As from R 2.8.0, ASCII strings will never be marked with a declared encoding, since their representation is the same in all encodings.

There are other ways for character strings to acquire a declared encoding apart from explicitly setting it. Functions `scan`, `read.table`, `readLines` and `parse` have an `encoding` argument that is used to declare encodings, `iconv` declares encodings from its `from` argument, and console input in suitable locales is also declared. `intToUtf8` declares its output as `"UTF-8"`, and output text connections are marked if running in a suitable locale.

Most character manipulation functions will set the encoding on output strings if it was declared on the corresponding input. These include `chartr`, `strsplit`, `strtrim`, `substr`, `tolower` and `toupper` as well as `sub(useBytes = FALSE)` and `gsub(useBytes = FALSE)`. (Also, under some circumstances `paste` will set an encoding.) Note that such functions do not *preserve* the encoding, but if they know the input encoding and that the string has been successfully re-encoded to the current encoding, they mark the output with the latter (if it is `"latin1"` or `"UTF-8"`).

As from R 2.7.0 `substr` does preserve the encoding, and `chartr`, `tolower` and `toupper` preserve UTF-8 encoding on systems with Unicode wide characters. With their `fixed` and `perl` options, `strsplit`, `sub` and `gsub` will give a UTF-8 result if any of the inputs are UTF-8.

As from R 2.8.0 `paste` and `sprintf` return a UTF-8 encoded element if any of the inputs to that element are UTF-8.

Value

A character vector.

Examples

```
## x is intended to be in latin1
x <- "fa\xE7ile"
Encoding(x)
Encoding(x) <- "latin1"
x
xx <- iconv(x, "latin1", "UTF-8")
Encoding(c(x, xx))
c(x, xx)
```

---

environment *Environment Access*

Description

Get, set, test for and create environments.

Usage

```
environment(fun = NULL)
environment(fun) <- value

is.environment(x)

.GlobalEnv
globalenv()
.BaseNamespaceEnv

emptyenv()
baseenv()

new.env(hash = FALSE, parent = parent.frame(), size = 29L)

parent.env(env)
parent.env(env) <- value

environmentName(env)

env.profile(env)
```

Arguments

fun	a function, a formula, or NULL, which is the default.
value	an environment to associate with the function
x	an arbitrary R object.
hash	a logical, if TRUE the environment will be hashed
parent	an environment to be used as the enclosure of the environment created.
env	an environment
size	an integer specifying the initial size for a hashed environment. An internal default value will be used if size is NA or zero. This argument is ignored if hash is FALSE.

Details

Environments consist of a *frame*, or collection of named objects, and a pointer to an *enclosing environment*. The most common example is the frame of variables local to a function call; its enclosure is the environment where the function was defined. The enclosing environment is distinguished from the *parent frame*: the latter (returned by `parent.frame`) refers to the environment of the caller of a function.

When `get` or `exists` search an environment with the default `inherits = TRUE`, they look for the variable in the frame, then in the enclosing frame, and so on.

The global environment `.GlobalEnv`, more often known as the user's workspace, is the first item on the search path. It can also be accessed by `globalenv()`. On the search path, each item's enclosure is the next item.

The object `.BaseNamespaceEnv` is the name space environment for the base package. The environment of the base package itself is available as `baseenv()`. The ultimate enclosure of any environment is the empty environment `emptyenv()`, to which nothing may be assigned. If one follows the `parent.env()` chain of enclosures back far enough from any environment, eventually one reaches the empty environment.

The replacement function `parent.env<-` is extremely dangerous as it can be used to destructively change environments in ways that violate assumptions made by the internal C code. It may be removed in the near future.

Value

If `fun` is a function or a formula then `environment(fun)` returns the environment associated with that function or formula. If `fun` is `NULL` then the current evaluation environment is returned.

The replacement form sets the environment of the function or formula `fun` to the `value` given.

`is.environment(obj)` returns `TRUE` if and only if `obj` is an `environment`.

`new.env` returns a new (empty) environment enclosed in the parent's environment, by default.

`parent.env` returns the parent environment of its argument.

`parent.env<-` sets the enclosing environment of its first argument.

`environmentName` returns a character string, that given when the environment is printed or `""` if it is not a named environment.

`env.profile` returns a list with the following components: `size` the number of chains that can be stored in the hash table, `nchains` the number of non-empty chains in the table (as reported by HASHPRI), and `counts` an integer vector giving the length of each chain (zero for empty chains). This function is intended to assess the performance of hashed environments. When `env` is a non-hashed environment, `NULL` is returned.

See Also

The `envir` argument of `eval`, `get`, and `exists`.

`ls` may be used to view the objects in an environment, and hence `ls.str` may be useful for an overview.

`sys.source` can be used to populate an environment.

Examples

```
f <- function() "top level function"

##-- all three give the same:
```

```
environment()
environment(f)
.GlobalEnv

ls(envir=environment(stats::approxfun(1:2,1:2, method="const")))

is.environment(.GlobalEnv) # TRUE

e1 <- new.env(parent = baseenv())  # this one has enclosure package:base.
e2 <- new.env(parent = e1)
assign("a", 3, envir=e1)
ls(e1)
ls(e2)
exists("a", envir=e2)   # this succeeds by inheritance
exists("a", envir=e2, inherits = FALSE)
exists("+", envir=e2)   # this succeeds by inheritance
```

EnvVar *Environment Variables*

Description

Details of some of the environment variables which affect an R session.

Details

It is impossible to list all the environment variables which can affect an R session: some affect the OS system functions which R uses, and others will affect add-on packages. But here are notes on some of the more important ones. Those that set the defaults for options are consulted only at startup (as are some of the others).

HOME: The user's 'home' directory.

LANGUAGE: Optional. The language(s) to be used for message translations. This is consulted when needed.

LC_ALL: (etc) Optional. Use to set various aspects of the locale – see `Sys.getlocale`. Consulted at startup.

R_BATCH: Optional – set in a batch session.

R_COMPLETION: Optional. If set to `FALSE`, comand-line completion is not used. (Not used by Mac OS GUI.)

R_DEFAULT_PACKAGES: A comma-separated list of packages which are to be loaded in every session. See `options`.

R_DOC_DIR: The location of the R 'doc' directory. Set by R.

R_DVIPSCMD: The path to `dvips`. Defaults to the value of `DVIPS`, and if that is unset to a value determined when R was built. Used by `R CMD Rd2dvi`, and at startup to set the default for `options("latexcmd")`, used by `help(offline = TRUE`.

R_ENVIRON: Optional. The path to the site environment file: see Startup. Consulted at startup.

R_GSCMD: Optional. The path to GhostScript, used by `dev2bitmap`.

R_HISTFILE: Optional. The path of the history file: see Startup. Consulted at startup and when the history is saved.

R_HISTSIZE: Optional. The maximum size of the history file, in lines. Exactly how this is used depends on the interface. For the `readline` command-line interface it takes effect when the history is saved (by `savehistory` or at the end of a session).

R_HOME: The top-level directory of the R installation: see `R.home`. Set by R.

R_INCLUDE_DIR: The location of the R 'include' directory. Set by R.

R_LATEXCMD: The path to `latex`. Defaults to the value of `LATEX`, and if that is unset to a value determined when R was built. Used by `R CMD Rd2dvi`, and at startup to set the default for `options("latexcmd")`, used by `help(offline = TRUE`.

R_LIBS: Optional. Used for initial setting of `.libPaths`.

R_LIBS_SITE: Optional. Used for initial setting of `.libPaths`.

R_LIBS_USER: Optional. Used for initial setting of `.libPaths`.

R_MAKEINDEXCMD: The path to `makeindex`. Defaults at startup to the value of `MAKEINDEX`, and if that is unset to a value determined when R was built. Used by `R CMD Rd2dvi`.

R_PAPERSIZE: Optional. Use to set the default for `options("papersize")`, e.g. used by `pdf`, `postscript` and `help(offline = TRUE)`.

R_PDFLATEXCMD: The path to `pdflatex`. Defaults at startup to the value of `PDFLATEX`, and if that is unset to a value determined when R was built. Used by `R CMD Rd2dvi`.

R_PDFVIEWER: The path to the default PDF viewer. Used by `R CMD Rd2dvi`.

R_PLATFORM: The platform – a string of the form *cpu–vendor–os*, see `R.Version`.

R_PROFILE: Optional. The path to the site profile file: see Startup. Consulted at startup.

R_RD4DVI: Options for `latex` processing of `Rd` files. Used by `R CMD Rd2dvi` and `help(offline = TRUE)`.

R_RD4PDF: Options for `pdflatex` processing of `Rd` files. Used by `R CMD Rd2dvi`.

R_SHARE_DIR: The location of the R 'share' directory. Set by R.

R_UNZIPCMD: The path to `unzip`. Sets the default value for `options("unzip")`.

R_ZIPCMD: The path to `zip`.

TMPDIR, TMP, TEMP: Consulted (in that order) when setting the temporary directory. See `tempdir`.

TZ: Optional. The current timezone. See `Sys.timezone` for the system-specific formats. Consulted as needed.

no_proxy, http_proxy, ftp_proxy: (and more). Optional. Settings for `download.file`: see its help for further details.

Unix-specific

Some variables set on Unix-alikes, and not (in general) on Windows.

DISPLAY: Optional: used by `X11`, Tk (in package **tcltk**), the data editor and various packages.

EDITOR: The path to the default editor.

PAGER: The path to the default pager.

R_BROWSER: The path to the default browser. Used to set the default value of `options("browser")`.

R_PDFVIEWER: The path to the default PDF viewer. Used by `R CMD Rd2dvi`.

R_PRINTCMD: Sets the default for `options("printcmd")`, which sets the default print command to be used by `postscript`.

See Also

`Sys.getenv` and `Sys.setenv` to read and set environmental variables in an R session.

`eval` *Evaluate an (Unevaluated) Expression*

Description

Evaluate an R expression in a specified environment.

Usage

```
eval(expr, envir = parent.frame(),
           enclos = if(is.list(envir) || is.pairlist(envir))
                       parent.frame() else baseenv())
evalq(expr, envir, enclos)
eval.parent(expr, n = 1)
local(expr, envir = new.env())
```

Arguments

`expr`	an object to be evaluated. See 'Details'.
`envir`	the `environment` in which `expr` is to be evaluated. May also be `NULL`, a list, a data frame, a pairlist or an integer as specified to `sys.call`.
`enclos`	Relevant when `envir` is a (pair)list or a data frame. Specifies the enclosure, i.e., where R looks for objects not found in `envir`. This can be `NULL` (interpreted as the base package environment) or an environment.
`n`	number of parent generations to go back

Details

`eval` evaluates the `expr` argument in the environment specified by `envir` and returns the computed value. If `envir` is not specified, then the default is `parent.frame()` (the environment where the call to `eval` was made).

Objects to be evaluated can be of types `call` or `expression` or name (when the name is looked up in the current scope and its binding is evaluated), a promise or any of the basic types such as vectors, functions and environments (which are returned unchanged).

The `evalq` form is equivalent to `eval(quote(expr), ...)`. `eval` evaluates its first argument in the current scope before passing it to the evaluator: `evalq` avoids this.

`eval.parent(expr, n)` is a shorthand for `eval(expr, parent.frame(n))`.

If `envir` is a list (such as a data frame) or pairlist, it is copied into a temporary environment (with enclosure `enclos`), and the temporary environment is used for evaluation. So if `expr` changes any of the components named in the (pair)list, the changes are lost.

If `envir` is `NULL` it is interpreted as an empty list so no values could be found in `envir` and look-up goes directly to `enclos`.

`local` evaluates an expression in a local environment. It is equivalent to `evalq` except that its default argument creates a new, empty environment. This is useful to create anonymous recursive functions and as a kind of limited name space feature since variables defined in the environment are not visible from the outside.

Value

The result of evaluating the object: for an expression vector this it the result of evaluating the last elements.

Note

Due to the difference in scoping rules, there are some differences between R and S in this area. In particular, the default enclosure in S is the global environment.

When evaluating expressions in data frames that has been passed as argument to a function, the relevant enclosure is often the caller's environment, i.e., one needs `eval(x, data, parent.frame())`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`eval` only.)

See Also

`expression`, `quote`, `sys.frame`, `parent.frame`, `environment`.

Further, `force` to *force* evaluation, typically of function arguments.

Examples

```
eval(2 ^ 2 ^ 3)
mEx <- expression(2^2^3); mEx; 1 + eval(mEx)
eval({ xx <- pi; xx^2}) ; xx

a <- 3 ; aa <- 4 ; evalq(evalq(a+b+aa, list(a=1)), list(b=5)) # == 10
a <- 3 ; aa <- 4 ; evalq(evalq(a+b+aa, -1), list(b=5))        # == 12

ev <- function() {
   e1 <- parent.frame()
   ## Evaluate a in e1
   aa <- eval(expression(a),e1)
   ## evaluate the expression bound to a in e1
   a <- expression(x+y)
   list(aa = aa, eval = eval(a, e1))
}
tst.ev <- function(a = 7) { x <- pi; y <- 1; ev() }
tst.ev()#-> aa : 7,  eval : 4.14

a <- list(a=3, b=4)
with(a, a <- 5) # alters the copy of a from the list, discarded.

##
## Example of evalq()
##

N <- 3
env <- new.env()
assign("N", 27, envir=env)
## this version changes the visible copy of N only, since the argument
## passed to eval is '4'.
eval(N <- 4, env)
```

```
N
get("N", envir=env)
## this version does the assignment in env, and changes N only there.
evalq(N <- 5, env)
N
get("N", envir=env)

##
## Uses of local()
##

# Mutually recursive.
# gg gets value of last assignment, an anonymous version of f.

gg <- local({
    k <- function(y)f(y)
    f <- function(x) if(x) x*k(x-1) else 1
})
gg(10)
sapply(1:5, gg)

# Nesting locals. a is private storage accessible to k
gg <- local({
    k <- local({
        a <- 1
        function(y){print(a <<- a+1);f(y)}
    })
    f <- function(x) if(x) x*k(x-1) else 1
})
sapply(1:5, gg)

ls(envir=environment(gg))
ls(envir=environment(get("k", envir=environment(gg))))
```

`exists` *Is an Object Defined?*

Description

Look for an R object of the given name.

Usage

```
exists(x, where = -1, envir = , frame, mode = "any",
       inherits = TRUE)
```

Arguments

`x`	a variable name (given as a character string).
`where`	where to look for the object (see the details section); if omitted, the function will search as if the name of the object appeared unquoted in an expression.
`envir`	an alternative way to specify an environment to look in, but it is usually simpler to just use the `where` argument.

`frame`	a frame in the calling list. Equivalent to giving `where` as `sys.frame(frame)`.
`mode`	the mode or type of object sought: see the 'Details' section.
`inherits`	should the enclosing frames of the environment be searched?

Details

The `where` argument can specify the environment in which to look for the object in any of several ways: as an integer (the position in the `search` list); as the character string name of an element in the search list; or as an `environment` (including using `sys.frame` to access the currently active function calls). The `envir` argument is an alternative way to specify an environment, but is primarily there for back compatibility.

This function looks to see if the name `x` has a value bound to it in the specified environment. If `inherits` is `TRUE` and a value is not found for `x` in the specified environment, the enclosing frames of the environment are searched until the name `x` is encountered. See `environment` and the 'R Language Definition' manual for details about the structure of environments and their enclosures.

Warning: `inherits = TRUE` is the default behaviour for R but not for S.

If `mode` is specified then only objects of that type are sought. The `mode` may specify one of the collections `"numeric"` and `"function"` (see `mode`): any member of the collection will suffice. (This is true even if a member of a collection is specified, so for example `mode="special"` will seek any type of function.)

Value

Logical, true if and only if an object of the correct name and mode is found.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`get`.

Examples

```
##  Define a substitute function if necessary:
if(!exists("some.fun", mode="function"))
  some.fun <- function(x) { cat("some.fun(x)\n"); x }
search()
exists("ls", 2) # true even though ls is in pos=3
exists("ls", 2, inherits = FALSE) # false
```

`expand.grid` *Create a Data Frame from All Combinations of Factors*

Description

Create a data frame from all combinations of the supplied vectors or factors. See the description of the return value for precise details of the way this is done.

Usage

```
expand.grid(..., KEEP.OUT.ATTRS = TRUE)
```

Arguments

`...`	vectors, factors or a list containing these.
`KEEP.OUT.ATTRS`	a logical indicating the `"out.attrs"` attribute (see below) should be computed and returned.

Value

A data frame containing one row for each combination of the supplied factors. The first factors vary fastest. The columns are labelled by the factors if these are supplied as named arguments or named components of a list. The row names are 'automatic'.

Attribute `"out.attrs"` is a list which gives the dimension and dimnames for use by `predict` methods.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`combn` (package `utils`) for the generation of all combinations of n elements, taken m at a time.

Examples

```
require(utils)

expand.grid(height = seq(60, 80, 5), weight = seq(100, 300, 50),
            sex = c("Male","Female"))

x <- seq(0,10, length.out=100)
y <- seq(-1,1, length.out=20)
d1 <- expand.grid(x=x, y=y)
d2 <- expand.grid(x=x, y=y, KEEP.OUT.ATTRS = FALSE)
object.size(d1) - object.size(d2)
##-> 5992 or 8832 (on 32- / 64-bit platform)
```

`expression` *Unevaluated Expressions*

Description

Creates or tests for objects of mode `"expression"`.

Usage

```
expression(...)

is.expression(x)
as.expression(x, ...)
```

Arguments

`...`	`expression`: R objects, typically calls, symbols or constants. `as.expression`: arguments to be passed to methods.
`x`	an arbitrary R object.

Details

'Expression' here is not being used in its colloquial sense, that of mathematical expressions. Those are calls (see `call`) in R, and an R expression vector is a list of calls, symbols etc, typically as returned by `parse`.

As an object of mode `"expression"` is a list, it can be subsetted by both `[` and by `[[`, the latter extracting individual calls etc.

Value

`expression` returns a vector of type `"expression"` containing its arguments (unevaluated).

`is.expression` returns `TRUE` if `expr` is an expression object and `FALSE` otherwise.

`as.expression` attempts to coerce its argument into an expression object. It is generic, and only the default method is described here. `NULL`, calls, symbols (see `as.symbol`) and pairlists are returned as the element of a length-one expression vector. Vectors (including lists) are placed element-by-element into an expression vector. Other types are not currently supported.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`call`, `eval`, `function`. Further, `text` and `legend` for plotting mathematical expressions.

Examples

```
length(ex1 <- expression(1+ 0:9))# 1
ex1
eval(ex1)# 1:10

length(ex3 <- expression(u,v, 1+ 0:9))# 3
mode(ex3 [3]) # expression
mode(ex3[[3]])# call
rm(ex3)
```

`Extract` *Extract or Replace Parts of an Object*

Description

Operators acting on vectors, matrices, arrays and lists to extract or replace parts.

Usage

```
x[i]
x[i, j, ... , drop = TRUE]
x[[i, exact = TRUE]]
x[[i, j, ..., exact = TRUE]]
x$name
```

Arguments

`x`	object from which to extract element(s) or in which to replace element(s).
`i, j, ...`	indices specifying elements to extract or replace. Indices are `numeric` or `character` vectors or empty (missing) or `NULL`. Numeric values are coerced to integer as by `as.integer` (and hence truncated towards zero). Character vectors will be matched to the `names` of the object (or for matrices/arrays, the `dimnames`): see 'Character indices' below for further details. For `[`-indexing only: `i, j, ...` can be logical vectors, indicating elements/slices to select. Such vectors are recycled if necessary to match the corresponding extent. `i, j, ...` can also be negative integers, indicating elements/slices to leave out of the selection. When indexing arrays by `[` a single argument `i` can be a matrix with as many columns as there are dimensions of `x`; the result is then a vector with elements corresponding to the sets of indices in each row of `i`. An index value of `NULL` is treated as if it were `integer(0)`.
`name`	A literal character string or a name (possibly backtick quoted). For extraction, this is normally (see under 'Environments') partially matched to the `names` of the object.
`drop`	For matrices and arrays. If `TRUE` the result is coerced to the lowest possible dimension (see the examples). This only works for extracting elements, not for the replacement. See `drop` for further details.
`exact`	Controls possible partial matching of `[[` when extracting by a character vector (for most objects, but see under 'Environments'). The default is no partial matching. Value `NA` allows partial matching but issues a warning when it occurs. Value `FALSE` allows partial matching without any warning.

Details

These operators are generic. You can write methods to handle indexing of specific classes of objects, see InternalMethods as well as `[.data.frame` and `[.factor`. The descriptions here apply only to the default methods. Note that separate methods are required for the replacement functions `[<-`, `[[<-` and `$<-` for use when indexing occurs on the assignment side of an expression.

The most important distinction between `[`, `[[` and `$` is that the `[` can select more than one element whereas the other two select a single element.

The default methods work somewhat differently for atomic vectors, matrices/arrays and for recursive (list-like, see `is.recursive`) objects. `$` is only valid for recursive objects, and is only discussed in the section below on recursive objects. Its use on non-recursive objects was deprecated in R 2.5.0 and removed in R 2.7.0.

Subsetting (except by an empty index) will drop all attributes except `names`, `dim` and `dimnames`.

Indexing can occur on the right-hand-side of an expression for extraction, or on the left-hand-side for replacement. When an index expression appears on the left side of an assignment (known as *subassignment*) then that part of `x` is set to the value of the right hand side of the assignment. In this case no partial matching of character indices is done, and the left-hand-side is coerced as needed to accept the values. Attributes are preserved (although `names`, `dim` and `dimnames` will be adjusted suitably). Subassignment is done sequentially, so if an index is specified more than once the latest assigned value for an index will result.

It is an error to apply any of these operators to an object which is not subsettable (e.g. a function).

Atomic vectors

The usual form of indexing is `"["`. `"[["` can be used to select a single element, but `"["` can also do so.

The index object `i` can be numeric, logical, character or empty. Indexing by factors is allowed and is equivalent to indexing by the numeric codes (see `factor`) and not by the character values which are printed (for which use `[as.character(i)]`).

An empty index selects all values: this is most often used to replace all the entries but keep the `attributes`.

Matrices and arrays

Matrices and arrays are vectors with a dimension attribute and so all the vector forms of indexing can be used with a single index. The result will be an unnamed vector unless `x` is one-dimensional when it will be a one-dimensional array.

The most common form of indexing a k-dimensional array is to specify k indices to `[`. As for vector indexing, the indices can be numeric, logical, character, empty or even factor. An empty index (a comma separated blank) indicates that all entries in that dimension are selected. The argument `drop` applies to this form of indexing.

A third form of indexing is via a numeric matrix with the one column for each dimension: each row of the index matrix then selects a single element of the array, and the result is a vector. Negative indices are not allowed in the index matrix. `NA` and zero values are allowed: rows of an index matrix containing a zero are ignored, whereas rows containing an `NA` produce an `NA` in the result.

A vector obtained by matrix indexing will be unnamed unless `x` is one-dimensional when the row names (if any) will be indexed to provide names for the result.

Recursive (list-like) objects

Indexing by `[` is similar to atomic vectors and selects a list of the specified element(s).

Both `[[` and `$` select a single element of the list. The main difference is that `$` does not allow computed indices, whereas `[[` does. `x$name` is equivalent to `x[["name", exact = FALSE]]`. Also, the partial matching behavior of `[[` can be controlled using the `exact` argument.

`[` and `[[` are sometimes applied to other recursive objects such as calls and expressions. Pairlists are coerced to lists for extraction by `[`, but all three operators can be used for replacement.

`[[` can be applied recursively to lists, so that if the single index `i` is a vector of length `p`, `alist[[i]]` is equivalent to `alist[[i1]]...[[ip]]` providing all but the final indexing results in a list.

When either `[[` or `$` is used for replacement, a value of `NULL` deletes the corresponding item of the list.

When `$<-` is applied to a `NULL` `x`, it first coerces `x` to `list()`. This is what also happens with `[[<-` if the replacement value `value` is of length greater than one: if `value` has length 1 or 0, `x` is first coerced to a zero-length vector of the type of `value`.

Environments

Both `$` and `[[` can be applied to environments. Only character indices are allowed and no partial matching is done. The semantics of these operations are those of `get(i, env=x, inherits=FALSE)`. If no match is found then `NULL` is returned. The replacement versions, `$<-` and `[[<-`, can also be used. Again, only character arguments are allowed. The semantics in this case are those of `assign(i, value, env=x, inherits=FALSE)`. Such an assignment will either create a new binding or change the existing binding in `x`.

NAs in indexing

When extracting, a numerical, logical or character `NA` index picks an unknown element and so returns `NA` in the corresponding element of a logical, integer, numeric, complex or character result, and `NULL` for a list. (It returns `00` for a raw result.]

When replacing (that is using indexing on the lhs of an assignment) `NA` does not select any element to be replaced. As there is ambiguity as to whether an element of the rhs should be used or not, this is only allowed if the rhs value is of length one (so the two interpretations would have the same outcome).

Argument matching

Note that these operations do not match their index arguments in the standard way: argument names are ignored and positional matching only is used. So `m[j=2,i=1]` is equivalent to `m[2,1]` and **not** to `m[1,2]`.

This may not be true for methods defined for them; for example it is not true for the `data.frame` methods described in `[.data.frame`.

To avoid confusion, do not name index arguments (but `drop` and `exact` must be named).

S4 methods

These operators are also S4 generic, but as primitives, S4 methods will be dispatched only on S4 objects.

S4 methods for `$` will be passed `name` as a character vector: despite the message given by `getGeneric("$")` you cannot usefully write methods based on the class of `name`.

Character indices

Character indices can in some circumstances be partially matched (see `pmatch`) to the names or dimnames of the object being subsetted (but never for subassignment). Unlike S (Becker *et al* p. 358)), R has never used partial matching when extracting by `[`. As from R 2.7.0, partial matching is not by default used by `[[` (see argument `exact`).

Thus the default behaviour is to use partial matching only when extracting from recursive objects (except environments) by `$`. Even in that case, warnings can be switched on by `options(warnPartialMatchAttr = TRUE)`.

Neither empty (`""`) nor `NA` indices match any names, not even empty nor missing names. If any object has no names or appropriate dimnames, they are taken as all `""` and so match nothing.

Note

The documented behaviour of S is that an `NA` replacement index 'goes nowhere' but uses up an element of `value` (Becker *et al* p. 359). However, that has not been true of other implementations.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`names` for details of matching to names, and `pmatch` for partial matching.

`list`, `array`, `matrix`.

`[.data.frame` and `[.factor` for the behaviour when applied to data.frame and factors.

`Syntax` for operator precedence, and the *R Language* reference manual about indexing details.

Examples

```
x <- 1:12; m <- matrix(1:6, nrow=2); li <- list(pi=pi, e = exp(1))
x[10]                  # the tenth element of x
x <- x[-1]             # delete the 1st element of x
m[1,]                  # the first row of matrix m
m[1, , drop = FALSE]   # is a 1-row matrix
m[,c(TRUE,FALSE,TRUE)]# logical indexing
m[cbind(c(1,2,1),3:1)]# matrix index
m <- m[,-1]            # delete the first column of m
li[[1]]                # the first element of list li
y <- list(1,2,a=4,5)
y[c(3,4)]              # a list containing elements 3 and 4 of y
y$a                    # the element of y named a

## non-integer indices are truncated:
(i <- 3.999999999) # "4" is printed
(1:5)[i]  # 3

## recursive indexing into lists
z <- list( a=list( b=9, c='hello'), d=1:5)
unlist(z)
z[[c(1, 2)]]
z[[c(1, 2, 1)]]  # both "hello"
z[[c("a", "b")]] <- "new"
```

```
unlist(z)

## check $ and [[ for environments
e1 <- new.env()
e1$a <- 10
e1[["a"]]
e1[["b"]] <- 20
e1$b
ls(e1)
```

`Extract.data.frame` *Extract or Replace Parts of a Data Frame*

Description

Extract or replace subsets of data frames.

Usage

```
## S3 method for class 'data.frame':
x[i, j, drop = ]
## S3 replacement method for class 'data.frame':
x[i, j] <- value
## S3 method for class 'data.frame':
x[[..., exact = TRUE]]
## S3 replacement method for class 'data.frame':
x[[i, j]] <- value
## S3 replacement method for class 'data.frame':
x$i <- value
```

Arguments

`x`	data frame.
`i, j, ...`	elements to extract or replace. For `[` and `[[`, these are `numeric` or `character` or, for `[` only, empty. Numeric values are coerced to integer as if by `as.integer`. For replacement by `[`, a logical matrix is allowed. For replacement by `$`, `i` is a name or literal character string.
`drop`	logical. If `TRUE` the result is coerced to the lowest possible dimension. The default is to drop if only one column is left, but **not** to drop if only one row is left.
`value`	A suitable replacement value: it will be repeated a whole number of times if necessary and it may be coerced: see the Coercion section. If `NULL`, deletes the column if a single column is selected.
`exact`	logical: see `[`, and applies to column names.

Details

Data frames can be indexed in several modes. When `[` and `[[` are used with a single index (`x[i]` or `x[[i]]`), they index the data frame as if it were a list. In this usage a `drop` argument is ignored, with a warning. Using `$` is equivalent to using `[[i, exact = FALSE]]`.

When `[` and `[[` are used with two indices (`x[i, j]` and `x[[i, j]]`) they act like indexing a matrix: `[[` can only be used to select one element.

If `[` returns a data frame it will have unique (and non-missing) row names, if necessary transforming the row names using `make.unique`. Similarly, column names will be transformed to be unique if necessary (e.g. if columns are selected more than once, or if more than one column of a given name is selected if the data frame has duplicate columns).

When `drop = TRUE`, this is applied to the subsetting of any matrices contained in the data frame as well as to the data frame itself.

The replacement methods can be used to add whole column(s) by specifying non-existent column(s), in which case the column(s) are added at the right-hand edge of the data frame and numerical indices must be contiguous to existing indices. On the other hand, rows can be added at any row after the current last row, and the columns will be in-filled with missing values. Missing values in the indices are not allowed for replacement.

For `[` the replacement value can be a list: each element of the list is used to replace (part of) one column, recycling the list as necessary. If columns specified by number are created, the names (if any) of the corresponding list elements are used to name the columns. If the replacement is not selecting rows, list values can contain `NULL` elements which will cause the corresponding columns to be deleted. (See the Examples.)

Matrix indexing using `[` is not recommended, and barely supported. For extraction, `x` is first coerced to a matrix. For replacement a logical matrix (only) can be used to select the elements to be replaced in the same way as for a matrix.

Both `[` and `[[` extraction methods partially match row names. By default neither partially match column names, but `[[` will unless `exact=TRUE`. If you want to do exact matching on row names use `match` as in the examples.

Value

For `[` a data frame, list or a single column (the latter two only when dimensions have been dropped). If matrix indexing is used for extraction a matrix results. If the result would be a data frame an error results if undefined columns are selected (as there is no general concept of a 'missing' column in a data frame). Otherwise if a single column is selected and this is undefined the result is `NULL`.

For `[[` a column of the data frame or `NULL` (extraction with one index) or a length-one vector (extraction with two indices).

For `$`, a column of the data frame (or `NULL`).

For `[<-`, `[[<-` and `$<-`, a data frame.

Coercion

The story over when replacement values are coerced is a complicated one, and one that has changed during R's development. This section is a guide only.

When `[` and `[[` are used to add or replace a whole column, no coercion takes place but `value` will be replicated (by calling the generic function `rep`) to the right length if an exact number of repeats can be used.

When `[` is used with a logical matrix, each value is coerced to the type of the column into which it is to be placed.

When `[` and `[[` are used with two indices, the column will be coerced as necessary to accommodate the value.

Note that when the replacement value is an array (including a matrix) it is *not* treated as a series of columns (as `data.frame` and `as.data.frame` do) but inserted as a single column.

Warning

The default behaviour when only one *row* is left is equivalent to specifying `drop = FALSE`. To drop from a data frame to a list, `drop = TRUE` has to be specified explicitly.

See Also

`subset` which is often easier for extraction, `data.frame`, `Extract`.

Examples

```
sw <- swiss[1:5, 1:4]  # select a manageable subset

sw[1:3]      # select columns
sw[, 1:3]    # same
sw[4:5, 1:3] # select rows and columns
sw[1]        # a one-column data frame
sw[, 1, drop = FALSE]  # the same
sw[, 1]      # a (unnamed) vector
sw[[1]]      # the same

sw[1,]       # a one-row data frame
sw[1,, drop=TRUE]  # a list

sw["C", ] # partially matches
sw[match("C", row.names(sw)), ] # no exact match


swiss[ c(1, 1:2), ]   # duplicate row, unique row names are created

sw[sw <= 6] <- 6  # logical matrix indexing
sw

## adding a column
sw["new1"] <- LETTERS[1:5]   # adds a character column
sw[["new2"]] <- letters[1:5] # ditto
sw[, "new3"] <- LETTERS[1:5] # ditto
sw$new4 <- 1:5
sapply(sw, class)
sw$new4 <- NULL              # delete the column
sw
sw[6:8] <- list(letters[10:14], NULL, aa=1:5)
# delete col7, update 6, append
sw

## matrices in a data frame
A <- data.frame(x=1:3, y=I(matrix(4:6)), z=I(matrix(letters[1:9],3,3)))
A[1:3, "y"] # a matrix
A[1:3, "z"] # a matrix
A[, "y"]    # a matrix
```

`Extract.factor` *Extract or Replace Parts of a Factor*

Description

Extract or replace subsets of factors.

Usage

```
## S3 method for class 'factor':
x[..., drop = FALSE]
## S3 method for class 'factor':
x[[...]]
## S3 replacement method for class 'factor':
x[...] <- value
```

Arguments

`x`	a factor
`...`	a specification of indices – see `Extract`.
`drop`	logical. If true, unused levels are dropped.
`value`	character: a set of levels. Factor values are coerced to character.

Details

When unused levels are dropped the ordering of the remaining levels is preserved.

If `value` is not in `levels(x)`, a missing value is assigned with a warning.

Any `contrasts` assigned to the factor are preserved unless `drop=TRUE`.

The `[[` method supports argument `exact`.

Value

A factor with the same set of levels as `x` unless `drop=TRUE`.

See Also

`factor`, `Extract`.

Examples

```
## following example(factor)
(ff <- factor(substring("statistics", 1:10, 1:10), levels=letters))
ff[, drop=TRUE]
factor(letters[7:10])[2:3, drop = TRUE]
```

`Extremes` *Maxima and Minima*

Description

Returns the (parallel) maxima and minima of the input values.

Usage

```
max(..., na.rm = FALSE)
min(..., na.rm = FALSE)

pmax(..., na.rm = FALSE)
pmin(..., na.rm = FALSE)

pmax.int(..., na.rm = FALSE)
pmin.int(..., na.rm = FALSE)
```

Arguments

`...`	numeric or character arguments (see Note).
`na.rm`	a logical indicating whether missing values should be removed.

Details

`max` and `min` return the maximum or minimum of *all* the values present in their arguments, as `integer` if all are `logical` or `integer`, as `double` if all are numeric, and character otherwise.

If `na.rm` is `FALSE` an `NA` value in any of the arguments will cause a value of `NA` to be returned, otherwise `NA` values are ignored.

The minimum and maximum of a numeric empty set are `+Inf` and `-Inf` (in this order!) which ensures *transitivity*, e.g., `min(x1, min(x2)) == min(x1, x2)`. For numeric `x` `max(x) == -Inf` and `min(x) == +Inf` whenever `length(x) == 0` (after removing missing values if requested). However, `pmax` and `pmin` return `NA` if all the parallel elements are `NA` even for `na.rm = TRUE`.

`pmax` and `pmin` take one or more vectors (or matrices) as arguments and return a single vector giving the 'parallel' maxima (or minima) of the vectors. The first element of the result is the maximum (minimum) of the first elements of all the arguments, the second element of the result is the maximum (minimum) of the second elements of all the arguments and so on. Shorter inputs are recycled if necessary. `attributes` (such as `names` or `dim`) are transferred from the first argument (if applicable).

`pmax.int` and `pmin.int` are faster internal versions only used when all arguments are atomic vectors and there are no classes: they drop all attributes. (Note that all versions fail for raw and complex vectors since these have no ordered.)

`max` and `min` are generic functions: methods can be defined for them individually or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

By definition the min/max of any vector containing an `NaN` is `NaN`, except that the min/max of any vector containing an `NA` is `NA` even if it also contains an `NaN`. Note that `max(NA, Inf) == NA` even though the maximum would be `Inf` whatever the missing value actually is.

The max/min of an empty character vector is a character `NA`. (One could argue that as `""` is the smallest character element, the maximum should be `""`, but there is no obvious candidate for the minimum.)

Value

For `min` or `max`, a length-one vector. For `pmin` or `pmax`, a vector of length the longest of the input vectors.

The type of the result will be that of the highest of the inputs in the hierarchy integer < real < character.

For `min` and `max` if there are only numeric inputs and all are empty (after possible removal of `NA`s), the result is double (`Inf` or `-Inf`).

S4 methods

`max` and `min` are part of the S4 `Summary` group generic. Methods for them must use the signature `x, ..., na.rm`.

Note

'Numeric' arguments are vectors of type integer and numeric, and logical (coerced to integer). For historical reasons, `NULL` is accepted as equivalent to `integer(0)`.

`pmax` and `pmin` will also work on classed objects with appropriate methods for comparison, `is.na` and `rep` (if recycling of arguments is needed).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`range` (*both* min and max) and `which.min` (`which.max`) for the *arg min*, i.e., the location where an extreme value occurs.

'plotmath' for the use of `min` in plot annotation.

Examples

```
require(stats); require(graphics)
 min(5:1, pi) #-> one number
pmin(5:1, pi) #->  5  numbers

x <- sort(rnorm(100));  cH <- 1.35
pmin(cH, quantile(x)) # no names
pmin(quantile(x), cH) # has names
plot(x, pmin(cH, pmax(-cH, x)), type='b', main= "Huber's function")
```

`factor` *Factors*

Description

The function `factor` is used to encode a vector as a factor (the terms 'category' and 'enumerated type' are also used for factors). If `ordered` is `TRUE`, the factor levels are assumed to be ordered. For compatibility with S there is also a function `ordered`.

`is.factor`, `is.ordered`, `as.factor` and `as.ordered` are the membership and coercion functions for these classes.

Usage

```
factor(x = character(),
       levels = sort(unique.default(x), na.last = TRUE),
       labels = levels, exclude = NA, ordered = is.ordered(x))

ordered(x, ...)

is.factor(x)
is.ordered(x)

as.factor(x)
as.ordered(x)

addNA(x, ifany=FALSE)
```

Arguments

`x`	a vector of data, usually taking a small number of distinct values.
`levels`	an optional vector of the values that `x` might have taken. The default is the set of values taken by `x`, sorted into increasing order.
`labels`	*either* an optional vector of labels for the levels (in the same order as `levels` after removing those in `exclude`), *or* a character string of length 1.
`exclude`	a vector of values to be excluded when forming the set of levels. This should be of the same type as `x`, and will be coerced if necessary.
`ordered`	logical flag to determine if the levels should be regarded as ordered (in the order given).
`...`	(in `ordered(.)`): any of the above, apart from `ordered` itself.
`ifany`	(in `addNA`): Only add an `NA` level if it is used, i.e. if `any(is.na(x))`.

Details

The type of the vector `x` is not restricted.

Ordered factors differ from factors only in their class, but methods and the model-fitting functions treat the two classes quite differently.

The encoding of the vector happens as follows. First all the values in `exclude` are removed from `levels`. If `x[i]` equals `levels[j]`, then the `i`-th element of the result is `j`. If no match is found for `x[i]` in `levels`, then the `i`-th element of the result is set to `NA`.

Normally the 'levels' used as an attribute of the result are the reduced set of levels after removing those in `exclude`, but this can be altered by supplying `labels`. This should either be a set of new labels for the levels, or a character string, in which case the levels are that character string with a sequence number appended.

`factor(x, exclude=NULL)` applied to a factor is a no-operation unless there are unused levels: in that case, a factor with the reduced level set is returned. If `exclude` is used it should also be a factor with the same level set as `x` or a set of codes for the levels to be excluded.

The codes of a factor may contain `NA`. For a numeric `x`, set `exclude=NULL` to make `NA` an extra level (prints as `<NA>`); by default, this is the last level.

If `NA` is a level, the way to set a code to be missing (as opposed to the code of the missing level) is to use `is.na` on the left-hand-side of an assignment (as in `is.na(f)[i] <- TRUE`; indexing inside `is.na` does not work). Under those circumstances missing values are currently printed as `<NA>`, i.e., identical to entries of level `NA`.

`is.factor` is generic: you can write methods to handle specific classes of objects, see Internal-Methods.

Value

`factor` returns an object of class `"factor"` which has a set of integer codes the length of `x` with a `"levels"` attribute of mode `character`. If `ordered` is true (or `ordered` is used) the result has class `c("ordered", "factor")`.

Applying `factor` to an ordered or unordered factor returns a factor (of the same type) with just the levels which occur: see also `[.factor` for a more transparent way to achieve this.

`is.factor` returns `TRUE` or `FALSE` depending on whether its argument is of type factor or not. Correspondingly, `is.ordered` returns `TRUE` when its argument is ordered and `FALSE` otherwise.

`as.factor` coerces its argument to a factor. It is an abbreviated form of `factor`.

`as.ordered(x)` returns `x` if this is ordered, and `ordered(x)` otherwise.

`addNA` modifies a factor by turning `NA` into an extra level (so that `NA` values are counted in tables, for instance).

Warning

The interpretation of a factor depends on both the codes and the `"levels"` attribute. Be careful only to compare factors with the same set of levels (in the same order). In particular, `as.numeric` applied to a factor is meaningless, and may happen by implicit coercion. To transform a factor `f` to its original numeric values, `as.numeric(levels(f))[f]` is recommended and slightly more efficient than `as.numeric(as.character(f))`.

The levels of a factor are by default sorted, but the sort order may well depend on the locale at the time of creation, and should not be assumed to be ASCII.

There are some anomalies associated with factors that have `NA` as a level. It is suggested to use them sparingly, e.g., only for tabulation purposes.

Comparison operators and group generic methods

There are `"factor"` and `"ordered"` methods for the group generic `Ops`, which provide methods for the Comparison operators. (The rest of the group and the `Math` and `Summary` groups generate an error as they are not meaningful for factors.)

Only == and != can be used for factors: a factor can only be compared to another factor with an identical set of levels (not necessarily in the same ordering) or to a character vector. Ordered factors

are compared in the same way, but the general dispatch mechanism precludes comparing ordered and unordered factors.

All the comparison operators are available for ordered factors. Sorting is done by the levels of the operands: if both operands are ordered factors they must have the same level set.

Note

Storing character data as a factor is more efficient storage if there is even a small proportion of repeats. On a 32-bit machine storing a string of n bytes takes $28 + 8\lceil (n+1)/8 \rceil$ bytes whereas storing a factor code takes 4 bytes. (On a 64-bit machine 28 is replaced by 56 or more.) Only if they were computed from the same values (or in some cases read from a file: see `scan`) will identical strings share storage.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`[.factor` for subsetting of factors.

`gl` for construction of balanced factors and `C` for factors with specified contrasts. `levels` and `nlevels` for accessing the levels, and `unclass` to get integer codes.

Examples

```
(ff <- factor(substring("statistics", 1:10, 1:10), levels=letters))
as.integer(ff)  # the internal codes
factor(ff)      # drops the levels that do not occur
ff[, drop=TRUE] # the same, more transparently

factor(letters[1:20], labels="letter")

class(ordered(4:1)) # "ordered", inheriting from "factor"

## suppose you want "NA" as a level, and to allow missing values.
(x <- factor(c(1, 2, NA), exclude = NULL))
is.na(x)[2] <- TRUE
x  # [1] 1    <NA> <NA>
is.na(x)
# [1] FALSE  TRUE FALSE

## Using addNA()
Month <- airquality$Month
table(addNA(Month))
table(addNA(Month, ifany=TRUE))
```

---

file.access *Ascertain File Accessibility*

Description

Utility function to access information about files on the user's file systems.

Usage

```
file.access(names, mode = 0)
```

Arguments

names	character vector containing file names.
mode	integer specifying access mode required.

Details

Tilde-expansion is done on names: see path.expand.

The mode value can be the exclusive or of the following values

0 test for existence.

1 test for execute permission.

2 test for write permission.

4 test for read permission.

Permission will be computed for real user ID and real group ID (rather than the effective IDs).

Please note that it is not good to use this function to test before trying to open a file. On a multi-tasking system, it is possible that the accessibility of a file will change between the time you call file.access() and the time you try to open the file. It is better to wrap file open attempts in try.

Value

An integer vector with values 0 for success and −1 for failure.

Note

This is intended as a replacement for the S-PLUS function access, a wrapper for the C function of the same name, which explains the return value encoding. Note that the return value is **false** for **success**.

See Also

file.info, try

Examples

```
fa <- file.access(dir("."))
table(fa) # count successes & failures
```

file.choose *Choose a File Interactively*

Description

Choose a file interactively.

Usage

```
file.choose(new = FALSE)
```

Arguments

new	Logical: choose the style of dialog box presented to the user: at present only new = FALSE is used.

Value

A character vector of length one giving the file path.

See Also

list.files for non-interactive selection.

file.info *Extract File Information*

Description

Utility function to extract information about files on the user's file systems.

Usage

```
file.info(...)
```

Arguments

...	character vectors containing file paths.

Details

The file paths are tilde-expanded: see path.expand.

What is meant by 'file access' and hence the last access time is system-dependent.

The file 'mode' follows POSIX conventions, giving three octal digits summarizing the permissions for the file owner, the owner's group and for anyone respectively. Each digit is the logical *or* of read (4), write (2) and execute/search (1) permissions.

On most systems symbolic links are followed, so information is given about the file to which the link points rather than about the link.

Value

A data frame with row names the file names and columns

`size`	double: File size in bytes.
`isdir`	logical: Is the file a directory?
`mode`	integer of class `"octmode"`. The file permissions, printed in octal, for example `644`.
`mtime, ctime, atime`	integer of class `"POSIXct"`: file modification, 'last status change' and last access times.
`uid`	integer: the user ID of the file's owner.
`gid`	integer: the group ID of the file's group.
`uname`	character: `uid` interpreted as a user name.
`grname`	character: `gid` interpreted as a group name.

Unknown user and group names will be `NA`.

Entries for non-existent or non-readable files will be `NA`. The `uid`, `gid`, `uname` and `grname` columns may not be supplied on a non-POSIX Unix system.

What is meant by the three file times depends on the OS and file system. On Windows `ctime` is the creation time.

Note

Some (broken) systems allow files of more than 2Gb to be created but not accessed by the `stat` system call. Such files will show up as non-readable (and very likely not be readable by any of R's input functions).

See Also

`files`, `file.access`, `list.files`, and `DateTimeClasses` for the date formats.

Examples

```
ncol(finf <- file.info(dir()))# at least six
## Not run: finf # the whole list
## Those that are more than 100 days old :
finf[difftime(Sys.time(), finf[,"mtime"], units="days") > 100 , 1:4]

file.info("no-such-file-exists")
```

`file.path` *Construct Path to File*

Description

Construct the path to a file from components in a platform-independent way.

Usage

```
file.path(..., fsep = .Platform$file.sep)
```

Arguments

`...`	character vectors.
`fsep`	the path separator to use.

Value

A character vector of the arguments concatenated term-by-term and separated by `fsep` if all arguments have positive length; otherwise, an empty character vector.

`file.show` *Display One or More Files*

Description

Display one or more files.

Usage

```
file.show(..., header = rep("", nfiles),
          title = "R Information",
          delete.file = FALSE, pager = getOption("pager"),
          encoding = "")
```

Arguments

`...`	one or more character vectors containing the names of the files to be displayed. These will be tilde-expanded: see `path.expand`.
`header`	character vector (of the same length as the number of files specified in `...`) giving a header for each file being displayed. Defaults to empty strings.
`title`	an overall title for the display. If a single separate window is used for the display, `title` will be used as the window title. If multiple windows are used, their titles should combine the title and the file-specific header.
`delete.file`	should the files be deleted after display? Used for temporary files.
`pager`	the pager to be used.
`encoding`	character string giving the encoding to be assumed for the file(s).

Details

This function provides the core of the R help system, but it can be used for other purposes as well, such as `page`.

How the pager is implemented is highly system-dependent.

The basic Unix version concatenates the files (using the headers) to a temporary file, and displays it in the pager selected by the `pager` argument, which is a character vector specifying a system command (usually `less`) to run on the set of files.

Most GUI systems will use a separate pager window for each file, and let the user leave it up while R continues running. The selection of such pagers could either be done using special pager names being intercepted by lower-level code (such as `"internal"` and `"console"` on Windows), or by letting `pager` be an R function which will be called with the same first four arguments as `file.show` and take care of interfacing to the GUI.

Not all implementations will honour `delete.file`.

Author(s)

Ross Ihaka, Brian Ripley.

See Also

`files`, `list.files`, `help`.

Examples

```
file.show(file.path(R.home("doc"), "COPYRIGHTS"))
```

---

files *File and Directory Manipulation*

Description

These functions provide a low-level interface to the computer's file system.

Usage

```
file.create(..., showWarnings = TRUE)
file.exists(...)
file.remove(...)
file.rename(from, to)
file.append(file1, file2)
file.copy(from, to, overwrite = FALSE)
file.symlink(from, to)
dir.create(path, showWarnings = TRUE, recursive = FALSE,
           mode = "0777")
Sys.chmod(paths, mode = "0777")
Sys.umask(mode = "0000")
```

Arguments

`..., file1, file2, from, to, paths`	character vectors, containing file names or paths.
`path`	a character vector containing a single path name.
`overwrite`	logical; should the destination files be overwritten?
`showWarnings`	logical; should the warnings on failure be shown?
`recursive`	logical: should elements of the path other than the last be created? If true, like Unix's `mkdir -p`.
`mode`	the file mode to be used on Unix-alikes: it will be coerced by `as.octmode`.

Details

The ... arguments are concatenated to form one character string: you can specify the files separately or as one vector. All of these functions expand path names: see path.expand.

file.create creates files with the given names if they do not already exist and truncates them if they do. They are created with the maximal permissions allowed by the umask setting.

file.exists returns a logical vector indicating whether the files named by its argument exist. (Here 'exists' is in the sense of the system's stat call: a file will be reported as existing only if you have the permissions needed by stat. Existence can also be checked by file.access, which might use different permissions and so obtain a different result. Note that the existence of a file does not imply that it is readable: for that use file.access.)

file.remove attempts to remove the files named in its argument. On most platforms 'file' includes *empty* directories, symbolic links, fifos and sockets.

file.rename attempts to rename a single file.

file.append attempts to append the files named by its second argument to those named by its first. The R subscript recycling rule is used to align names given in vectors of different lengths.

file.copy works in a similar way to file.append but with the arguments in the natural order for copying. Copying to existing destination files is skipped unless overwrite = TRUE. The to argument can specify a single existing directory.

file.symlink makes symbolic links on those Unix-like platforms which support them. The to argument can specify a single existing directory.

dir.create creates the last element of the path, unless recursive = TRUE. Trailing path separators are removed. The mode will be modified by the umask setting in the same way as for the system function mkdir. What modes can be set is OS-dependent, and it is unsafe to assume that more than three octal digits will be used. For more details see your OS's documentation on the system call mkdir (and not that on the command-line utility of that name).

Sys.chmod sets the file permissions of one or more files. It may not be supported (when a warning is issued). See the comments for dir.create for how modes are interpreted. Changing mode on a symbolic link is unlikely to work (nor be necessary). For more details see your OS's documentation on the system call chmod (and not that on the command-line utility of that name).

Sys.umask sets the umask. It may not be supported (when a warning is issued and "0000" returned). For more details see your OS's documentation on the system call umask.

Value

dir.create and file.rename return a logical, true for success.

Sys.umask returns the previous value of the umask, invisibly, as a length-one object of class "octmode".

The remaining functions return a logical vector indicating which operation succeeded for each of the files attempted.

dir.create will return failure if the directory already exists.

If showWarnings = TRUE, file.create and dir.create will give a warning for an unexpected failure (e.g. not for a missing value nor for an already existing component for dir.create(recursive = TRUE)).

Using a missing value for a file or path name will always be regarded as a failure.

Author(s)

Ross Ihaka, Brian Ripley

See Also

file.info, file.access, file.path, file.show, list.files, unlink, basename, path.expand.

file_test.

Examples

```
cat("file A\n", file="A")
cat("file B\n", file="B")
file.append("A", "B")
file.create("A")
file.append("A", rep("B", 10))
if(interactive()) file.show("A")
file.copy("A", "C")
dir.create("tmp")
file.copy(c("A", "B"), "tmp")
list.files("tmp")
setwd("tmp")
file.remove("B")
file.symlink(file.path("..", c("A", "B")), ".")
setwd("..")
unlink("tmp", recursive=TRUE)
file.remove("A", "B", "C")
```

---

findInterval *Find Interval Numbers or Indices*

Description

Find the indices of x in vec, where vec must be sorted (non-decreasingly); i.e., if i <- findInterval(x,v), we have $v_{i_j} \le x_j < v_{i_j+1}$ where $v_0 := -\infty$, $v_{N+1} := +\infty$, and N <- length(vec). At the two boundaries, the returned index may differ by 1, depending on the optional arguments rightmost.closed and all.inside.

Usage

```
findInterval(x, vec, rightmost.closed = FALSE, all.inside = FALSE)
```

Arguments

x	numeric.
vec	numeric, sorted (weakly) increasingly, of length N, say.
rightmost.closed	logical; if true, the rightmost interval, vec[N-1] .. vec[N] is treated as *closed*, see below.
all.inside	logical; if true, the returned indices are coerced into $\{1, \dots, N-1\}$, i.e., 0 is mapped to 1 and N to $N-1$.

Details

The function `findInterval` finds the index of one vector `x` in another, `vec`, where the latter must be non-decreasing. Where this is trivial, equivalent to `apply( outer(x, vec, ">="), 1, sum)`, as a matter of fact, the internal algorithm uses interval search ensuring $O(n \log N)$ complexity where `n <- length(x)` (and `N <- length(vec)`). For (almost) sorted `x`, it will be even faster, basically $O(n)$.

This is the same computation as for the empirical distribution function, and indeed, `findInterval(t, sort(X))` is *identical* to $nF_n(t; X_1, \ldots, X_n)$ where F_n is the empirical distribution function of $X_1, \ldots, X_n$.

When `rightmost.closed = TRUE`, the result for `x[j] = vec[N]` ($= \max(vec)$), is `N - 1` as for all other values in the last interval.

Value

vector of length `length(x)` with values in `0:N` (and `NA`) where `N <- length(vec)`, or values coerced to `1:(N-1)` if and only if `all.inside = TRUE` (equivalently coercing all x values *inside* the intervals). Note that `NA`s are propagated from `x`, and `Inf` values are allowed in both `x` and `vec`.

Author(s)

Martin Maechler

See Also

`approx(*, method = "constant")` which is a generalization of `findInterval()`, `ecdf` for computing the empirical distribution function which is (up to a factor of n) also basically the same as findInterval(.).

Examples

```
N <- 100
X <- sort(round(stats::rt(N, df=2), 2))
tt <- c(-100, seq(-2,2, len=201), +100)
it <- findInterval(tt, X)
tt[it < 1 | it >= N] # only first and last are outside range(X)
```

---

force *Force Evaluation of an Argument*

Description

Forces the evaluation of a function argument.

Usage

```
force(x)
```

Arguments

`x`	a formal argument of the enclosing function.

Details

`force` forces the evaluation of a formal argument. This can be useful if the argument will be captured in a closure by the lexical scoping rules and will later be altered by an explicit assignment or an implicit assignment in a loop or an apply function.

Note

This is semantic sugar: just evaluating the symbol will do the same thing (see the examples).

`force` does not force the evaluation of other promises. (It works by forcing the promise that is created when the actual arguments of a call are matched to the formal arguments of a closure, the mechanism which implements *lazy evaluation*.)

Examples

```
f <- function(y) function() y
lf <- vector("list", 5)
for (i in seq(along = lf)) lf[[i]] <- f(i)
lf[[1]]()  # returns 5

g <- function(y) { force(y); function() y }
lg <- vector("list", 5)
for (i in seq(along = lg)) lg[[i]] <- g(i)
lg[[1]]()  # returns 1

## This is identical to
g <- function(y) { y; function() y }
```

`Foreign` *Foreign Function Interface*

Description

Functions to make calls to compiled code that has been loaded into R.

Usage

```
        .C(name, ..., NAOK = FALSE, DUP = TRUE, PACKAGE, ENCODING)
 .Fortran(name, ..., NAOK = FALSE, DUP = TRUE, PACKAGE, ENCODING)
.External(name, ..., PACKAGE)
    .Call(name, ..., PACKAGE)

.External.graphics(name, ..., PACKAGE)
    .Call.graphics(name, ..., PACKAGE)
```

Arguments

`name`	a character string giving the name of a C function or Fortran subroutine, or an object of class `"NativeSymbolInfo"` or `"NativeSymbol"` referring to such a name.
`...`	arguments to be passed to the foreign function.

NAOK	if TRUE then any NA or NaN or Inf values in the arguments are passed on to the foreign function. If FALSE, the presence of NA or NaN or Inf values is regarded as an error.
DUP	if TRUE then arguments are duplicated before their address is passed to C or Fortran.
PACKAGE	if supplied, confine the search for the name to the DLL given by this argument (plus the conventional extension, '.so', '.sl', '.dll', ...). This is intended to add safety for packages, which can ensure by using this argument that no other package can override their external symbols. Use PACKAGE="base" for symbols linked in to R.
ENCODING	optional name for an encoding to be assumed for character vectors. See 'Details'.

Details

The functions .C and .Fortran can be used to make calls to compiled C and Fortran code.

.External and .External.graphics can be used to call compiled code that uses R objects in the same way as internal R functions.

.Call and .Call.graphics can be used to call compiled code which makes use of internal R objects. The arguments are passed to the C code as a sequence of R objects. It is included to provide compatibility with S version 4.

Specifying ENCODING overrides any declared encodings (see link{Encoding}) which are otherwise used to translate to the current locale before passing the strings to the compiled code.

For details about how to write code to use with .Call and .External, see the chapter on "System and foreign language interfaces" in the "Writing R Extensions" manual.

Value

The functions .C and .Fortran return a list similar to the ... list of arguments passed in, but reflecting any changes made by the C or Fortran code.

.External, .Call, .External.graphics, and .Call.graphics return an R object.

These calls are typically made in conjunction with dyn.load which links DLLs to R.

The .graphics versions of .Call and .External are used when calling code which makes low-level graphics calls. They take additional steps to ensure that the device driver display lists are updated correctly.

Argument types

The mapping of the types of R arguments to C or Fortran arguments in .C or .Fortran is

R	C	Fortran
integer	int *	integer
numeric	double *	double precision
– or –	float *	real
complex	Rcomplex *	double complex
logical	int *	integer
character	char **	[see below]
raw	unsigned char *	not allowed
list	SEXP *	not allowed
other	SEXP	not allowed

Numeric vectors in R will be passed as type `double *` to C (and as `double precision` to Fortran) unless (i) `.C` or `.Fortran` is used, (ii) `DUP` is true and (iii) the argument has attribute `Csingle` set to `TRUE` (use `as.single` or `single`). This mechanism is only intended to be used to facilitate the interfacing of existing C and Fortran code.

The C type `Rcomplex` is defined in 'Complex.h' as a `typedef struct {double r; double i;}`. Fortran type `double complex` is an extension to the Fortran standard, and the availability of a mapping of `complex` to Fortran may be compiler dependent.

Note: The C types corresponding to `integer` and `logical` are `int`, not `long` as in S. This difference matters on 64-bit platforms.

The first character string of a character vector is passed as a C character array to Fortran: that string may be usable as `character*255` if its true length is passed separately. Only up to 255 characters of the string are passed back. (How well this works, or even if it works at all, depends on the C and Fortran compilers and the platform.)

Missing (`NA`) string values are passed to `.C` as the string "NA". As the C `char` type can represent all possible bit patterns there appears to be no way to distinguish missing strings from the string `"NA"`. If this distinction is important use `.Call`.

Functions, expressions, environments and other language elements are passed as the internal R pointer type `SEXP`. This type is defined in 'Rinternals.h' or the arguments can be declared as generic pointers, `void *`. Lists are passed as C arrays of `SEXP` and can be declared as `void *` or `SEXP *`. Note that you cannot assign values to the elements of the list within the C routine. Assigning values to elements of the array corresponding to the list bypasses R's memory management/garbage collection and will cause problems. Essentially, the array corresponding to the list is read-only. If you need to return S objects created within the C routine, use the `.Call` interface.

R functions can be invoked using `call_S` or `call_R` and can be passed lists or the simple types as arguments.

Warning

`DUP=FALSE` is dangerous.

There are two dangers with using `DUP=FALSE`.

The first is that if you pass a local variable to `.C`/`.Fortran` with `DUP=FALSE`, your compiled code can alter the local variable and not just the copy in the return list. Worse, if you pass a local variable that is a formal parameter of the calling function, you may be able to change not only the local variable but the variable one level up. This will be very hard to trace.

The second is that lists are passed as a single R `SEXP` with `DUP=FALSE`, not as an array of `SEXP`. This means the accessor macros in 'Rinternals.h' are needed to get at the list elements and the lists cannot be passed to `call_S`/`call_R`. New code using R objects should be written using `.Call` or `.External`, so this is now only a minor issue.

In addition, character vectors and lists cannot be used with `DUP=FALSE`.

It is safe and useful to set `DUP=FALSE` if you do not change any of the variables that might be affected, e.g.,

```
.C("Cfunction", input=x, output=numeric(10)).
```

In this case the output variable did not exist before the call so it cannot cause trouble. If the input variable is not changed in the C code of `Cfunction` you are safe.

Neither `.Call` nor `.External` copy their arguments. You should treat arguments you receive through these interfaces as read-only.

Fortran symbol names

All compilers that can be used with R map symbol names to lower case, and so does `.Fortran`.

Symbol names containing underscores are not valid Fortran 77 (although they are valid in Fortran 9x). Many Fortran 77 compilers (including `g77`) will allow them but translate them in a different way to names not containing underscores. Such names will work with `.Fortran`, but portable code should not use Fortran names containing underscores.

Use `.Fortran` with care for compiled Fortran 9x code: it may not work if the Fortran 9x compiler used differs from the Fortran compiler used when configuring R, especially if the subroutine name is not lower-case or includes an underscore.

Header files for external code

Writing code for use with `.External` and `.Call` will need to use internal R structures. If possible use just those defined in 'Rinternals.h' and/or the macros in 'Rdefines.h', as other header files are not installed and are even more likely to be changed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`.C` and `.Fortran`.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (`.Call`.)

See Also

`dyn.load`.

`formals` *Access to and Manipulation of the Formal Arguments*

Description

Get or set the formal arguments of a function.

Usage

```
formals(fun = sys.function(sys.parent()))
formals(fun, envir = environment(fun)) <- value
```

Arguments

`fun`	a function object, or see 'Details'.
`envir`	environment in which the function should be defined.
`value`	a list (or pairlist) of R expressions.

Details

For the first form, `fun` can also be a character string naming the function to be manipulated, which is searched for from the parent environment. If it is not specified, the function calling `formals` is used.

Only *closures* have formals, not primitive functions.

Value

`formals` returns the formal argument list of the function specified, as a `pairlist`, or `NULL` for a non-function or primitive.

The replacement form sets the formals of a function to the list/pairlist on the right hand side, and (potentially) resets the environment of the function.

See Also

`args` for a human-readable version, `alist`, `body`, `function`.

Examples

```
require(stats); require(graphics)
length(formals(lm))      # the number of formal arguments
names(formals(boxplot))  # formal arguments names

f <- function(x) a+b
formals(f) <- alist(a=,b=3) # function(a,b=3)a+b
f(2) # result = 5
```

`format` *Encode in a Common Format*

Description

Format an R object for pretty printing.

Usage

```
format(x, ...)

## Default S3 method:
format(x, trim = FALSE, digits = NULL, nsmall = 0,
       justify = c("left", "right", "centre", "none"),
       width = NULL, na.encode = TRUE, scientific = NA,
       big.mark = "",   big.interval = 3,
     small.mark = "", small.interval = 5,
   decimal.mark = ".", zero.print = NULL, drop0trailing = FALSE, ...)

## S3 method for class 'data.frame':
format(x, ..., justify = "none")

## S3 method for class 'factor':
format(x, ...)

## S3 method for class 'AsIs':
format(x, width = 12, ...)
```

Arguments

x	any R object (conceptually); typically numeric.
trim	logical; if FALSE, logical, numeric and complex values are right-justified to a common width: if TRUE the leading blanks for justification are suppressed.
digits	how many significant digits are to be used for numeric and complex x. The default, NULL, uses getOption(digits). This is a suggestion: enough decimal places will be used so that the smallest (in magnitude) number has this many significant digits, and also to satisfy nsmall. (For the interpretation for complex numbers see signif.)
nsmall	the minimum number of digits to the right of the decimal point in formatting real/complex numbers in non-scientific formats. Allowed values are 0 <= nsmall <= 20.
justify	should a *character* vector be left-justified (the default), right-justified, centred or left alone.
width	default method: the *minimum* field width or NULL or 0 for no restriction. AsIs method: the *maximum* field width for non-character objects. NULL corresponds to the default 12.
na.encode	logical: should NA strings be encoded? Note this only applies to elements of character vectors, not to numerical or logical NAs, which are always encoded as "NA".
scientific	Either a logical specifying whether elements of a real or complex vector should be encoded in scientific format, or an integer penalty (see options("scipen"). Missing values correspond to the current default penalty.
...	further arguments passed to or from other methods.
big.mark, big.interval, small.mark, small.interval, decimal.mark, zero.print, dr	used for prettying (longish) decimal sequences, passed to prettyNum: that help page explains the details.

Details

format is a generic function. Apart from the methods described here there are methods for dates (see format.Date), date-times (see format.POSIXct)) and for other classes such as format.octmode and format.dist.

format.data.frame formats the data frame column by column, applying the appropriate method of format for each column. Methods for columns are often similar to as.character but offer more control. Matrix and data-frame columns will be converted to separate columns in the result, and character columns (normally all) will be given class "AsIs".

format.factor converts the factor to a character vector and then calls the default method (and so justify applies).

format.AsIs deals with columns of complicated objects that have been extracted from a data frame. Character objects are passed to the default method (and so width does not apply). Otherwise it calls toString to convert the object to character (if a vector or list, element by element) and then right-justifies the result.

Justification for character vectors (and objects converted to character vectors by their methods) is done on display width (see nchar), taking double-width characters and the rendering of special characters (as escape sequences, including escaping backslash: see print.default) into account. Character strings are padded with blanks to the display width of the widest. (If na.encode

= FALSE missing character strings are not included in the width computations and are not encoded.)

Numeric vectors are encoded with the minimum number of decimal places needed to display all the elements to at least the digit significant digits. However, if all the elements then have trailing zeroes, the number of decimal places is reduced until at least one element has a non-zero final digit; see also the argument documentation for big.*, small.* etc, above.

Raw vectors are converted to their 2-digit hexadecimal representation by as.character.

Value

An object of similar structure to x containing character representations of the elements of the first argument x in a common format, and in the current locale's encoding.

For character, numeric, complex or factor x, dims and dimnames are preserved on matrices/arrays and names on vectors: no other attributes are copied.

If x is a list, the result is a character vector obtained by applying format.default(x, ...) to each element of the list (after unlisting elements which are themselves lists), and then collapsing the result for each element with paste(collapse = ", "). The defaults in this case are trim = TRUE, justify = "none" since one does not usually want alignment in the collapsed strings.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

format.info indicates how an atomic vector would be formatted.

formatC, paste, as.character, sprintf, print, prettyNum, toString, encodeString.

Examples

```
format(1:10)
format(1:10, trim = TRUE)

zz <- data.frame("(row names)"= c("aaaaa", "b"), check.names=FALSE)
format(zz)
format(zz, justify = "left")

## use of nsmall
format(13.7)
format(13.7, nsmall = 3)
format(c(6.0, 13.1), digits = 2)
format(c(6.0, 13.1), digits = 2, nsmall = 1)

## use of scientific
format(2^31-1)
format(2^31-1, scientific = TRUE)

## a list
z <- list(a=letters[1:3], b=(-pi+0i)^((-2:2)/2), c=c(1,10,100,1000),
          d=c("a", "longer", "character", "string"))
```

```
format(z, digits = 2)
format(z, digits = 2, justify = "left", trim = FALSE)
```

`format.Date` *Date Conversion Functions to and from Character*

Description

Functions to convert between character representations and objects of class `"Date"` representing calendar dates.

Usage

```
as.Date(x, ...)
## S3 method for class 'character':
as.Date(x, format = "", ...)
## S3 method for class 'numeric':
as.Date(x, origin, ...)

## S3 method for class 'Date':
format(x, ...)

## S3 method for class 'Date':
as.character(x, ...)
```

Arguments

`x`	An object to be converted.
`format`	A character string. The default is `"%Y-%m-%d"`. For details see `strftime`.
`origin`	a Date object, or something which can be coerced by `as.Date(origin, ...)` to such an object.
`...`	Further arguments to be passed from or to other methods, including `format` for `as.character` and `as.Date` methods.

Details

The usual vector re-cycling rules are applied to `x` and `format` so the answer will be of length that of the longer of the vectors.

Locale-specific conversions to and from character strings are used where appropriate and available. This affects the names of the days and months.

The `as.Date` methods accept character strings, factors, logical `NA` and objects of classes `"POSIXlt"` and `"POSIXct"`. (The last are converted to days by ignoring the time after midnight in the representation of the time in UTC.) Also objects of class `"date"` (from package **date** or **survival**) and `"dates"` (from package **chron**). Character strings are processed as far as necessary for the format specified: any trailing characters are ignored.

`as.Date` will accept numeric data (the number of days since an epoch), but *only* if `origin` is supplied.

The `format` and `as.character` methods ignore any fractional part of the date.

Value

The `format` and `as.character` methods return a character vector representing the date. `NA` dates are returned as `NA_character_`.

The `as.Date` methods return an object of class `"Date"`.

Note

The default formats follow the rules of the ISO 8601 international standard which expresses a day as `"2001-02-03"`.

If the date string does not specify the date completely, the returned answer may be system-specific. The most common behaviour is to assume that a missing year, month or day is the current one. If it specifies a date incorrectly, reliable implementations will give an error and the date is reported as `NA`. Unfortunately some common implementations (such as 'glibc') are unreliable and guess at the intended meaning.

Years before 1CE (aka 1AD) will probably not be handled correctly.

References

International Organization for Standardization (2004, 1988, 1997, ...) *ISO 8601. Data elements and interchange formats – Information interchange – Representation of dates and times.* For links to versions available on-line see (at the time of writing) http://www.qsl.net/g1smd/isopdf.htm; for information on the current official version, see http://www.iso.org/iso/en/prods-services/popstds/datesandtime.html.

See Also

Date for details of the date class; `locales` to query or set a locale.

Your system's help pages on `strftime` and `strptime` to see how to specify their formats.

Examples

```
## locale-specific version of the date
format(Sys.Date(), "%a %b %d")

## read in date info in format 'ddmmmyyyy'
## This will give NA(s) in some locales; setting the C locale
## as in the commented lines will overcome this on most systems.
## lct <- Sys.getlocale("LC_TIME"); Sys.setlocale("LC_TIME", "C")
x <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
z <- as.Date(x, "%d%b%Y")
## Sys.setlocale("LC_TIME", lct)
z

## read in date/time info in format 'm/d/y'
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
as.Date(dates, "%m/%d/%y")

## date given as number of days since 1900-01-01 (a date in 1989)
as.Date(32768, origin="1900-01-01")
```

`format.info` *format(.) Information*

Description

Information is returned on how `format(x, digits, nsmall)` would be formatted.

Usage

```
format.info(x, digits = NULL, nsmall = 0)
```

Arguments

`x`	an atomic vector; a potential argument of `format(x, ...)`.
`digits`	how many significant digits are to be used for numeric and complex `x`. The default, `NULL`, uses `getOption(digits)`.
`nsmall`	(see `format(..., nsmall)`).

Value

An `integer vector` of length 1, 3 or 6, say `r`.

For logical, integer and character vectors a single element, the width which would be used by `format` if `width = NULL`.

For numeric vectors:

`r[1]`	width (in characters) used by `format(x)`
`r[2]`	number of digits after decimal point.
`r[3]`	in `0:2`; if ≥ 1, *exponential* representation would be used, with exponent length of `r[3]+1`.

For a complex vector the first three elements refer to the real parts, and there are three further elements corresponding to the imaginary parts.

See Also

`format`, `formatC`.

Examples

```
dd <- options("digits") ; options(digits = 7) #-- for the following
format.info(123)    # 3 0 0
format.info(pi)     # 8 6 0
format.info(1e8)    # 5 0 1 - exponential "1e+08"
format.info(1e222) # 6 0 2 - exponential "1e+222"

x <- pi*10^c(-10,-2,0:2,8,20)
names(x) <- formatC(x, width=1, digits=3, format="g")
cbind(sapply(x,format))
t(sapply(x, format.info))

## using at least 8 digits right of "."
```

```
t(sapply(x, format.info, nsmall = 8))

# Reset old options:
options(dd)
```

`format.pval` *Format P Values*

Description

`format.pval` is intended for formatting p-values.

Usage

```
format.pval(pv, digits = max(1, getOption("digits") - 2),
            eps = .Machine$double.eps, na.form = "NA")
```

Arguments

`pv`	a numeric vector.
`digits`	how many significant digits are to be used.
`eps`	a numerical tolerance: see 'Details'.
`na.form`	character representation of `NA`s.

Details

`format.pval` is mainly an auxiliary function for `print.summary.lm` etc., and does separate formatting for fixed, floating point and very small values; those less than `eps` are formatted as `"< [eps]"` (where '[eps]' stands for `format(eps, digits)`).

Value

A character vector.

Examples

```
format.pval(c(stats::runif(5), pi^-100, NA))
format.pval(c(0.1, 0.0001, 1e-27))
```

formatC *Formatting Using C-style Formats*

Description

Formatting numbers individually and flexibly, using C style format specifications.

Usage

```
formatC(x, digits = NULL, width = NULL,
        format = NULL, flag = "", mode = NULL,
        big.mark = "", big.interval = 3,
      small.mark = "", small.interval = 5,
    decimal.mark = ".", preserve.width = "individual",
        zero.print = NULL, drop0trailing = FALSE)

prettyNum(x, big.mark = "",   big.interval = 3,
          small.mark = "", small.interval = 5,
          decimal.mark = ".",
          preserve.width = c("common", "individual", "none"),
          zero.print = NULL, drop0trailing = FALSE, ...)
```

Arguments

x	an atomic numerical or character object, typically a vector of real numbers.
digits	the desired number of digits after the decimal point (format = "f") or *significant* digits (format = "g", = "e" or = "fg"). Default: 2 for integer, 4 for real numbers. If less than 0, the C default of 6 digits is used. If specified as more than 50, 50 will be used with a warning unless format = "f" where it is limited to typically 324. (Not more than 15–21 digits need be accurate, depending on the OS and compiler used. This limit is just a precaution against segfaults in the underlying C runtime.)
width	the total field width; if both digits and width are unspecified, width defaults to 1, otherwise to digits + 1. width = 0 will use width = digits, width < 0 means left justify the number in this field (equivalent to flag ="-"). If necessary, the result will have more characters than width. For character data this is interpreted in characters (not bytes nor display width).
format	equal to "d" (for integers), "f", "e", "E", "g", "G", "fg" (for reals), or "s" (for strings). Default is "d" for integers, "g" for reals. "f" gives numbers in the usual xxx.xxx format; "e" and "E" give n.ddde+nn or n.dddE+nn (scientific format); "g" and "G" put x[i] into scientific format only if it saves space to do so. "fg" uses fixed format as "f", but digits as the minimum number of *significant* digits. This can lead to quite long result strings, see examples below. Note that unlike signif this prints large numbers with more significant digits than digits. Trailing zeros are *dropped* in this format, unless flag contains "#".
flag	For formatC, a character string giving a format modifier as in Kernighan and Ritchie (1988, page 243). "0" pads leading zeros; "-" does left adjustment, others are "+", " ", and "#". There can be more than one of these, in any order.

mode	"double" (or "real"), "integer" or "character". Default: Determined from the storage mode of x.
big.mark	character; if not empty used as mark between every big.interval decimals *before* (hence big) the decimal point.
big.interval	see big.mark above; defaults to 3.
small.mark	character; if not empty used as mark between every small.interval decimals *after* (hence small) the decimal point.
small.interval	see small.mark above; defaults to 5.
decimal.mark	the character to be used to indicate the numeric decimal point.
preserve.width	string specifying if the string widths should be preserved where possible in those cases where marks (big.mark or small.mark) are added. "common", the default, corresponds to format-like behavior whereas "individual" is the default in formatC().
zero.print	logical, character string or NULL specifying if and how *zeros* should be formatted specially. Useful for pretty printing 'sparse' objects.
drop0trailing	logical, indicating if trailing zeros, i.e., "0" *after* the decimal mark, should be removed; also drops "e+00" in exponential formats.
...	arguments passed to format.

Details

If you set format it overrides the setting of mode, so formatC(123.45, mode="double", format="d") gives 123.

The rendering of scientific format is platform-dependent: some systems use n.ddde+nnn or n.dddenn rather than n.ddde+nn.

formatC does not necessarily align the numbers on the decimal point, so formatC(c(6.11, 13.1), digits=2, format="fg") gives c("6.1", " 13"). If you want common formatting for several numbers, use format.

prettyNum is the utility function for prettifying x. If x is not a character, format(x[i], ...) is applied to each element, and then it is left unchanged if all the other arguments are at their defaults. Note that prettyNum(x) may behave unexpectedly if x is a character vector not resulting from something like format(<number>): in particular it assumes that a period is a decimal mark.

Value

A character object of same size and attributes as x, in the current locale's encoding. Unlike format, each number is formatted individually. Looping over each element of x, the C function sprintf(...) is called for numeric inputs (inside the C function str_signif).

formatC: for character x, do simple (left or right) padding with white space.

Author(s)

formatC was originally written by Bill Dunlap, later much improved by Martin Maechler. It was first adapted for R by Friedrich Leisch.

References

Kernighan, B. W. and Ritchie, D. M. (1988) *The C Programming Language.* Second edition. Prentice Hall.

See Also

`format`.

`sprintf` for more general C like formatting.

Examples

```
xx  <- pi * 10^(-5:4)
cbind(format(xx, digits=4), formatC(xx))
cbind(formatC(xx, width = 9, flag = "-"))
cbind(formatC(xx, digits = 5, width = 8, format = "f", flag = "0"))
cbind(format(xx, digits=4), formatC(xx, digits = 4, format = "fg"))

formatC(    c("a", "Abc", "no way"), width = -7)  # <=> flag = "-"
formatC(c((-1:1)/0,c(1,100)*pi), width=8, digits=1)

xx <- c(1e-12,-3.98765e-10,1.45645e-69,1e-70,pi*1e37,3.44e4)
##       1        2             3        4      5       6
formatC(xx)
formatC(xx, format="fg")       # special "fixed" format.
formatC(xx[1:4], format="f", digits=75) #>> even longer strings

formatC(c(3.24, 2.3e-6), format="f", digits=11, drop0trailing=TRUE)

r <- c("76491283764.97430", "29.12345678901", "-7.1234", "-100.1","1123")
## American:
prettyNum(r, big.mark = ",")
## Some Europeans:
prettyNum(r, big.mark = "'", decimal.mark = ",")

(dd <- sapply(1:10, function(i)paste((9:0)[1:i],collapse="")))
prettyNum(dd, big.mark="'")

## examples of 'small.mark'
pN <- stats::pnorm(1:7, lower.tail = FALSE)
cbind(format (pN, small.mark = " ", digits = 15))
cbind(formatC(pN, small.mark = " ", digits = 17, format = "f"))

cbind(ff <- format(1.2345 + 10^(0:5), width = 11, big.mark = "'"))
## all with same width (one more than the specified minimum)

## individual formatting to common width:
fc <- formatC(1.234 + 10^(0:8), format="fg", width=11, big.mark = "'")
cbind(fc)
```

`formatDL` *Format Description Lists*

Description

Format vectors of items and their descriptions as 2-column tables or LaTeX-style description lists.

Usage

```
formatDL(x, y, style = c("table", "list"),
         width = 0.9 * getOption("width"), indent = NULL)
```

Arguments

x	a vector giving the items to be described, or a list of length 2 or a matrix with 2 columns giving both items and descriptions.
y	a vector of the same length as x with the corresponding descriptions. Only used if x does not already give the descriptions.
style	a character string specifying the rendering style of the description information. If "table", a two-column table with items and descriptions as columns is produced (similar to Texinfo's @table environment. If "list", a LaTeX-style tagged description list is obtained.
width	a positive integer giving the target column for wrapping lines in the output.
indent	a positive integer specifying the indentation of the second column in table style, and the indentation of continuation lines in list style. Must not be greater than width/2, and defaults to width/3 for table style and width/9 for list style.

Details

After extracting the vectors of items and corresponding descriptions from the arguments, both are coerced to character vectors.

In table style, items with more than indent - 3 characters are displayed on a line of their own.

Value

a character vector with the formatted entries.

Examples

```
## Use R to create the 'INDEX' for package 'splines' from its 'CONTENTS'
x <- read.dcf(file = system.file("CONTENTS", package = "splines"),
              fields = c("Entry", "Description"))
x <- as.data.frame(x)
writeLines(formatDL(x$Entry, x$Description))
## or equivalently: writeLines(formatDL(x))
## Same information in tagged description list style:
writeLines(formatDL(x$Entry, x$Description, style = "list"))
## or equivalently: writeLines(formatDL(x, style = "list"))
```

`function` *Function Definition*

Description

These functions provide the base mechanisms for defining new functions in the R language.

Usage

```
function( arglist ) expr
return(value)
```

Arguments

`arglist`	Empty or one or more name or name=expression terms.
`value`	An expression.

Details

The names in an argument list can be back-quoted non-standard names (see 'backquote').

If `value` is missing, `NULL` is returned. If it is a single expression, the value of the evaluated expression is returned. (The expression is evaluated as soon as `return` is called, in the evaluation frame of the function and before any `on.exit` expression is evaluated.)

If the end of a function is reached without calling `return`, the value of the last evaluated expression is returned.

Warning

Prior to R 1.8.0, `value` could be a series of non-empty expressions separated by commas. In that case the value returned is a list of the evaluated expressions, with names set to the expressions where these are the names of R objects. That is, `a=foo()` names the list component `a` and gives it the value which results from evaluating `foo()`.

This has been deprecated (and a warning is given), as it was never documented in S, and whether or not the list is named differs by S versions. Supply a (named) list `value` instead.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`args` and `body` for accessing the arguments and body of a function.

`debug` for debugging; using `invisible` inside `return(.)` for returning *invisibly*.

Examples

```
norm <- function(x) sqrt(x%*%x)
norm(1:4)

## An anonymous function:
(function(x,y){ z <- x^2 + y^2; x+y+z })(0:7, 1)
```

`funprog` *Common Higher-Order Functions in Functional Programming Languages*

Description

`Reduce` uses a binary function to successively combine the elements of a given vector and a possibly given initial value. `Filter` extracts the elements of a vector for which a predicate (logical) function gives true. `Find` and `Position` give the first or last such element and its position in the vector, respectively. `Map` applies a function to the corresponding elements of given vectors. `Negate` creates the negation of a given function.

Usage

```
Reduce(f, x, init, right = FALSE, accumulate = FALSE)
Filter(f, x)
Find(f, x, right = FALSE, nomatch = NULL)
Map(f, ...)
Negate(f)
Position(f, x, right = FALSE, nomatch = NA_integer_)
```

Arguments

`f`	a function of the appropriate arity (binary for `Reduce`, unary for `Filter`, `Find` and `Position`, k-ary for `Map` if this is called with k arguments). An arbitrary predicate function for `Negate`.
`x`	a vector.
`init`	an R object of the same kind as the elements of `x`.
`right`	a logical indicating whether to proceed from left to right (default) or from right to left.
`accumulate`	a logical indicating whether the successive reduce combinations should be accumulated. By default, only the final combination is used.
`nomatch`	the value to be returned in the case when "no match" (no element satisfying the predicate) is found.
`...`	vectors.

Details

If `init` is given, `Reduce` logically adds it to the start (when proceeding left to right) or the end of `x`, respectively. If this possibly augmented vector v has $n > 1$ elements, `Reduce` successively applies f to the elements of v from left to right or right to left, respectively. I.e., a left reduce computes $l_1 = f(v_1, v_2)$, $l_2 = f(l_1, v_3)$, etc., and returns $l_{n-1} = f(l_{n-2}, v_n)$, and a right reduce does $r_{n-1} = f(v_{n-1}, v_n)$, $r_{n-2} = f(v_{n-2}, r_{n-1})$ and returns $r_1 = f(v_1, r_2)$. (E.g., if v is the sequence (2, 3, 4) and f is division, left and right reduce give $(2/3)/4 = 1/6$ and $2/(3/4) = 8/3$, respectively.) If v has only a single element, this is returned; if there are no elements, `NULL` is returned. Thus, it is ensured that `f` is always called with 2 arguments.

The current implementation is non-recursive to ensure stability and scalability.

`Reduce` is patterned after Common Lisp's `reduce`. A reduce is also known as a fold (e.g., in Haskell) or an accumulate (e.g., in the C++ Standard Template Library). The accumulative version corresponds to Haskell's scan functions.

`Filter` applies the unary predicate function `f` to each element of `x`, coercing to logical if necessary, and returns the subset of `x` for which this gives true. Note that possible `NA` values are currently always taken as false; control over `NA` handling may be added in the future. `Filter` corresponds to `filter` in Haskell or `remove-if-not` in Common Lisp.

`Find` and `Position` are patterned after Common Lisp's `find-if` and `position-if`, respectively. If there is an element for which the predicate function gives true, then the first or last such element or its position is returned depending on whether `right` is false (default) or true, respectively. If there is no such element, the value specified by `nomatch` is returned. The current implementation is not optimized for performance.

`Map` is a simple wrapper to `mapply` which does not attempt to simplify the result, similar to Common Lisp's `mapcar` (with arguments being recycled, however). Future versions may allow some control of the result type.

`Negate` corresponds to Common Lisp's `complement`. Given a (predicate) function `f`, it creates a function which returns the logical negation of what `f` returns.

Examples

```
## A general-purpose adder:
add <- function(x) Reduce("+", x)
add(list(1, 2, 3))
## Like sum(), but can also used for adding matrices etc., as it will
## use the appropriate '+' method in each reduction step.
## More generally, many generics meant to work on arbitrarily many
## arguments can be defined via reduction:
FOO <- function(...) Reduce(FOO2, list(...))
FOO2 <- function(x, y) UseMethod("FOO2")
## FOO() methods can then be provided via FOO2() methods.

## A general-purpose cumulative adder:
cadd <- function(x) Reduce("+", x, accumulate = TRUE)
cadd(seq_len(7))

## A simple function to compute continued fractions:
cfrac <- function(x) Reduce(function(u, v) u + 1 / v, x, right = TRUE)
## Continued fraction approximation for pi:
cfrac(c(3, 7, 15, 1, 292))
## Continued fraction approximation for Euler's number (e):
cfrac(c(2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8))

## Iterative function application:
Funcall <- function(f, ...) f(...)
## Compute log(exp(acos(cos(0))
Reduce(Funcall, list(log, exp, acos, cos), 0, right = TRUE)
## n-fold iterate of a function, functional style:
Iterate <- function(f, n = 1)
    function(x) Reduce(Funcall, rep.int(list(f), n), x, right = TRUE)
## Continued fraction approximation to the golden ratio:
Iterate(function(x) 1 + 1 / x, 30)(1)
## which is the same as
cfrac(rep.int(1, 31))
## Computing square root approximations for x as fixed points of the
```

```
## function t |-> (t + x / t) / 2, as a function of the initial value:
asqrt <- function(x, n) Iterate(function(t) (t + x / t) / 2, n)
asqrt(2, 30)(10) # Starting from a positive value => +sqrt(2)
asqrt(2, 30)(-1) # Starting from a negative value => -sqrt(2)

## A list of all functions in the base environment:
funs <- Filter(is.function, sapply(ls(baseenv()), get, baseenv()))
## Functions in base with more than 10 arguments:
names(Filter(function(f) length(formals(args(f))) > 10, funs))
## Number of functions in base with a '...' argument:
length(Filter(function(f)
              any(names(formals(args(f))) %in% "..."),
              funs))

## Find all objects in the base environment which are *not* functions:
Filter(Negate(is.function),  sapply(ls(baseenv()), get, baseenv()))
```

`gc` *Garbage Collection*

Description

A call of `gc` causes a garbage collection to take place. `gcinfo` sets a flag so that automatic collection is either silent (`verbose=FALSE`) or prints memory usage statistics (`verbose=TRUE`).

Usage

```
gc(verbose = getOption("verbose"), reset=FALSE)
gcinfo(verbose)
```

Arguments

`verbose`	logical; if `TRUE`, the garbage collection prints statistics about cons cells and the space allocated for vectors.
`reset`	logical; if `TRUE` the values for maximum space used are reset to the current values.

Details

A call of `gc` causes a garbage collection to take place. This will also take place automatically without user intervention, and the primary purpose of calling `gc` is for the report on memory usage.

However, it can be useful to call `gc` after a large object has been removed, as this may prompt R to return memory to the operating system.

R allocates space for vectors in multiples of 8 bytes: hence the report of `"Vcells"`, a relict of an earlier allocator (that used a vector heap).

When `gcinfo(TRUE)` is in force, messages are sent to the message connection at each garbage collection of the form

```
Garbage collection 12 = 10+0+2 (level 0) ...
6.4 Mbytes of cons cells used (58%)
2.0 Mbytes of vectors used (32%)
```

Here the last two lines give the current memory usage rounded up to the next 0.1Mb and as a percentage of the current trigger value. The first line gives a breakdown of the number of garbage collections at various levels (for an explanation see the 'R Internals' manual).

Value

`gc` returns a matrix with rows `"Ncells"` (*cons cells*), usually 28 bytes each on 32-bit systems and 56 bytes on 64-bit systems, and `"Vcells"` (*vector cells*, 8 bytes each), and columns `"used"` and `"gc trigger"`, each also interpreted in megabytes (rounded up to the next 0.1Mb).

If maxima have been set for either `"Ncells"` or `"Vcells"`, a fifth column is printed giving the current limits in Mb (with `NA` denoting no limit).

The final two columns show the maximum space used since the last call to `gc(reset=TRUE)` (or since R started).

`gcinfo` returns the previous value of the flag.

See Also

The 'R Internals' manual.

`Memory` on R's memory management, and `gctorture` if you are an R developer.

`reg.finalizer` for actions to happen at garbage collection.

Examples

```
gc() #- do it now
gcinfo(TRUE) #-- in the future, show when R does it
x <- integer(100000); for(i in 1:18) x <- c(x,i)
gcinfo(verbose = FALSE)#-- don't show it anymore

gc(TRUE)

gc(reset=TRUE)
```

---

gc.time *Report Time Spent in Garbage Collection*

Description

This function reports the time spent in garbage collection so far in the R session while GC timing was enabled.

Usage

```
gc.time(on = TRUE)
```

Arguments

`on`	logical; if `TRUE`, GC timing is enabled.

Value

A numerical vector of length 5 giving the user CPU time, the system CPU time, the elapsed time and children's user and system CPU times (normally both zero), of time spent doing garbage collection whilst GC timing was enabled.

Warnings

This is experimental functionality, likely to be removed as soon as the next release.

The timings are rounded up by the sampling interval for timing processes, and so are likely to be over-estimates.

See Also

`gc`, `proc.time` for the timings for the session.

Examples

```
gc.time()
```

---

gctorture *Torture Garbage Collector*

Description

Provokes garbage collection on (nearly) every memory allocation. Intended to ferret out memory protection bugs. Also makes R run *very* slowly, unfortunately.

Usage

```
gctorture(on = TRUE)
```

Arguments

`on`	logical; turning it on/off.

Value

Previous value.

Author(s)

Peter Dalgaard

`get` *Return the Value of a Named Object*

Description

Search for an R object with a given name and return it.

Usage

```
get(x, pos = -1, envir = as.environment(pos), mode = "any",
    inherits = TRUE)

mget(x, envir, mode = "any",
     ifnotfound = list(function(x)
         stop(paste("value for '", x, "' not found", sep = ""),
              call. = FALSE)),
     inherits = FALSE)
```

Arguments

`x`	a variable name (given as a character string).
`pos`	where to look for the object (see the details section); if omitted, the function will search as if the name of the object appeared unquoted in an expression.
`envir`	an alternative way to specify an environment to look in; see the 'Details' section.
`mode`	the mode or type of object sought: see the 'Details' section.
`inherits`	should the enclosing frames of the environment be searched?
`ifnotfound`	A `list` of values to be used if the item is not found: it will be coerced to list if necessary.

Details

The `pos` argument can specify the environment in which to look for the object in any of several ways: as an integer (the position in the `search` list); as the character string name of an element in the search list; or as an `environment` (including using `sys.frame` to access the currently active function calls). The `envir` argument is an alternative way to specify an environment, but is primarily there for back compatibility.

This function looks to see if the name `x` has a value bound to it in the specified environment. If `inherits` is `TRUE` and a value is not found for `x` in the specified environment, the enclosing frames of the environment are searched until the name `x` is encountered. See `environment` and the 'R Language Definition' manual for details about the structure of environments and their enclosures.

Warning: `inherits = TRUE` is the default behaviour for R but not for S.

If `mode` is specified then only objects of that type are sought. The `mode` may specify one of the collections `"numeric"` and `"function"` (see `mode`): any member of the collection will suffice.

Using a `NULL` environment is equivalent to using the current environment.

For `mget` multiple values are returned in a named `list`. This is true even if only one value is requested. The value in `mode` and `ifnotfound` can be either the same length as the number of requested items or of length 1. The argument `ifnotfound` must be a list containing either the

value to use if the requested item is not found or a function of one argument which will be called if the item is not found, with argument the name of the item being requested. The default value for `inherits` is `FALSE`, in contrast to the default behavior for `get`.

`mode` here is a mixture of the meanings of `typeof` and `mode`: `"function"` covers primitive functions and operators, `"numeric"`, `"integer"`, `"real"` and `"double"` all refer to any numeric type, `"symbol"` and `"name"` are equivalent *but* `"language"` must be used.

Value

The object found. (If no object is found an error results.)

Note

The reverse of `a <- get(nam)` is `assign(nam, a)`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`exists`, `assign`.

Examples

```
get("%o%")

##test mget
e1 <- new.env()
mget(letters, e1, ifnotfound = as.list(LETTERS))
```

---

getDLLRegisteredRoutines

Reflectance Information for C/Fortran routines in a DLL

Description

This function allows us to query the set of routines in a DLL that are registered with R to enhance dynamic lookup, error handling when calling native routines, and potentially security in the future. This function provides a description of each of the registered routines in the DLL for the different interfaces, i.e. `.C`, `.Call`, `.Fortran` and `.External`.

Usage

```
getDLLRegisteredRoutines(dll, addNames = TRUE)
```

Arguments

dll	a character string or DLLInfo object. The character string specifies the file name of the DLL of interest, and is given without the file name extension (e.g., the '.dll' or '.so') and with no directory/path information. So a file 'MyPackage/libs/MyPackage.so' would be specified as 'MyPackage'. The DLLInfo objects can be obtained directly in calls to dyn.load and library.dynam, or can be found after the DLL has been loaded using getLoadedDLLs, which returns a list of DLLInfo objects (index-able by DLL file name). The DLLInfo approach avoids any ambiguities related to two DLLs having the same name but corresponding to files in different directories.
addNames	a logical value. If this is TRUE, the elements of the returned lists are named using the names of the routines (as seen by R via registration or raw name). If FALSE, these names are not computed and assigned to the lists. As a result, the call should be quicker. The name information is also available in the NativeSymbolInfo objects in the lists.

Details

This takes the registration information after it has been registered and processed by the R internals. In other words, it uses the extended information

Value

A list with four elements corresponding to the routines registered for the .C, .Call, .Fortran and .External interfaces. Each element is a list with as many elements as there were routines registered for that interface. Each element identifies a routine and is an object of class NativeSymbolInfo. An object of this class has the following fields:

name	the registered name of the routine (not necessarily the name in the C code).
address	the memory address of the routine as resolved in the loaded DLL. This may be NULL if the symbol has not yet been resolved.
dll	an object of class DLLInfo describing the DLL. This is same for all elements returned.
numParameters	the number of arguments the native routine is to be called with. In the future, we will provide information about the types of the parameters also.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

"Writing R Extensions Manual" for symbol registration. R News, Volume 1/3, September 2001. "In search of C/C++ & Fortran Symbols"

See Also

getLoadedDLLs

Examples

```
dlls <- getLoadedDLLs()
getDLLRegisteredRoutines(dlls[["base"]])

getDLLRegisteredRoutines("stats")
```

---

getLoadedDLLs *Get DLLs Loaded in Current Session*

Description

This function provides a way to get a list of all the DLLs (see `dyn.load`) that are currently loaded in the R session.

Usage

```
getLoadedDLLs()
```

Details

This queries the internal table that manages the DLLs.

Value

An object of class `"DLLInfoList"` which is a list with an element corresponding to each DLL that is currently loaded in the session. Each element is an object of class `"DLLInfo"` which has the following entries.

`name`	the abbreviated name.
`path`	the fully qualified name of the loaded DLL.
`dynamicLookup`	a logical value indicating whether R uses only the registration information to resolve symbols or whether it searches the entire symbol table of the DLL.
`handle`	a reference to the C-level data structure that provides access to the contents of the DLL. This is an object of class `"DLLHandle"`.

Note that the class `DLLInfo` has an overloaded method for `$` which can be used to resolve native symbols within that DLL. Therefore, one must access the R-level elements described above using `[[`, e.g. `x[["name"]]` or `x[["handle"]]`.

Note

We are starting to use the `handle` elements in the DLL object to resolve symbols more directly in R.

Author(s)

Duncan Temple Lang ⟨duncan@wald.ucdavis.edu⟩.

See Also

`getDLLRegisteredRoutines`, `getNativeSymbolInfo`

Examples

```
getLoadedDLLs()
```

`getNativeSymbolInfo` *Obtain a Description of one or more Native (C/Fortran) Symbols*

Description

This finds and returns as comprehensive a description of one or more dynamically loaded or 'exported' built-in native symbols. For each name, it returns information about the name of the symbol, the library in which it is located and, if available, the number of arguments it expects and by which interface it should be called (i.e `.Call`, `.C`, `.Fortran`, or `.External`). Additionally, it returns the address of the symbol and this can be passed to other C routines which can invoke. Specifically, this provides a way to explicitly share symbols between different dynamically loaded package libraries. Also, it provides a way to query where symbols were resolved, and aids diagnosing strange behavior associated with dynamic resolution.

This is vectorized in the `name` argument so can process multiple symbols in a single call. The result is a list that can be indexed by the given symbol names.

Usage

```
getNativeSymbolInfo(name, PACKAGE, unlist = TRUE,
                    withRegistrationInfo = FALSE)
```

Arguments

`name`	the name(s) of the native symbol(s) as used in a call to `is.loaded`, etc. Note that Fortran symbols should be supplied as-is, not wrapped in `symbol.For`.
`PACKAGE`	an optional argument that specifies to which DLL we restrict the search for this symbol. If this is `"base"`, we search in the R executable itself.
`unlist`	a logical value which controls how the result is returned if the function is called with the name of a single symbol. If `unlist` is `TRUE` and the number of symbol names in `name` is one, then the `NativeSymbolInfo` object is returned. If it is `FALSE`, then a list of `NativeSymbolInfo` objects is returned. This is ignored if the number of symbols passed in `name` is more than one. To be compatible with earlier versions of this function, this defaults to `TRUE`.
`withRegistrationInfo`	a logical value indicating whether, if `TRUE`, to return information that was registered with R about the symbol and its parameter types if such information is available, or if `FALSE` to return the address of the symbol.

Details

This uses the same mechanism for resolving symbols as is used in all the native interfaces (`.Call`, etc.). If the symbol has been explicitly registered by the DLL in which it is contained, information about the number of arguments and the interface by which it should be called will be returned. Otherwise, a generic native symbol object is returned.

Value

Generally, a list of `NativeSymbolInfo` elements whose elements can be indexed by the elements of `name` in the call. Each `NativeSymbolInfo` object is a list containing the following elements:

`name` the name of the symbol, as given by the `name` argument.

`address` if `withRegistrationInfo` is `FALSE`, this is the native memory address of the symbol which can be used to invoke the routine, and also to compare with other symbol addresses. This is an external pointer object and of class `NativeSymbol`. If `withRegistrationInfo` is `TRUE` and registration information is available for the symbol, then this is an object of class `RegisteredNativeSymbol` and is a reference to an internal data type that has access to the routine pointer and registration information. This too can be used in calls to `.Call`, `.C`, `.Fortran` and `.External`.

`package` a list containing 3 elements:

name the short form of the library name which can be used as the value of the `PACKAGE` argument in the different native interface functions.

path the fully qualified name of the DLL.

dynamicLookup a logical value indicating whether dynamic resolution is used when looking for symbols in this library, or only registered routines can be located.

`numParameters` the number of arguments that should be passed in a call to this routine.

Additionally, the list will have an additional class, being `CRoutine`, `CallRoutine`, `FortranRoutine` or `ExternalRoutine` corresponding to the R interface by which it should be invoked.

If any of the symbols is not found, an error is immediately raised.

If `name` contains only one symbol name and `unlist` is `TRUE`, then the single `NativeSymbolInfo` is returned rather than the list containing that one element.

Note

One motivation for accessing this reflectance information is to be able to pass native routines to C routines as function pointers in C. This allows us to treat native routines and R functions in a similar manner, such as when passing an R function to C code that makes callbacks to that function at different points in its computation (e.g., `nls`). Additionally, we can resolve the symbol just once and avoid resolving it repeatedly or using the internal cache. In the future, one may be able to treat `NativeSymbol` objects directly as callback objects.

Author(s)

Duncan Temple Lang

References

For information about registering native routines, see "In Search of C/C++ & FORTRAN Routines", R News, volume 1, number 3, 2001, p20–23 (http://CRAN.R-project.org/doc/Rnews/).

See Also

getDLLRegisteredRoutines, is.loaded, .C, .Fortran, .External, .Call, dyn.load.

Examples

```
library(stats) # normally loaded
getNativeSymbolInfo("dansari")

getNativeSymbolInfo("hcass2")  # a Fortran symbol
```

---

getNumCConverters *Management of .C argument conversion list*

Description

These functions provide facilities to manage the extensible list of converters used to translate R objects to C pointers for use in .C calls. The number and a description of each element in the list can be retrieved. One can also query and set the activity status of individual elements, temporarily ignoring them. And one can remove individual elements.

Usage

```
getNumCConverters()
getCConverterDescriptions()
getCConverterStatus()
setCConverterStatus(id, status)
removeCConverter(id)
```

Arguments

id	either a number or a string identifying the element of interest in the converter list. A string is matched against the description strings for each element to identify the element. Integers are specified starting at 1 (rather than 0).
status	a logical value specifying whether the element is to be considered active (TRUE) or not (FALSE).

Details

The internal list of converters is potentially used when converting individual arguments in a .C call. If an argument has a non-trivial class attribute, we iterate over the list of converters looking for the first that matches. If we find a matching converter, we have it create the C-level pointer corresponding to the R object. When the call to the C routine is complete, we use the same converter for that argument to reverse the conversion and create an R object from the current value in the C pointer. This is done separately for all the arguments.

The functions documented here provide R user-level capabilities for investigating and managing the list of converters. There is currently no mechanism for adding an element to the converter list within the R language. This must be done in C code using the routine R_addToCConverter().

Value

`getNumCConverters` returns an integer giving the number of elements in the list, both active and inactive.

`getCConverterDescriptions` returns a character vector containing the description string of each element of the converter list.

`getCConverterStatus` returns a logical vector with a value for each element in the converter list. Each value indicates whether that converter is active (`TRUE`) or inactive (`FALSE`). The names of the elements are the description strings returned by `getCConverterDescriptions`.

`setCConverterStatus` returns the logical value indicating the activity status of the specified element before the call to change it took effect. This is `TRUE` for active and `FALSE` for inactive.

`removeCConverter` returns `TRUE` if an element in the converter list was identified and removed. In the case that no such element was found, an error occurs.

Author(s)

Duncan Temple Lang

References

http://developer.R-project.org/CObjectConversion.pdf

See Also

`.C`

Examples

```
getNumCConverters()
getCConverterDescriptions()
getCConverterStatus()
## Not run:
old <- setCConverterStatus(1, FALSE)

setCConverterStatus(1, old)
## End(Not run)
## Not run:
removeCConverter(1)
removeCConverter(getCConverterDescriptions()[1])
## End(Not run)
```

`getpid` *Get the Process ID of the R Session*

Description

Get the process ID of the R Session. It is guaranteed by the operating system that two R sessions running simultaneously will have different IDs, but it is possible that R sessions running at different times will have the same ID.

Usage

```
Sys.getpid()
```

Value

An integer, usually a small integer between 0 and 32767 under Unix-alikes and a much small integer under Windows.

Examples

```
Sys.getpid()
```

`gettext` *Translate Text Messages*

Description

If Native Language Support was enabled in this build of R, attempt to translate character vectors or set where the translations are to be found.

Usage

```
gettext(..., domain = NULL)

ngettext(n, msg1, msg2, domain = NULL)

bindtextdomain(domain, dirname = NULL)
```

Arguments

`...`	One or more character vectors.
`domain`	The 'domain' for the translation.
`n`	a non-negative integer.
`msg1`	the message to be used in English for `n = 1`.
`msg2`	the message to be used in English for `n = 0, 2, 3, ...`.
`dirname`	The directory in which to find translated message catalogs for the domain.

Details

If `domain` is `NULL` or `""`, a domain is searched for based on the name space which contains the function calling `gettext` or `ngettext`. If a suitable domain can be found, each character string is offered for translation, and replaced by its translation into the current language if one is found.

Conventionally the domain for R warning/error messages in package **pkg** is `"R-pkg"`, and that for C-level messages is `"pkg"`.

For `gettext`, leading and trailing whitespace is ignored when looking for the translation.

`ngettext` is used where the message needs to vary by a single integer. Translating such messages is subject to very specific rules for different languages: see the GNU Gettext Manual. The string will often contain a single instance of `%d` to be used in `sprintf`. If English is used, `msg1` is returned if `n == 1` and `msg2` in all other cases.

Value

For `gettext`, a character vector, one element per string in `...`. If translation is not enabled or no domain is found or no translation is found in that domain, the original strings are returned.

For `ngettext`, a character string.

For `bindtextdomain`, a character string giving the current base directory, or `NULL` if setting it failed.

See Also

`stop` and `warning` make use of `gettext` to translate messages.

`xgettext` for extracting translatable strings from R source files.

Examples

```
bindtextdomain("R")  # non-null if and only if NLS is enabled

for(n in 0:3)
    print(sprintf(ngettext(n, "%d variable has missing values",
                              "%d variables have missing values"),
                  n))

## Not run:
## for translation, those strings should appear in R-pkg.pot as
msgid        "%d variable has missing values"
msgid_plural "%d variables have missing values"
msgstr[0] ""
msgstr[1] ""
## End(Not run)

miss <- c("one", "or", "another")
cat(ngettext(length(miss), "variable", "variables"),
    paste(sQuote(miss), collapse=", "),
    ngettext(length(miss), "contains", "contain"), "missing values\n")

## better for translators would be to use
cat(sprintf(ngettext(length(miss),
                     "variable %s contains missing values\n",
                     "variables %s contain missing values\n"),
            paste(sQuote(miss), collapse=", ")))
```

---

getwd *Get or Set Working Directory*

Description

`getwd` returns an absolute filename representing the current working directory of the R process; `setwd(dir)` is used to set the working directory to `dir`.

Usage

```
getwd()
setwd(dir)
```

Arguments

dir	A character string.

Value

getwd returns a character vector, or NULL if the working directory is not available.

setwd returns the current directory before the change, invisibly. It will give an error if it does not succeed.

Note

These functions are not implemented on all platforms.

See Also

list.files for the *contents* of a directory.

Examples

```
(WD <- getwd())
if (!is.null(WD)) setwd(WD)
```

---

gl *Generate Factor Levels*

Description

Generate factors by specifying the pattern of their levels.

Usage

```
gl(n, k, length = n*k, labels = 1:n, ordered = FALSE)
```

Arguments

n	an integer giving the number of levels.
k	an integer giving the number of replications.
length	an integer giving the length of the result.
labels	an optional vector of labels for the resulting factor levels.
ordered	a logical indicating whether the result should be ordered or not.

Value

The result has levels from 1 to n with each value replicated in groups of length k out to a total length of length.

gl is modelled on the *GLIM* function of the same name.

See Also

The underlying factor().

Examples

```
## First control, then treatment:
gl(2, 8, labels = c("Control", "Treat"))
## 20 alternating 1s and 2s
gl(2, 1, 20)
## alternating pairs of 1s and 2s
gl(2, 2, 20)
```

---

grep *Pattern Matching and Replacement*

Description

`grep` searches for matches to `pattern` (its first argument) within the character vector `x` (second argument). `regexpr` and `gregexpr` do too, but return more detail in a different format.

`sub` and `gsub` perform replacement of matches determined by regular expression matching.

Usage

```
grep(pattern, x, ignore.case = FALSE, extended = TRUE,
     perl = FALSE, value = FALSE, fixed = FALSE, useBytes = FALSE)

sub(pattern, replacement, x,
    ignore.case = FALSE, extended = TRUE, perl = FALSE,
    fixed = FALSE, useBytes = FALSE)

gsub(pattern, replacement, x,
     ignore.case = FALSE, extended = TRUE, perl = FALSE,
     fixed = FALSE, useBytes = FALSE)

regexpr(pattern, text, ignore.case = FALSE, extended = TRUE,
        perl = FALSE, fixed = FALSE, useBytes = FALSE)

gregexpr(pattern, text, ignore.case = FALSE, extended = TRUE,
         perl = FALSE, fixed = FALSE, useBytes = FALSE)
```

Arguments

`pattern`	character string containing a regular expression (or character string for `fixed = TRUE`) to be matched in the given character vector. Coerced by `as.character` to a character string if possible.
`x, text`	a character vector where matches are sought, or an object which can be coerced by `as.character` to a character vector.
`ignore.case`	if `FALSE`, the pattern matching is *case sensitive* and if `TRUE`, case is ignored during matching.
`extended`	if `TRUE`, extended regular expression matching is used, and if `FALSE` basic regular expressions are used.
`perl`	logical. Should perl-compatible regexps be used? Has priority over `extended`.

value	if FALSE, a vector containing the (integer) indices of the matches determined by grep is returned, and if TRUE, a vector containing the matching elements themselves is returned.
fixed	logical. If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.
useBytes	logical. If TRUE the matching is done byte-by-byte rather than character-by-character. See 'Details'.
replacement	a replacement for matched pattern in sub and gsub. Coerced to character if possible. For fixed = FALSE this can include backreferences "\1" to "\9" to parenthesized subexpressions of pattern. For perl = TRUE only, it can also contain "\U" or "\L" to convert the rest of the replacement to upper or lower case.

Details

Arguments which should be character strings or character vectors are coerced to character if possible.

The two *sub functions differ only in that sub replaces only the first occurrence of a pattern whereas gsub replaces all occurrences.

For regexpr it is an error for pattern to be NA, otherwise NA is permitted and gives an NA match.

The regular expressions used are those specified by POSIX 1003.2, either extended or basic, depending on the value of the extended argument, unless perl = TRUE when they are those of PCRE, http://www.pcre.org/. (The exact set of patterns supported may depend on the version of PCRE installed on the system in use if R was configured to use the system PCRE.)

useBytes is only used if fixed = TRUE or perl = TRUE. Its main effect is to avoid errors/warnings about invalid inputs and spurious matches, but for regexpr it changes the interpretation of the output.

PCRE only supports caseless matching for a non-ASCII pattern in a UTF-8 locale (and not for useBytes = TRUE in any locale).

Value

For grep a vector giving either the indices of the elements of x that yielded a match or, if value is TRUE, the matched elements of x (after coercion, preserving names but no other attributes).

For sub and gsub a character vector of the same length and with the same attributes as x (after possible coercion). Elements of character vectors x which are not substituted will be return unchanged (including any declared encoding). If useBytes = FALSE, either perl = TRUE or fixed = TRUE and any element of pattern, replacement and x is declared to be in UTF-8, the result will be in UTF-8. Otherwise changed elements of the result will be have the encoding declared as that of the current locale (see Encoding if the corresponding input had a declared encoding and the current locale is either Latin-1 or UTF-8.

For regexpr an integer vector of the same length as text giving the starting position of the first match, or -1 if there is none, with attribute "match.length" giving the length of the matched text (or -1 for no match). In a multi-byte locale these quantities are in characters rather than bytes unless useBytes = TRUE is used with fixed = TRUE or perl = TRUE.

For gregexpr a list of the same length as text each element of which is an integer vector as in regexpr, except that the starting positions of every (disjoint) match are given.

If in a multi-byte locale the pattern or replacement is not a valid sequence of bytes, an error is thrown. An invalid string in `x` or `text` is a non-match with a warning for `grep` or `regexpr`, but an error for `sub` or `gsub`.

Warning

The standard regular-expression code has been reported to be very slow when applied to extremely long character strings (tens of thousands of characters or more): the code used when `perl = TRUE` seems much faster and more reliable for such usages.

The standard version of `gsub` does not substitute correctly repeated word-boundaries (e.g. `pattern = "\b"`). Use `perl = TRUE` for such matches.

The `perl = TRUE` option is only implemented for single-byte and UTF-8 encodings, and will warn if used in a non-UTF-8 multi-byte locale (unless `useBytes = TRUE`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (`grep`)

See Also

regular expression (aka `regexp`) for the details of the pattern specification.

`glob2rx` to turn wildcard matches into regular expressions.

`agrep` for approximate matching.

`tolower`, `toupper` and `chartr` for character translations. `charmatch`, `pmatch`, `match`. `apropos` uses regexps and has nice examples.

Examples

```
grep("[a-z]", letters)

txt <- c("arm","foot","lefroo", "bafoobar")
if(length(i <- grep("foo",txt)))
   cat("'foo' appears at least once in\n\t",txt,"\n")
i # 2 and 4
txt[i]

## Double all 'a' or 'b's;  "\" must be escaped, i.e., 'doubled'
gsub("([ab])", "\\1_\\1_", "abc and ABC")

txt <- c("The", "licenses", "for", "most", "software", "are",
  "designed", "to", "take", "away", "your", "freedom",
  "to", "share", "and", "change", "it.",
   "", "By", "contrast,", "the", "GNU", "General", "Public", "License",
   "is", "intended", "to", "guarantee", "your", "freedom", "to",
   "share", "and", "change", "free", "software", "--",
   "to", "make", "sure", "the", "software", "is",
   "free", "for", "all", "its", "users")
( i <- grep("[gu]", txt) ) # indices
stopifnot( txt[i] == grep("[gu]", txt, value = TRUE) )

## Note that in locales such as en_US this includes B as the
## collation order is aAbBcCdEe ...
(ot <- sub("[b-e]",".", txt))
```

```
txt[ot != gsub("[b-e]",".", txt)]#- gsub does "global" substitution

txt[gsub("g","#", txt) !=
    gsub("g","#", txt, ignore.case = TRUE)] # the "G" words

regexpr("en", txt)

gregexpr("e", txt)

## trim trailing white space
str <- 'Now is the time      '
sub(' +$', '', str)  ## spaces only
sub('[[:space:]]+$', '', str) ## white space, POSIX-style
sub('\\s+$', '', str, perl = TRUE) ## Perl-style white space

## capitalizing
gsub("(\\w)(\\w*)", "\\U\\1\\L\\2", "a test of capitalizing", perl=TRUE)
gsub("\\b(\\w)", "\\U\\1", "a test of capitalizing", perl=TRUE)
```

`groupGeneric` *S3 Group Generic Functions*

Description

Group generic methods can be defined for four pre-specified groups of functions, `Math`, `Ops`, `Summary` and `Complex`. (There are no objects of these names in base R, but there are in the **methods** package.)

A method defined for an individual member of the group takes precedence over a method defined for the group as a whole.

Usage

```
## S3 methods for group generics have prototypes:
Math(x, ...)
Ops(e1, e2)
Complex(z)
Summary(..., na.rm = FALSE)
```

Arguments

`x, z, e1, e2`	objects.
`...`	further arguments passed to methods.
`na.rm`	logical: should missing values be removed?

Details

There are four *groups* for which S3 methods can be written, namely the `"Math"`, `"Ops"`, `"Summary"` and `"Complex"` groups. These are not R objects, but methods can be supplied for them and base R contains `factor`, `data.frame` and `difftime` methods for the first three groups. (There is also a `ordered` method for `Ops`, `POSIXt` and `Date` methods for `Math` and `Ops`, `package_version` methods for `Ops` and `Summary`, as well as a `ts` method for `Ops` in package **stats**.)

1. Group "Math":
   - abs, sign, sqrt,
     floor, ceiling, trunc,
     round, signif
   - exp, log, expm1, log1p,
     cos, sin, tan,
     acos, asin, atan
     cosh, sinh, tanh,
     acosh, asinh, atanh
   - lgamma, gamma, digamma, trigamma
   - cumsum, cumprod, cummax, cummin

   Members of this group dispatch on x. Most members accept only one argument, but members log, round and signif accept one or two arguments, and trunc accepts one or more.

2. Group "Ops":
   - "+", "-", "*", "/", "^", "%%", "%/%"
   - "&", "|", "!"
   - "==", "!=", "<", "<=", ">=", ">"

   This group contains both binary and unary operators (+, - and !): when a unary operator is encountered the Ops method is called with one argument and e2 is missing.

   The classes of both arguments are considered in dispatching any member of this group. For each argument its vector of classes is examined to see if there is a matching specific (preferred) or Ops method. If a method is found for just one argument or the same method is found for both, it is used. If different methods are found, there is a warning about 'incompatible methods': in that case or if no method is found for either argument the internal method is used.

   If the members of this group are called as functions, any argument names are removed to ensure that positional matching is always used.

3. Group "Summary":
   - all, any
   - sum, prod
   - min, max
   - range

   Members of this group dispatch on the first argument supplied.

4. Group "Complex":
   - Arg, Conj, Im, Mod, Re

   Members of this group dispatch on z.

Note that a method will used for either one of these groups or one of its members *only* if it corresponds to a "class" attribute, as the internal code dispatches on oldClass and not on class. This is for efficiency: having to dispatch on, say, Ops.integer would be too slow.

The number of arguments supplied for primitive members of the "Math" group generic methods is not checked prior to dispatch.

There is no lazy evaluation of arguments for group-generic functions.

Technical Details

These functions are all primitive and internal generic.

The details of method dispatch and variables such as `.Generic` are discussed in the help for `UseMethod`. There are a few small differences:

- For the operators of group `Ops`, the object `.Method` is a length-two character vector with elements the methods selected for the left and right arguments respectively. (If no method was selected, the corresponding element is `""`.)
- Object `.Group` records the group used for dispatch (if a specific method is used this is `""`).

References

Appendix A, *Classes and Methods* of
Chambers, J. M. and Hastie, T. J. eds (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`methods` for methods of non-Internal generic functions.

S4groupGeneric for group generics for S4 methods.

Examples

```
require(utils)

d.fr <- data.frame(x=1:9, y=stats::rnorm(9))
class(1 + d.fr) == "data.frame" ##-- add to d.f. ...

methods("Math")
methods("Ops")
methods("Summary")
methods("Complex")  # none in base R
```

---

gzcon *(De)compress I/O Through Connections*

Description

`gzcon` provides a modified connection that wraps an existing connection, and decompresses reads or compresses writes through that connection. Standard `gzip` headers are assumed.

Usage

```
gzcon(con, level = 6, allowNonCompressed = TRUE)
```

Arguments

`con`	a connection.
`level`	integer between 0 and 9, the compression level when writing.
`allowNonCompressed`	logical. When reading, should non-compressed input be allowed?

Details

If `con` is open then the modified connection is opened. Closing the wrapper connection will also close the underlying connection.

Reading from a connection which does not supply a `gzip` magic header is equivalent to reading from the original connection if `allowNonCompressed` is true, otherwise an error.

The original connection becomes unusable: any object pointing to it will now refer to the modified connection.

When the connection is opened for reading, the input is expected to start with the `gzip` magic header. If it does not and if `allowNonCompressed = TRUE` (the default) the input is read as-is.

Value

An object inheriting from class `"connection"`. This is the same connection *number* as supplied, but with a modified internal structure. It has binary mode.

See Also

`gzfile`

Examples

```
## Uncompress a data file from a URL
z <- gzcon(url("http://www.stats.ox.ac.uk/pub/datasets/csb/ch12.dat.gz"))
# read.table can only read from a text-mode connection.
raw <- textConnection(readLines(z))
close(z)
dat <- read.table(raw)
close(raw)
dat[1:4, ]


## gzfile and gzcon can inter-work.
## Of course here one would used gzfile, but file() can be replaced by
## any other connection generator.
zz <- gzfile("ex.gz", "w")
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
readLines(zz <- gzcon(file("ex.gz", "rb")))
close(zz)
unlink("ex.gz")

zz <- gzcon(file("ex2.gz", "wb"))
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
readLines(zz <- gzfile("ex2.gz"))
close(zz)
unlink("ex2.gz")
```

hexmode *Display Numbers in Hexadecimal*

Description

Convert or print integers in hexadecimal format, with as many digits as are needed to display the largest, using leading zeroes as necessary.

Usage

```
as.hexmode(x)

## S3 method for class 'hexmode':
as.character(x, upper.case = FALSE, ...)

## S3 method for class 'hexmode':
format(x, upper.case = FALSE, ...)

## S3 method for class 'hexmode':
print(x, ...)
```

Arguments

`x`	An object, for the methods inheriting from class `"hexmode"`.
`upper.case`	a logical indicating whether to use upper-case letters or lower-case letters (default).
`...`	further arguments passed to or from other methods.

Details

Class `"hexmode"` consists of integer vectors with that class attribute, used merely to ensure that they are printed in hex.

See Also

`octmode`

Hyperbolic *Hyperbolic Functions*

Description

These functions give the obvious hyperbolic functions. They respectively compute the hyperbolic cosine, sine, tangent, and their inverses, arc-cosine, arc-sine, arc-tangent (or '*area cosine*', etc).

Usage

```
cosh(x)
sinh(x)
tanh(x)
acosh(x)
asinh(x)
atanh(x)
```

Arguments

`x`	a numeric or complex vector

Details

These are generic functions: methods can be defined for them individually or via the `Math` group generic.

Branch cuts are consistent with the inverse trigonometric functions `asin()` et seq, and agree with those defined in Abramowitz and Stegun, figure 4.7, page 86.

S4 methods

All are S4 generic functions: methods can be defined for them individually or via the `Math` group generic.

References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* New York: Dover. Chapter 4. Elementary Transcendental Functions: Logarithmic, Exponential, Circular and Hyperbolic Functions

See Also

The trigonometric functions, `cos`, `sin`, `tan`, and their inverses `acos`, `asin`, `atan`.

The logistic distribution function `plogis` is a shifted version of `tanh()` for numeric `x`.

`iconv` *Convert Character Vector between Encodings*

Description

This uses system facilities to convert a character vector between encodings: the 'i' stands for 'internationalization'.

Usage

```
iconv(x, from ="", to = "", sub = NA)

iconvlist()
```

Arguments

x	A character vector, or an object to be converted to a character vector by as.character.
from	A character string describing the current encoding.
to	A character string describing the target encoding.
sub	character string. If not NA it is used to replace any non-convertible bytes in the input. (This would normally be a single character, but can be more.) If "byte", the indication is "<xx>" with the hex code of the byte.

Details

The names of encodings and which ones are available (and indeed, if any are) is platform-dependent. On all systems that support iconv you can use "" for the encoding of the current locale, as well as "latin1" and "UTF-8".

On many platforms iconvlist provides an alphabetical list of the supported encodings. On others, the information is on the man page for iconv(5) or elsewhere in the man pages (and beware that the system command iconv may not support the same set of encodings as the C functions R calls). Unfortunately, the names are rarely common across platforms.

Elements of x which cannot be converted (perhaps because they are invalid or because they cannot be represented in the target encoding) will be returned as NA unless sub is specified.

Most versions of iconv will allow transliteration by appending //TRANSLIT to the to encoding: see the examples.

Any encoding bits (see Encoding) on elements of x are ignored: they will always be translated as if from from even if declared otherwise.

As from R 2.7.0 "UTF8" will be accepted as meaning the (more correct) "UTF-8".

Value

A character vector of the same length and the same attributes as x (after conversion).

The elements of the result have a declared encoding if from is "latin1" or "UTF-8", or if from = "" and the current locale's encoding is detected as Latin-1 or UTF-8.

Note

Not all platforms support these functions, although almost all support iconv. See also capabilities("iconv").

See Also

localeToCharset, file.

Examples

```
utils::head(iconvlist(), n = 50)

## Not run:
## convert from Latin-2 to UTF-8: two of the glibc iconv variants.
iconv(x, "ISO_8859-2", "UTF-8")
iconv(x, "LATIN2", "UTF-8")
## End(Not run)
```

```
## Both x below are in latin1 and will only display correctly in a
## locale that can represent and display latin1.
x <- "fa\xE7ile"
Encoding(x) <- "latin1"
x
charToRaw(xx <- iconv(x, "latin1", "UTF-8"))
xx

iconv(x, "latin1", "ASCII")           #   NA
iconv(x, "latin1", "ASCII", "?")      # "fa?ile"
iconv(x, "latin1", "ASCII", "")       # "faile"
iconv(x, "latin1", "ASCII", "byte")   # "fa<e7>ile"

# Extracts from R help files
x <- c("Ekstr\xf8m", "J\xf6reskog", "bi\xdfchen Z\xfcrcher")
Encoding(x) <- "latin1"
x
try(iconv(x, "latin1", "ASCII//TRANSLIT"))  # platform-dependent
iconv(x, "latin1", "ASCII", sub="byte")
```

`icuSetCollate` *Setup Collation by ICU*

Description

Controls the way collation is done by ICU (an optional part of the R build).

Usage

```
icuSetCollate(...)
```

Arguments

`...`	Named arguments, see 'Details'.

Details

Optionally, R can be built to collate character strings by ICU (http://www.icu-project.org). For such systems, `icuSetCollate` can be used to tune the way collation is done. On other builds calling this function does nothing, with a warning.

Possible arguments are

`locale`: A character string such as `"da_DK"` giving the country whose collation rules are to be used. If present, this should be the first argument.

`case_first`: `"upper"`, `"lower"` or `"default"`, asking for upper- or lower-case characters to be sorted first. The default is usually lower-case first, but not in all languages (see the Danish example).

`alternate_handling`: Controls the handling of 'variable' characters (mainly punctuation and symbols). Possible values are `"non_ignorable"` (primary strength) and `"shifted"` (quaternary strength).

`strength`: Which components should be used? Possible values `"primary"`, `"secondary"`, `"tertiary"` (default), `"quaternary"` and `"identical"`.

french_collation: In a French locale the way accents affect collation is from right to left, whereas in most other locales it is from left to right. Possible values "on", "off" and "default".

normalization: Should strings be normalized? Possible values "on" and "off" (default). This affects the collation of composite characters.

case_level: An additional level between secondary and tertiary, used to distinguish large and small Japanese Kana characters. Possible values "on" and "off" (default).

hiragana_quaternary: Possible values "on" (sort Hiragana first at quaternary level) and "off".

Only the first three are likely to be of interest except to those with a detailed understanding of collation and specialized requirements.

Some examples are case_level="on", strength="primary" to ignore accent differences, alternate_handling="shifted" to ignore space and punctuation characters.

See Also

Comparison, sort

The ICU user guide chapter on collation (http://www.icu-project.org/userguide/Collate_Intro.html).

Examples

```
x <- c("Aarhus", "aarhus", "safe", "test", "Zoo")
sort(x)
icuSetCollate(case_first="upper"); sort(x)
icuSetCollate(case_first="lower"); sort(x)

icuSetCollate(locale="da_DK", case_first="default"); sort(x)
icuSetCollate(locale="et_EE"); sort(x)
```

---

identical *Test Objects for Exact Equality*

Description

The safe and reliable way to test two objects for being *exactly* equal. It returns TRUE in this case, FALSE in every other case.

Usage

```
identical(x, y)
```

Arguments

x, y	any R objects.

Details

A call to `identical` is the way to test exact equality in `if` and `while` statements, as well as in logical expressions that use `&&` or `||`. In all these applications you need to be assured of getting a single logical value.

Users often use the comparison operators, such as `==` or `!=`, in these situations. It looks natural, but it is not what these operators are designed to do in R. They return an object like the arguments. If you expected `x` and `y` to be of length 1, but it happened that one of them wasn't, you will *not* get a single `FALSE`. Similarly, if one of the arguments is `NA`, the result is also `NA`. In either case, the expression `if(x == y)....` won't work as expected.

The function `all.equal` is also sometimes used to test equality this way, but was intended for something different: it allows for small differences in numeric results.

The computations in `identical` are also reliable and usually fast. There should never be an error. The only known way to kill `identical` is by having an invalid pointer at the C level, generating a memory fault. It will usually find inequality quickly. Checking equality for two large, complicated objects can take longer if the objects are identical or nearly so, but represent completely independent copies. For most applications, however, the computational cost should be negligible.

`identical` sees `NaN` as different from `NA_real_`, but all `NaN`s are equal (and all `NA` of the same type are equal).

Comparison of attributes view them as a set (and not a vector, so order is not tested).

Value

A single logical value, `TRUE` or `FALSE`, never `NA` and never anything other than a single value.

Author(s)

John Chambers and R Core

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`all.equal` for descriptions of how two objects differ; Comparison for operators that generate elementwise comparisons. `isTRUE` is a simple wrapper based on `identical`.

Examples

```
identical(1, NULL) ## FALSE -- don't try this with ==
identical(1, 1.)   ## TRUE in R (both are stored as doubles)
identical(1, as.integer(1)) ## FALSE, stored as different types

x <- 1.0; y <- 0.99999999999
## how to test for object equality allowing for numeric fuzz :
(E <- all.equal(x,y))
isTRUE(E) # which is simply defined to just use
identical(TRUE, E)
## If all.equal thinks the objects are different, it returns a
## character string, and the above expression evaluates to FALSE

# even for unusual R objects :
identical(.GlobalEnv, environment())
```

identity *Identity function*

Description

A trivial identity function returning its argument.

Usage

```
identity(x)
```

Arguments

x	an R object.

ifelse *Conditional Element Selection*

Description

`ifelse` returns a value with the same shape as `test` which is filled with elements selected from either `yes` or `no` depending on whether the element of `test` is `TRUE` or `FALSE`.

Usage

```
ifelse(test, yes, no)
```

Arguments

test	an object which can be coerced to logical mode.
yes	return values for true elements of `test`.
no	return values for false elements of `test`.

Details

If `yes` or `no` are too short, their elements are recycled. `yes` will be evaluated if and only if any element of `test` is true, and analogously for `no`.

Missing values in `test` give missing values in the result.

Value

A vector of the same length and attributes (including class) as `test` and data values from the values of `yes` or `no`. The mode of the answer will be coerced from logical to accommodate first any values taken from `yes` and then any values taken from `no`.

Warning

The mode of the result may depend on the value of `test`, and the class attribute of the result is taken from `test` and may be inappropriate for the values selected from `yes` and `no`.

Sometimes it is better to use a construction such as `(tmp <- yes; tmp[!test] <- no[!test]; tmp)`, possibly extended to handle missing values in `test`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`if`.

Examples

```
x <- c(6:-4)
sqrt(x)#- gives warning
sqrt(ifelse(x >= 0, x, NA))# no warning

## Note: the following also gives the warning !
ifelse(x >= 0, sqrt(x), NA)

## example of different return modes:
yes <- 1:3
no <- pi^(0:3)
typeof(ifelse(NA, yes, no))    # logical
typeof(ifelse(TRUE, yes, no)) # integer
typeof(ifelse(FALSE, yes, no))# double
```

`integer` *Integer Vectors*

Description

Creates or tests for objects of type `"integer"`.

Usage

```
integer(length = 0)
as.integer(x, ...)
is.integer(x)
```

Arguments

`length`	desired length.
`x`	object to be coerced or tested.
`...`	further arguments passed to or from other methods.

Details

Integer vectors exist so that data can be passed to C or Fortran code which expects them, and so that small integer data can be represented exactly and compactly.

Note that on almost all implementations of R the range of representable integers is restricted to about $\pm 2 \times 10^9$: `double`s can hold much larger integers exactly.

Value

`integer` creates a integer vector of the specified length. Each element of the vector is equal to `0`.

`as.integer` attempts to coerce its argument to be of integer type. The answer will be `NA` unless the coercion succeeds. Real values larger in modulus than the largest integer are coerced to `NA` (unlike S which gives the most extreme integer of the same sign). Non-integral numeric values are truncated towards zero (i.e., `as.integer(x)` equals `trunc(x)` there), and imaginary parts of complex numbers are discarded (with a warning). Character strings containing either a decimal representation or a hexadecimal representation (starting with `0x` or `0X`) can be converted, as well as any allowed by the platform for real numbers. Like `as.vector` it strips attributes including names. (To ensure that an object is of integer type without stripping attributes, use `storage.mode`.)

`is.integer` returns `TRUE` or `FALSE` depending on whether its argument is of integer type or not, unless it is a factor when it returns `FALSE`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`numeric`, `storage.mode`.

`round` (and `ceiling` and `floor` on that help page) to convert to integral values.

Examples

```
## as.integer() truncates:
x <- pi * c(-1:1,10)
as.integer(x)
```

`interaction` *Compute Factor Interactions*

Description

`interaction` computes a factor which represents the interaction of the given factors. The result of `interaction` is always unordered.

Usage

```
interaction(..., drop = FALSE, sep = ".", lex.order = FALSE)
```

Arguments

`...`	the factors for which interaction is to be computed, or a single list giving those factors.
`drop`	if `drop` is `TRUE`, unused factor levels are dropped from the result. The default is to retain all factor levels.
`sep`	string to construct the new level labels by joining the constituent ones.
`lex.order`	logical indicating if the order of factor concatenation should be lexically ordered.

Value

A factor which represents the interaction of the given factors. The levels are labelled as the levels of the individual factors joined by `sep` which is `.` by default.

By default, when `lex.order = FALSE`, the levels are ordered so the level of the first factor varies fastest, then the second and so on. This is the reverse of lexicographic ordering (which you can get by `lex.order = TRUE`), and differs from `:`. (It is done this way for compatibility with S.)

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`factor`; `:` where `f:g` is similar to `interaction(f, g, sep=":")` when `f` and `g` are factors.

Examples

```
a <- gl(2, 4, 8)
b <- gl(2, 2, 8, labels = c("ctrl", "treat"))
s <- gl(2, 1, 8, labels = c("M", "F"))
interaction(a, b)
interaction(a, b, s, sep = ":")
stopifnot(identical(a:s,
                    interaction(a, s, sep = ":", lex.order = TRUE)),
          identical(a:s:b,
                    interaction(a, s, b, sep = ":", lex.order = TRUE)))
```

---

interactive *Is R Running Interactively?*

Description

Return `TRUE` when R is being used interactively and `FALSE` otherwise.

Usage

```
interactive()
```

See Also

`source`, `.First`

Examples

```
.First <- function() if(interactive()) x11()
```

Internal *Call an Internal Function*

Description

`.Internal` performs a call to an internal code which is built in to the R interpreter.

Only true R wizards should even consider using this function, and only R developers can add to the list of internal functions.

Usage

```
.Internal(call)
```

Arguments

`call`	a call expression

See Also

`.Primitive`, `.External` (the nearest equivalent available to users).

InternalMethods *Internal Generic Functions*

Description

Many R-internal functions are *generic* and allow methods to be written for.

Details

The following primitive and internal functions are *generic*, i.e., you can write `methods` for them:

`[`, `[[`, `$`, `[<-`, `[[<-`, `$<-`,

`length`, `length<-`, `dimnames`, `dimnames<-`, `dim`, `dim<-`, `names`, `names<-`, `levels<-`,

`c`, `unlist`, `cbind`, `rbind`,

`as.character`, `as.complex`, `as.double`, `as.integer`, `as.logical`, `as.raw`, `as.vector`, `is.array`, `is.matrix`, `is.na`, `is.nan`, `is.numeric`, `rep` and `seq.int` (which dispatches methods for `"seq"`).

In addition, `is.name` is a synonym for `is.symbol` and dispatches methods for the latter.

Note that all of the group generic functions are also internal/primitive and allow methods to be written for them.

`.S3PrimitiveGenerics` is a character vector listing the primitives which are internal generic and not group generic. Currently `as.vector`, `cbind`, `rbind` and `unlist` are the internal non-primitive functions which are internally generic.

For efficiency, internal dispatch only occurs on *objects*, that is those for which `is.object` returns true.

See Also

`methods` for the methods which are available.

invisible *Change the Print Mode to Invisible*

Description

Return a (temporarily) invisible copy of an object.

Usage

```
invisible(x)
```

Arguments

x	an arbitrary R object.

Details

This function can be useful when it is desired to have functions return values which can be assigned, but which do not print when they are not assigned.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

withVisible, return, function.

Examples

```
# These functions both return their argument
f1 <- function(x) x
f2 <- function(x) invisible(x)
f1(1)# prints
f2(1)# does not
```

is.finite *Finite, Infinite and NaN Numbers*

Description

is.finite and is.infinite return a vector of the same length as x, indicating which elements are finite (not infinite and not missing).

Inf and -Inf are positive and negative infinity whereas NaN means 'Not a Number'. (These apply to numeric values and real and imaginary parts of complex values but not to values of integer vectors.) All are reserved words in the R language.

Usage

```
is.finite(x)
is.infinite(x)
Inf
NaN
is.nan(x)
```

Arguments

x	(numerical) object to be tested.

Details

`is.finite` returns a vector of the same length as `x` the jth element of which is `TRUE` if `x[j]` is finite (i.e., it is not one of the values `NA`, `NaN`, `Inf` or `-Inf`). All elements of types other than logical, integer, numeric and complex vectors are false. Complex numbers are finite if both the real and imaginary parts are.

`is.infinite` returns a vector of the same length as `x` the jth element of which is `TRUE` if `x[j]` is infinite (i.e., equal to one of `Inf` or `-Inf`). This will be false unless `x` is numeric or complex. Complex numbers are infinite if either the real and imaginary part is.

`is.nan` tests if a numeric value is `NaN`. Do not test equality to `NaN`, or even use `identical`, since systems typically have many different NaN values. One of these is used for the numeric missing value `NA`, and `is.nan` is false for that value. A complex number is regarded as `NaN` if either the real or imaginary part is `NaN` but not `NA`.

All three functions are generic: you can write methods to handle specific classes of objects, see InternalMethods. The default methods handle real and complex vectors.

Note

In R, basically all mathematical functions (including basic `Arithmetic`), are supposed to work properly with `+/- Inf` and `NaN` as input or output.

The basic rule should be that calls and relations with `Inf`s really are statements with a proper mathematical *limit*.

References

The IEC 60559 standard, also known as the ANSI/IEEE 754 Floating-Point Standard.

D. Goldberg (1991) *What Every Computer Scientist Should Know about Floating-Point Arithmetic* ACM Computing Surveys, **23(1)**.
Postscript version available at `http://www.validlab.com/goldberg/paper.ps` Extended PDF version at `http://www.validlab.com/goldberg/paper.pdf`

`http://grouper.ieee.org/groups/754/` for accessible information.

The C99 function `isfinite` is used for `is.finite` if available.

See Also

`NA`, '*Not Available*' which is not a number as well, however usually used for missing values and applies to many modes, not just numeric.

Examples

```
pi / 0 ## = Inf a non-zero number divided by zero creates infinity
0 / 0  ## =  NaN

1/0 + 1/0# Inf
1/0 - 1/0# NaN

stopifnot(
    1/0 == Inf,
    1/Inf == 0
)
sin(Inf)
cos(Inf)
tan(Inf)
```

---

is.function *Is an Object of Type (Primitive) Function?*

Description

Checks whether its argument is a (primitive) function.

Usage

```
is.function(x)
is.primitive(x)
```

Arguments

x	an R object.

Details

`is.primitive(x)` tests if `x` is a primitive function (either a `"builtin"` or `"special"` as described for `typeof`)?

Value

`TRUE` if `x` is a (primitive) function, and `FALSE` otherwise.

Examples

```
is.function(1) # FALSE
is.function(is.primitive)  # TRUE: it is a function, but ..
is.primitive(is.primitive) # FALSE:it's not a primitive one, whereas
is.primitive(is.function)  # TRUE: that one *is*
```

is.language *Is an Object a Language Object?*

Description

`is.language` returns `TRUE` if `x` is a variable `name`, a `call`, or an `expression`.

Usage

```
is.language(x)
```

Arguments

`x`	object to be tested.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
ll <- list(a = expression(x^2 - 2*x + 1), b = as.name("Jim"),
           c = as.expression(exp(1)), d = call("sin", pi))
sapply(ll, typeof)
sapply(ll, mode)
stopifnot(sapply(ll, is.language))
```

is.object *Is an Object "internally classed"?*

Description

A function rather for internal use. It returns `TRUE` if the object `x` has the R internal `OBJECT` bit set, and `FALSE` otherwise. The `OBJECT` bit is set when a `"class"` attribute is added and removed when that attribute is removed, so this is a very efficient way to check if an object has a class attribute. (S4 objects always should.)

Usage

```
is.object(x)
```

Arguments

`x`	object to be tested.

See Also

`class`, and `methods`.

`isS4`.

Examples

```
is.object(1) # FALSE
is.object(as.factor(1:3)) # TRUE
```

`is.R` *Are we using R, rather than S?*

Description

Test if running under R.

Usage

```
is.R()
```

Details

The function has been written such as to correctly run in all versions of R, S and S-PLUS. In order for code to be runnable in both R and S dialects previous to S-PLUS 8.0, your code must either define `is.R` or use it as

```
if (exists("is.R") && is.function(is.R) && is.R()) {
## R-specific code
} else {
## S-version of code
}
```

Value

`is.R` returns `TRUE` if we are using R and `FALSE` otherwise.

See Also

`R.version`, `system`.

Examples

```
x <- stats::runif(20); small <- x < 0.4
## In the early years of R, 'which()' only existed in R:
if(is.R()) which(small) else seq(along=small)[small]
```

`is.recursive` *Is an Object Atomic or Recursive?*

Description

`is.atomic` returns `TRUE` if `x` is an atomic vector (or `NULL`) and `FALSE` otherwise.

`is.recursive` returns `TRUE` if `x` has a recursive (list-like) structure and `FALSE` otherwise.

Usage

```
is.atomic(x)
is.recursive(x)
```

Arguments

`x`	object to be tested.

Details

`is.atomic` is true for the atomic vector types (`"logical"`, `"integer"`, `"numeric"`, `"complex"`, `"character"` and `"raw"`) and `NULL`.

Most types of language objects are regarded as recursive: those which are not are the atomic vector types, `NULL` and symbols (as given by `as.name`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`is.list`, `is.language`, etc, and the `demo("is.things")`.

Examples

```
require(stats)

is.a.r <- function(x) c(is.atomic(x), is.recursive(x))

is.a.r(c(a=1,b=3))      # TRUE FALSE
is.a.r(list())          # FALSE TRUE ??
is.a.r(list(2))         # FALSE TRUE
is.a.r(lm)              # FALSE TRUE
is.a.r(y ~ x)           # FALSE TRUE
is.a.r(expression(x+1)) # FALSE TRUE (not in 0.62.3!)
```

is.single *Is an Object of Single Precision Type?*

Description

`is.single` reports an error. There are no single precision values in R.

Usage

```
is.single(x)
```

Arguments

`x`	object to be tested.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

is.unsorted *Test if an Object is Not Sorted*

Description

Test if an object is not sorted, without the cost of sorting it.

Usage

```
is.unsorted(x, na.rm = FALSE, strictly = FALSE)
```

Arguments

`x`	an R object with a class or a numeric, complex, character or logical vector.
`na.rm`	logical. Should missing values be removed before checking?
`strictly`	logical indicating if the check should be for *strictly* increasing values.

Value

A length-one logical value. All objects of length 0 or 1 are sorted: the result will be `NA` for objects of length 2 or more except for atomic vectors and objects with a class (where the >= or > method is used).

See Also

`sort`, `order`.

isS4 *Test for an S4 object*

Description

Tests whether the object is an instance of an S4 class.

Usage

```
isS4(object)

asS4(object, value = TRUE)
```

Arguments

object	Any R object.
value	A single logical value; not NA.

Details

Note that isS4 does not rely on the **methods** package, so in particular it can be used to detect the need to require that package. (But asS4 does depend on **methods**.)

You should not set the flag directly unless you really know why. As from R 2.6.0, S4 methods are restricted to S4 objects for primitive functions; asS4 allows method dispatch of S4 methods on primitives for S3 classes. For all other purposes, an object will satisfy isS4(x) if and only if it should.

Note that S4 methods can only be set on those primitives which are 'internal generic' and %*%.

Value

isS4 always returns TRUE or FALSE according to whether the internal flag marking an S4 object has been turned on for this object.

asS4 will turn this flag on or off. But see the details.

See Also

is.object

Examples

```
isS4(pi) # FALSE
isS4(getClass("MethodDefinition")) # TRUE
```

`isSymmetric` *Test if a Matrix or other Object is Symmetric*

Description

Generic function to test if `object` is symmetric or not. Currently only a matrix method is implemented.

Usage

```
isSymmetric(object, ...)
## S3 method for class 'matrix':
isSymmetric(object, tol = 100 * .Machine$double.eps, ...)
```

Arguments

`object`	any R object; a `matrix` for the matrix method.
`tol`	numeric scalar >= 0. Smaller differences are not considered, see `all.equal.numeric`.
`...`	further arguments passed to methods; the matrix method passes these to `all.equal`.

Details

The `matrix` method is used inside `eigen` by default to test symmetry of matrices *up to rounding error*, using `all.equal`. It might not be appropriate in all situations.

Value

logical indicating if `object` is symmetric or not.

See Also

`eigen` which calls `isSymmetric` when its `symmetric` argument is missing, as per default.

Examples

```
isSymmetric(D3 <- diag(3)) # -> TRUE

D3[2,1] <- 1e-100
D3
isSymmetric(D3) # TRUE
isSymmetric(D3, tol = 0) # FALSE for zero-tolerance
```

`jitter` *Add 'Jitter' (Noise) to Numbers*

Description

Add a small amount of noise to a numeric vector.

Usage

```
jitter(x, factor=1, amount = NULL)
```

Arguments

`x`	numeric vector to which *jitter* should be added.
`factor`	numeric
`amount`	numeric; if positive, used as *amount* (see below), otherwise, if `= 0` the default is `factor * z/50`. Default (`NULL`): `factor * d/5` where `d` is about the smallest difference between `x` values.

Details

The result, say `r`, is `r <- x + runif(n, -a, a)` where `n <- length(x)` and `a` is the `amount` argument (if specified).

Let `z <- max(x) - min(x)` (assuming the usual case). The amount `a` to be added is either provided as *positive* argument `amount` or otherwise computed from `z`, as follows:

If `amount == 0`, we set `a <- factor * z/50` (same as S).

If `amount` is `NULL` (*default*), we set `a <- factor * d/5` where *d* is the smallest difference between adjacent unique (apart from fuzz) `x` values.

Value

`jitter(x,...)` returns a numeric of the same length as `x`, but with an `amount` of noise added in order to break ties.

Author(s)

Werner Stahel and Martin Maechler, ETH Zurich

References

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P.A. (1983) *Graphical Methods for Data Analysis.* Wadsworth; figures 2.8, 4.22, 5.4.

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`rug` which you may want to combine with `jitter`.

Examples

```
round(jitter(c(rep(1,3),  rep(1.2, 4), rep(3,3))), 3)
## These two 'fail' with S-plus 3.x:
jitter(rep(0, 7))
jitter(rep(10000,5))
```

`kappa` *Compute or Estimate the Condition Number of a Matrix*

Description

The condition number of a regular (square) matrix is the product of the *norm* of the matrix and the norm of its inverse (or pseudo-inverse), and hence depends on the kind of matrix-norm.

`kappa()` computes an estimate of the 2-norm condition number of a matrix or of the R matrix of a QR decomposition, perhaps of a linear fit. The 2-norm condition number can be shown to be the ratio of the largest to the smallest *non-zero* singular value of the matrix.

Usage

```
kappa(z, ...)
## Default S3 method:
kappa(z, exact = FALSE,
      norm = NULL, method = c("qr", "direct"), ...)
## S3 method for class 'lm':
kappa(z, ...)
## S3 method for class 'qr':
kappa(z, ...)

kappa.tri(z, exact = FALSE, LINPACK = TRUE, norm=NULL, ...)

rcond(x, norm = c("O","I","1"), triangular = FALSE, ...)
```

Arguments

`z,x`	A matrix or a the result of `qr` or a fit from a class inheriting from `"lm"`.
`exact`	logical. Should the result be exact?
`norm`	character string, specifying the matrix norm wrt to which the condition number is to be computed. `"O"`, the default, means the **O**ne- or 1-norm. The (currently only) other possible value is `"I"` for the infinity norm.
`method`	character string, specifying the method to be used; `"qr"` is default for back-compatibility, mainly.
`triangular`	logical. If true, the matrix used is just the lower triangular part of `z`.
`LINPACK`	logical. If true and `z` is not complex, the Linpack routine `dtrco()` is called; otherwise the relevant Lapack routine is.
`...`	further arguments passed to or from other methods.

Details

For `kappa()`, if `exact = FALSE` (the default) the 2-norm condition number is estimated by a cheap approximation. Following S, by default, this uses the LINPACK routine `dtrco()`. However, in R (or S) the exact calculation (via `svd`) is also likely to be quick enough.

Note that the 1- and Inf-norm condition numbers are much faster to calculate, and `rcond()` computes these ***r**eciprocal* condition numbers, also for complex matrices, using standard Lapack routines.

`kappa.tri` is an internal function called by `kappa.qr`.

Value

The condition number, *kappa*, or an approximation if `exact = FALSE`.

Author(s)

The design was inspired by (but differs considerably from) the S function of the same name described in Chambers (1992).

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`svd` for the singular value decomposition and `qr` for the QR one.

Examples

```
kappa(x1 <- cbind(1,1:10))# 15.71
kappa(x1, exact = TRUE)        # 13.68
kappa(x2 <- cbind(x1,2:11))# high! [x2 is singular!]

hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
sv9 <- svd(h9 <- hilbert(9))$ d
kappa(h9)# pretty high!
kappa(h9, exact = TRUE) == max(sv9) / min(sv9)
kappa(h9, exact = TRUE) / kappa(h9) # .677 (i.e., rel.error = 32%)
```

---

kronecker *Kronecker products on arrays*

Description

Computes the generalised kronecker product of two arrays, `X` and `Y`. `kronecker(X, Y)` returns an array `A` with dimensions `dim(X) * dim(Y)`.

Usage

```
kronecker(X, Y, FUN = "*", make.dimnames = FALSE, ...)
X %x% Y
```

Arguments

`X`	A vector or array.
`Y`	A vector or array.
`FUN`	a function; it may be a quoted string.
`make.dimnames`	Provide dimnames that are the product of the dimnames of `X` and `Y`.
`...`	optional arguments to be passed to `FUN`.

Details

If `X` and `Y` do not have the same number of dimensions, the smaller array is padded with dimensions of size one. The returned array comprises submatrices constructed by taking `X` one term at a time and expanding that term as `FUN(x, Y, ...)`.

`%x%` is an alias for `kronecker` (where `FUN` is hardwired to `"*"`).

Author(s)

Jonathan Rougier, ⟨J.C.Rougier@durham.ac.uk⟩

References

Shayle R. Searle (1982) *Matrix Algebra Useful for Statistics.* John Wiley and Sons.

See Also

`outer`, on which `kronecker` is built and `%*%` for usual matrix multiplication.

Examples

```
# simple scalar multiplication
( M <- matrix(1:6, ncol=2) )
kronecker(4, M)
# Block diagonal matrix:
kronecker(diag(1, 3), M)

# ask for dimnames

fred <- matrix(1:12, 3, 4, dimnames=list(LETTERS[1:3], LETTERS[4:7]))
bill <- c("happy" = 100, "sad" = 1000)
kronecker(fred, bill, make.dimnames = TRUE)

bill <- outer(bill, c("cat"=3, "dog"=4))
kronecker(fred, bill, make.dimnames = TRUE)
```

l10n_info *Localization Information*

Description

Report on localization information.

Usage

```
l10n_info()
```

Value

A list with three logical components:

MBCS	If a multi-byte character set in use?
UTF-8	Is this a UTF-8 locale?
Latin-1	Is this a Latin-1 locale?

See Also

Sys.getlocale, localeconv

Examples

```
l10n_info()
```

labels *Find Labels from Object*

Description

Find a suitable set of labels from an object for use in printing or plotting, for example. A generic function.

Usage

```
labels(object, ...)
```

Arguments

object	Any R object: the function is generic.
...	further arguments passed to or from other methods.

Value

A character vector or list of such vectors. For a vector the results is the names or `seq(along=x)` and for a data frame or array it is the dimnames (with `NULL` expanded to `seq(len=d[i])`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

`lapply` *Apply a Function over a List or Vector*

Description

`lapply` returns a list of the same length as `X`, each element of which is the result of applying `FUN` to the corresponding element of `X`.

`sapply` is a user-friendly version of `lapply` by default returning a vector or matrix if appropriate.

`replicate` is a wrapper for the common use of `sapply` for repeated evaluation of an expression (which will usually involve random number generation).

Usage

```
lapply(X, FUN, ...)

sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)

replicate(n, expr, simplify = TRUE)
```

Arguments

`X`	a vector (atomic or list) or an expressions vector. Other objects (including classed objects) will be coerced by `as.list`.
`FUN`	the function to be applied to each element of `X`: see 'Details'. In the case of functions like +, `%*%`, etc., the function name must be backquoted or quoted.
`...`	optional arguments to `FUN`.
`simplify`	logical; should the result be simplified to a vector or matrix if possible?
`USE.NAMES`	logical; if `TRUE` and if `X` is character, use `X` as `names` for the result unless it had names already.
`n`	number of replications.
`expr`	expression (language object, usually a call) to evaluate repeatedly.

Details

`FUN` is found by a call to `match.fun` and typically is specified as a function or a symbol (e.g. a backquoted name) or a character string specifying a function to be searched for from the environment of the call to `lapply`.

Function `FUN` must be able to accept as input any of the elements of `X`. If the latter is an atomic vector, `FUN` will always be passed a length-one vector of the same type as `X`.

Simplification in `sapply` is only attempted if `X` has length greater than zero and if the return values from all elements of `X` are all of the same (positive) length. If the common length is one the result is a vector, and if greater than one is a matrix with a column corresponding to each element of `X`.

Users of S4 classes should pass a list to `lapply`: the internal coercion is done by the system `as.list` in the base namespace and not one defined by a user (e.g. by setting S4 methods on the system function).

Value

For `lapply` and `sapply(simplify=FALSE)`, a list.

For `sapply(simplify=TRUE)` and `replicate`: if `X` has length zero or `n = 0`, an empty list. Otherwise an atomic vector or matrix or list of the same length as `X` (of length `n` for `replicate`). If simplification occurs, the output type is determined from the highest type of the return values in the hierarchy NULL < raw < logical < integer < real < complex < character < list < expression, after coercion of pairlists tolists.

Note

`sapply(*, simplify = FALSE, USE.NAMES = FALSE)` is equivalent to `lapply(*)`.

For historical reasons, the calls created by `lapply` are unevaluated, and code has been written (e.g. `bquote`) that relies on this. This means that the recorded call is always of the form `FUN(X[[0L]], ...)`, with `0L` replaced by the current integer index. This not normally a problem, but it can be if `FUN` uses `sys.call` or `match.call` or if it is a primitive function that makes use of the call. This means that it is often safer to call primitive functions with a wrapper, so that e.g. `lapply(ll, function(x) is.numeric(x))` is required in R 2.7.1 to ensure that method dispatch for `is.numeric` occurs correctly.

If `expr` is a function call, be aware of assumptions about where it is evaluated, and in particular what `...` might refer to. You can pass additional named arguments to a function call as additional named arguments to `replicate`: see 'Examples'.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`apply`, `tapply`, `mapply` for applying a function to **m**ultiple arguments, and `rapply` for a **r**ecursive version of `lapply()`, `eapply` for applying a function to each entry in an `environment`.

Examples

```
require(stats); require(graphics)

x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
# compute the list mean for each list element
lapply(x,mean)
# median and quartiles for each list element
lapply(x, quantile, probs = 1:3/4)
sapply(x, quantile)
i39 <- sapply(3:9, seq) # list of vectors
sapply(i39, fivenum)

hist(replicate(100, mean(rexp(10))))

## use of replicate() with parameters:
foo <- function(x=1, y=2) c(x,y)
# does not work: bar <- function(n, ...) replicate(n, foo(...))
bar <- function(n, x) replicate(n, foo(x=x))
bar(5, x=3)
```

`Last.value` *Value of Last Evaluated Expression*

Description

The value of the internal evaluation of a top-level R expression is always assigned to `.Last.value` (in `package:base`) before further processing (e.g., printing).

Usage

```
.Last.value
```

Details

The value of a top-level assignment *is* put in `.Last.value`, unlike S.

Do not assign to `.Last.value` in the workspace, because this will always mask the object of the same name in `package:base`.

See Also

`eval`

Examples

```
## These will not work correctly from example(),
## but they will in make check or if pasted in,
## as example() does not run them at the top level
gamma(1:15)          # think of some intensive calculation...
fac14 <- .Last.value # keep them

library("splines") # returns invisibly
.Last.value    # shows what library(.) above returned
```

`length` *Length of an Object*

Description

Get or set the length of vectors (including lists) and factors, and of any other R object for which a method has been defined.

Usage

```
length(x)
length(x) <- value
```

Arguments

`x`	an R object. For replacement, a vector or factor.
`value`	an integer.

Details

Both functions are generic: you can write methods to handle specific classes of objects, see InternalMethods. `length<-` has a `"factor"` method.

The replacement form can be used to reset the length of a vector. If a vector is shortened, extra values are discarded and when a vector is lengthened, it is padded out to its new length with `NA`s (`nul` for raw vectors).

Value

The default method currently returns an `integer` of length 1. Since this may change in the future and may differ for other methods, programmers should not rely on it. (Should the length exceed the maximum representable integer, it is returned as `NA`.)

For vectors (including lists) and factors the length is the number of elements. For an environment it is the number of objects in the environment, and `NULL` has length 0. For expressions and pairlists (including language objects and dotlists) it is the length of the pairlist chain. All other objects (including functions) have length one: note that for functions this differs from S.

The replacement form removes all the attributes of `x` except its names.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`nchar` for counting the number of characters in character vectors.

Examples

```
length(diag(4))# = 16 (4 x 4)
length(options())# 12 or more
length(y ~ x1 + x2 + x3)# 3
length(expression(x, {y <- x^2; y+2}, x^y)) # 3

## from example(warpbreaks)
require(stats)

fm1 <- lm(breaks ~ wool * tension, data = warpbreaks)
length(fm1$call) # 3, lm() and two arguments.
length(formula(fm1)) # 3, ~ lhs rhs
```

`levels` *Levels Attributes*

Description

`levels` provides access to the levels attribute of a variable. The first form returns the value of the levels of its argument and the second sets the attribute.

Usage

```
levels(x)
levels(x) <- value
```

Arguments

`x`	an object, for example a factor.
`value`	A valid value for `levels(x)`. For the default method, `NULL` or a character vector. For the `factor` method, a vector of character strings with length at least the number of levels of `x`, or a named list specifying how to rename the levels.

Details

Both the extractor and replacement forms are generic and new methods can be written for them. The most important method for the replacment function is that for `factor`s.

For the factor replacement method, a `NA` in `value` causes that level to be removed from the levels and the elements formerly with that level to be replaced by `NA`.

Note that for a factor, replacing the levels via `levels(x) <- value` is not the same as (and is preferred to) `attr(x, "levels") <- value`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`nlevels`, `relevel`, `reorder`.

Examples

```
## assign individual levels
x <- gl(2, 4, 8)
levels(x)[1] <- "low"
levels(x)[2] <- "high"
x

## or as a group
y <- gl(2, 4, 8)
levels(y) <- c("low", "high")
y

## combine some levels
z <- gl(3, 2, 12)
levels(z) <- c("A", "B", "A")
z

## same, using a named list
z <- gl(3, 2, 12)
levels(z) <- list(A=c(1,3), B=2)
z

## we can add levels this way:
```

```
f <- factor(c("a","b"))
levels(f) <- c("c", "a", "b")
f

f <- factor(c("a","b"))
levels(f) <- list(C="C", A="a", B="b")
f
```

`libPaths` *Search Paths for Packages*

Description

`.libPaths` gets/sets the library trees within which packages are looked for.

Usage

```
.libPaths(new)

.Library
.Library.site
```

Arguments

`new`	a character vector with the locations of R library trees. Tilde expansion (`path.expand`) is done, and if any element contains one of `*?[`, globbing is done where supported by the platform: see `Sys.glob`.

Details

`.Library` is a character string giving the location of the default library, the "library" subdirectory of `R_HOME`.

`.Library.site` is a (possibly empty) character vector giving the locations of the site libraries, by default the "site-library" subdirectory of `R_HOME` (which may not exist).

`.libPaths` is used for getting or setting the library trees that R knows about (and hence uses when looking for packages). If called with argument `new`, the library search path is set to the existing directories in `unique(c(new, .Library.site, .Library))` and this is returned. If given no argument, a character vector with the currently active library trees is returned.

The library search path is initialized at startup from the environment variable `R_LIBS` (which should be a colon-separated list of directories at which R library trees are rooted) followed by those in environment variable `R_LIBS_USER`. Only directories which exist at the time will be included.

By default `R_LIBS` is unset, and `R_LIBS_USER` is set to directory "R/*R.version$platform*-library/*x.y*" of the home directory (or "Library/R/*x.y*/library" for Mac OS X AQUA builds), for R *x.y.z*.

`.Library.site` can be set via the environment variable `R_LIBS_SITE` (as a colon-separated list of library trees).

Both `R_LIBS_USER` and `R_LIBS_SITE` feature possible expansion of specifiers for R version specific information as part of the startup process. The possible conversion specifiers all start with a "`%`" and are followed by a single letter (use "`%%`" to obtain "`%`"), with currently available conversion specifications as follows:

%V R version number including the patchlevel (e.g., '2.5.0').

%v R version number excluding the patchlevel (e.g., '2.5').

%p the platform for which R was built.

%o the underlying operating system.

%a the architecture (CPU) R was built on/for.

(See version for details on R version information.)

Function .libPaths always uses the values of .Library and .Library.site in the base name space. .Library.site can be set by the site in 'Rprofile.site', which should be followed by a call to .libPaths(.libPaths()) to make use of the updated value.

Value

A character vector of file paths.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

library

Examples

```
.libPaths()                # all library trees R knows about
```

---

library *Loading and Listing of Packages*

Description

library and require load add-on packages.

.First.lib is called when a package is loaded; .Last.lib is called when a package is detached.

Usage

```
library(package, help, pos = 2, lib.loc = NULL,
        character.only = FALSE, logical.return = FALSE,
        warn.conflicts = TRUE,
        keep.source = getOption("keep.source.pkgs"),
        verbose = getOption("verbose"),
        version)

require(package, lib.loc = NULL, quietly = FALSE,
        warn.conflicts = TRUE,
        keep.source = getOption("keep.source.pkgs"),
        character.only = FALSE, version, save = TRUE)

.First.lib(libname, pkgname)
.Last.lib(libpath)
```

Arguments

`package, help`
: the name of a package, given as a name or literal character string, or a character string, depending on whether `character.only` is `FALSE` (default) or `TRUE`).

`pos`
: the position on the search list at which to attach the loaded package. Note that `.First.lib` may attach other packages, and `pos` is computed *after* `.First.lib` has been run. Can also be the name of a position on the current search list as given by `search()`.

`lib.loc`
: a character vector describing the location of R library trees to search through, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. Non-existent library trees are silently ignored.

`character.only`
: a logical indicating whether `package` or `help` can be assumed to be character strings.

`version`
: A character string denoting a version number of the package to be loaded, for use with *versioned installs*: see the section later in this document.

`logical.return`
: logical. If it is `TRUE`, `FALSE` or `TRUE` is returned to indicate success.

`warn.conflicts`
: logical. If `TRUE`, warnings are printed about `conflicts` from attaching the new package, unless that package contains an object `.conflicts.OK`. A conflict is a function masking a function, or a non-function masking a non-function.

`keep.source`
: logical. If `TRUE`, functions 'keep their source' including comments, see argument `keep.source` to `options`. This applies only to the named package, and not to any packages or name spaces which might be loaded to satisfy dependencies or imports.

  This argument does not apply to packages using lazy-loading. Whether they have kept source is determined when they are installed (and is most likely false).

`verbose`
: a logical. If `TRUE`, additional diagnostics are printed.

`quietly`
: a logical. If `TRUE`, no message confirming package loading is printed.

`save`
: logical or environment. If `TRUE`, a call to `require` from the source for a package will save the name of the required package in the variable `".required"`, allowing function `detach` to warn if a required package is detached. See section 'Packages that require other packages' below.

`libname`
: a character string giving the library directory where the package was found.

`pkgname`
: a character string giving the name of the package, including the version number if the package was installed with `--with-package-versions` (deprecated).

`libpath`
: a character string giving the complete path to the package.

Details

`library(package)` and `require(package)` both load the package with name `package`. `require` is designed for use inside other functions; it returns `FALSE` and gives a warning (rather than an error as `library()` does by default) if the package does not exist. Both functions check and update the list of currently loaded packages and do not reload a package which is already loaded. (Furthermore, if the package has a name space and a name space of that name is already

loaded, they work from the existing name space rather than reloading from the file system. If you want to reload a package, call `detach` or `unloadNamespace` first.)

To suppress messages during the loading of packages use `suppressPackageStartupMessages`: this will suppress all messages from R itself but not necessarily all those from package authors.

If `library` is called with no `package` or `help` argument, it lists all available packages in the libraries specified by `lib.loc`, and returns the corresponding information in an object of class `"libraryIQR"`. The structure of this class may change in future versions. In earlier versions of R, only the names of all available packages were returned; use `.packages(all = TRUE)` for obtaining these. Note that `installed.packages()` returns even more information.

`library(help = somename)` computes basic information about the package `somename`, and returns this in an object of class `"packageInfo"`. The structure of this class may change in future versions. When used with the default value (`NULL`) for `lib.loc`, the loaded packages are searched before the libraries.

`.First.lib` is called when a package without a name space is loaded by `library`. (For packages with name spaces see `.onLoad`.) It is called with two arguments, the name of the library directory where the package was found (i.e., the corresponding element of `lib.loc`), and the name of the package (in that order, and with the package name including the version for a versioned install, e.g. `tree_1.0-16`). It is a good place to put calls to `library.dynam` which are needed when loading a package into this function (don't call `library.dynam` directly, as this will not work if the package is not installed in a standard location). `.First.lib` is invoked after the search path interrogated by `search()` has been updated, so `as.environment(match("package:name", search()))` will return the environment in which the package is stored. If calling `.First.lib` gives an error the loading of the package is abandoned, and the package will be unavailable. Similarly, if the option `".First.lib"` has a list element with the package's name, this element is called in the same manner as `.First.lib` when the package is loaded. This mechanism allows the user to set package 'load hooks' in addition to startup code as provided by the package maintainers, but `setHook` is preferred.

`.Last.lib` is called when a package is detached. Beware that it might be called if `.First.lib` has failed, so it should be written defensively. (It is called within `try`, so errors will not stop the package being detached.)

Value

`library` returns the list of loaded (or available) packages (or `TRUE` if `logical.return` is `TRUE`). `require` returns a logical indicating whether the required package is available, invisibly

Packages that require other packages

NB: This mechanism has been almost entirely superseded by using the `Depends:` field in the 'DESCRIPTION' file of a package.

The source code for a package that requires one or more other packages should have a call to `require`, preferably near the beginning of the source, and of course before any code that uses functions, classes or methods from the other package. The default for argument `save` will save the names of all required packages in the environment of the new package. The saved package names are used by `detach` when a package is detached to warn if other packages still require the to-be-detached package. Also, if a package is installed with saved image (see INSTALL), the saved package names are used to require these packages when the new package is attached.

Formal methods

`library` takes some further actions when package **methods** is attached (as it is by default). Packages may define formal generic functions as well as re-defining functions in other packages (notably **base**) to be generic, and this information is cached whenever such a package is loaded after **methods** and re-defined functions (implicit generics) are excluded from the list of conflicts. The caching and check for conflicts require looking for a pattern of objects; the search may be avoided by defining an object `.noGenerics` (with any value) in the package. Naturally, if the package *does* have any such methods, this will prevent them from being used.

Versioned installs

Packages can be installed with version information by `R CMD INSTALL --with-package-versions` or `install.packages(installWithVers = TRUE)`. This allows more than one version of a package to be installed in a library directory, using package directory names like `foo_1.5-1`. When such packages are loaded, it is this *versioned* name that `search()` returns. Some utility functions require the versioned name and some the unversioned name (here `foo`).

If `version` *is not* specified, `library` looks first for a packages of that name, and then for versioned installs of the package, selecting the one with the latest version number. If `version` *is* specified, a versioned install with an exactly matching version is looked for.

If `version` is not specified, `require` will accept any version that is already loaded, whereas `library` will look for an unversioned install even if a versioned install is already loaded.

Loading more than one version of a package into an R session is not currently supported. Support for versioned installs is patchy.

Note

`library` and `require` can only load an *installed* package, and this is detected by having a 'DESCRIPTION' file containing a `Built:` field.

Under Unix-alikes, the code checks that the package was installed under a similar operating system as given by `R.version$platform` (the canonical name of the platform under which R was compiled), provided it contains compiled code. Packages which do not contain compiled code can be shared between Unix-alikes, but not to other OSes because of potential problems with line endings and OS-specific help files. If sub-architectures are used, the OS similarity is not checked since the OS used to build may differ (e.g. `i386-pc-linux-gnu` code can be built on an `x86_64-unknown-linux-gnu` OS).

The package name given to `library` and `require` must match the name given in the package's 'DESCRIPTION' file exactly, even on case-insensitive file systems such as MS Windows.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`.libPaths`, `.packages`.

`attach`, `detach`, `search`, `objects`, `autoload`, `library.dynam`, `data`, `install.packages` and `installed.packages`; `INSTALL`, `REMOVE`.

Examples

```
library()                   # list all available packages
library(lib.loc = .Library) # list all packages in the default library
library(help = splines)     # documentation on package 'splines'
library(splines)            # load package 'splines'
require(splines)            # the same
search()                    # "splines", too
detach("package:splines")

# if the package name is in a character vector, use
pkg <- "splines"
library(pkg, character.only = TRUE)
detach(pos = match(paste("package", pkg, sep=":"), search()))

require(pkg, character.only = TRUE)
detach(pos = match(paste("package", pkg, sep=":"), search()))

require(nonexistent)        # FALSE
## Not run:
## Suppose a package needs to call a DLL named 'fooEXT',
## where 'EXT' is the system-specific extension.  Then you should use
.First.lib <- function(lib, pkg)
  library.dynam("foo", pkg, lib)

## if you want to mask as little as possible, use
library(mypkg, pos = "package:base")
## End(Not run)
```

---

library.dynam *Loading DLLs from Packages*

Description

Load the specified file of compiled code if it has not been loaded already, or unloads it.

Usage

```
library.dynam(chname, package = NULL, lib.loc = NULL,
              verbose = getOption("verbose"),
              file.ext = .Platform$dynlib.ext, ...)

library.dynam.unload(chname, libpath,
              verbose = getOption("verbose"),
              file.ext = .Platform$dynlib.ext)

.dynLibs(new)
```

Arguments

chname	a character string naming a DLL (also known as a dynamic shared object or library) to load.
package	a character vector with the names of packages to search through, or NULL. By default, all packages in the search path are used.

lib.loc	a character vector describing the location of R library trees to search through, or NULL. The default value of NULL corresponds to all libraries currently known.
libpath	the path to the loaded package whose DLL is to be unloaded.
verbose	a logical value indicating whether an announcement is printed on the console before loading the DLL. The default value is taken from the verbose entry in the system options.
file.ext	the extension to append to the file name to specify the library to be loaded. This defaults to the appropriate value for the operating system.
...	additional arguments needed by some libraries that are passed to the call to dyn.load to control how the library is loaded.
new	a list of "DLLInfo" objects corresponding to the DLLs loaded by packages. Can be missing.

Details

See dyn.load for what sort of objects these functions handle.

library.dynam is designed to be used inside a package rather than at the command line, and should really only be used inside .First.lib or .onLoad. The system-specific extension for DLLs (e.g., '.so' or '.sl' on Unix systems, '.so' on Mac OS X, '.dll' on Windows) should not be added. Note that to allow for versioned installs, the chname argument should not be set to the pkgname argument of .First.lib or .onLoad.

If ... does not include a named argument Dllpath, dyn.load is called with DLLpath set to the package's 'libs' directory. (Currently only used under Windows.)

library.dynam.unload is designed for use in .Last.lib or .onUnload: it unloads the DLL and updates the value of .dynLibs()

.dynLibs is used for getting (with no argument) or setting the DLLs which are currently loaded by packages (using library.dynam).

Value

If chname is not specified, library.dynam returns an object of class "DLLInfoList" corresponding to the DLLs loaded by packages.

If chname is specified, an object of class "DLLInfo" that identifies the DLL and can be used in future calls is returned invisibly. For packages that have name spaces, a list of these objects is stored in the name space's environment for use at run-time.

Note that the class DLLInfo has an overloaded method for $ which can be used to resolve native symbols within that DLL.

library.dynam.unload invisibly returns an object of class "DLLInfo" identifying the DLL successfully unloaded.

.dynLibs returns an object of class "DLLInfoList" corresponding corresponding to its current value.

Warning

Do not use dyn.unload on a DLL loaded by library.dynam: use library.dynam.unload to ensure that .dynLibs gets updated. Otherwise a subsequent call to library.dynam will be told the object is already loaded.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`getLoadedDLLs` for information on `"DLLInfo"` and `"DLLInfoList"` objects.

`.First.lib`, `library`, `dyn.load`, `.packages`, `.libPaths`

`SHLIB` for how to create suitable DLLs.

Examples

```
## Which DLLs were "dynamically loaded" by packages?
library.dynam()
```

---

license *The R License Terms*

Description

The license terms under which R is distributed.

Usage

```
license()
licence()
```

Details

R is distributed under the terms of the GNU GENERAL PUBLIC LICENSE Version 2, June 1991. A copy of this license is in file '*R_HOME*/COPYING' and can be viewed by `RShowDoc("COPYING")`.

A small number of files (the API header files) are distributed under the LESSER GNU GENERAL PUBLIC LICENSE version 2.1. A copy of this license is in file '$R_DOC_DIR/COPYING.LIB' and can be viewed by `RShowDoc("COPYING.LIB")`.

---

list *Lists – Generic and Dotted Pairs*

Description

Functions to construct, coerce and check for both kinds of R lists.

Usage

```
list(...)
pairlist(...)

as.list(x, ...)
## S3 method for class 'environment':
as.list(x, all.names = FALSE, ...)
as.pairlist(x)

is.list(x)
is.pairlist(x)

alist(...)
```

Arguments

`...`	objects, possibly named.
`x`	object to be coerced or tested.
`all.names`	a logical indicating whether to copy all values or (default) only those whose names do not begin with a dot.

Details

Most lists in R internally are *Generic Vectors*, whereas traditional *dotted pair* lists (as in LISP) are available but rarely seen by users (except as `formals` of functions).

The arguments to `list` or `pairlist` are of the form `value` or `tag=value`. The functions return a list or dotted pair list composed of its arguments with each value either tagged or untagged, depending on how the argument was specified.

`alist` handles its arguments as if they described function arguments. So the values are not evaluated, and tagged arguments with no value are allowed whereas `list` simply ignores them. `alist` is most often used in conjunction with `formals`.

`as.list` attempts to coerce its argument to a list. For functions, this returns the concatenation of the list of formal arguments and the function body. For expressions, the list of constituent elements is returned. `as.list` is generic, and as the default method calls `as.vector(mode="list")` methods for `as.vector` may be invoked. `as.list` turns a factor into a list of one-element factors. All attributes will be dropped unless the argument already is a list. (This is inconsistent with functions such as `as.character`, and is for efficiency since lists can be expensive to copy.)

`is.list` returns `TRUE` if and only if its argument is a `list` *or* a `pairlist` of `length` > 0. `is.pairlist` returns `TRUE` if and only if the argument is a pairlist or `NULL` (see below).

The `"environment"` method for `as.list` copies the name-value pairs (for names not beginning with a dot) from an environment to a named list. The user can request that all named objects are copied. The list is in no particular order (the order depends on the order of creation of objects and whether the environment is hashed). No parent environments are searched. (Objects copied are duplicated so this can be an expensive operation.)

An empty pairlist, `pairlist()` is the same as `NULL`. This is different from `list()`.

`as.pairlist` is implemented as `as.vector(x, "pairlist")`, and hence will dispatch methods for the generic function `as.vector`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`vector("list", length)` for creation of a list with empty components; `c`, for concatenation; `formals`. `unlist` is an approximate inverse to `as.list()`.

'plotmath' for the use of `list` in plot annotation.

Examples

```
require(graphics)

# create a plotting structure
pts <- list(x=cars[,1], y=cars[,2])
plot(pts)

is.pairlist(.Options)  # a user-level pairlist

## "pre-allocate" an empty list of length 5
vector("list", 5)

# Argument lists
f <- function() x
# Note the specification of a "..." argument:
formals(f) <- al <- alist(x=, y=2+3, ...=)
f
al

## environment->list coercion

e1 <- new.env()
e1$a <- 10
e1$b <- 20
as.list(e1)
```

`list.files` *List the Files in a Directory/Folder*

Description

These functions produce a character vector of the names of files in the named directory.

Usage

```
list.files(path = ".", pattern = NULL, all.files = FALSE,
           full.names = FALSE, recursive = FALSE,
           ignore.case = FALSE)

       dir(path = ".", pattern = NULL, all.files = FALSE,
           full.names = FALSE, recursive = FALSE,
           ignore.case = FALSE)
```

Arguments

path	a character vector of full path names; the default corresponds to the working directory getwd(). Missing values will be ignored.
pattern	an optional regular expression. Only file names which match the regular expression will be returned.
all.files	a logical value. If FALSE, only the names of visible files are returned. If TRUE, all file names will be returned.
full.names	a logical value. If TRUE, the directory path is prepended to the file names. If FALSE, only the file names are returned.
recursive	logical. Should the listing recurse into directories?
ignore.case	logical. Should pattern-matching be case-insensitive?

Value

A character vector containing the names of the files in the specified directories, or "" if there were no files. If a path does not exist or is not a directory or is unreadable it is skipped, with a warning.

The files are sorted in alphabetical order, on the full path if full.names = TRUE. Directories are included only if recursive = FALSE.

Note

File naming conventions are platform dependent.

recursive = TRUE is not supported on all platforms and may be ignored (with a warning).

Author(s)

Ross Ihaka, Brian Ripley

See Also

file.info, file.access and files for many more file handling functions and file.choose for interactive selection.

glob2rx to convert wildcards (as used by system file commands and shells) to regular expressions.

Sys.glob for wildcard expansion on file paths.

Examples

```
list.files(R.home())
## Only files starting with a-l or r
## Note that a-l is locale-dependent, but using case-insensitive
## matching makes it unambiguous in English locales
dir("../..", pattern = "^[a-lr]",full.names=TRUE, ignore.case = TRUE)
```

load *Reload Saved Datasets*

Description

Reload datasets written with the function save.

Usage

```
load(file, envir = parent.frame())
```

Arguments

file	a (readable binary) connection or a character string giving the name of the file to load.
envir	the environment where the data should be loaded.

Details

load can load R objects saved in the current or any earlier format. It can read a compressed file (see save) directly from a file or from a suitable connection (including a call to url).

Only R objects saved in the current format (used since R 1.4.0) can be read from a connection. If no input is available on a connection a warning will be given, but any input not in the current format will result in a error.

Loading from an earlier version will give a warning about the 'magic number': magic numbers 1971:1977 are from R < 0.99.0, and R[ADX]1 from R 0.99.0 to R 1.3.1.

Value

A character vector of the names of objects created, invisibly.

Warning

Saved R objects are binary files, even those saved with ascii = TRUE, so ensure that they are transferred without conversion of end of line markers. load tries to detect this case and give an informative error message.

See Also

save, download.file.

Examples

```
## save all data
xx <- pi # to ensure there is some data
save(list = ls(all=TRUE), file= "all.Rdata")
rm(xx)

## restore the saved values to the current environment
local({
   load("all.Rdata")
   ls()
```

```
})
## restore the saved values to the user's workspace
load("all.Rdata", .GlobalEnv)

unlink("all.Rdata")

## Not run:
con <- url("http://some.where.net/R/data/example.rda")
## print the value to see what objects were created.
print(load(con))
close(con) # url() always opens the connection
## End(Not run)
```

`localeconv` *Find Details of the Numerical and Monetary Representations in the Current Locale*

Description

Get details of the numerical and monetary representations in the current locale.

Usage

```
Sys.localeconv()
```

Details

These settings are usually controlled by the environment variables `LC_NUMERIC` and `LC_MONETARY` and if not set the values of `LC_ALL` or `LANG`.

Normally R is run without looking at the value of `LC_NUMERIC`, so the decimal point remains '`.`'. So the first three of these values will not be useful unless you have set `LC_NUMERIC` in the current R session.

Value

A character vector with 18 named components. See your ISO C documentation for details of the meaning.

It is possible to compile R without support for locales, in which case the value will be `NULL`.

See Also

`Sys.setlocale` for ways to set locales.

Examples

```
Sys.localeconv()
## The results in the C locale are
##    decimal_point     thousands_sep          grouping   int_curr_symbol
##              "."                ""                ""                ""
##  currency_symbol mon_decimal_point mon_thousands_sep      mon_grouping
##                ""                ""                ""                ""
##    positive_sign     negative_sign   int_frac_digits       frac_digits
##                ""                ""             "127"             "127"
```

```
##    p_cs_precedes    p_sep_by_space     n_cs_precedes    n_sep_by_space
##            "127"             "127"             "127"             "127"
##      p_sign_posn       n_sign_posn
##            "127"             "127"

## Now try your default locale (which might be "C").
## Not run:
old <- Sys.getlocale()
Sys.setlocale(locale = "")
Sys.localeconv()
Sys.setlocale(locale = old)
## End(Not run)

## Not run: read.table("foo", dec=Sys.localeconv()["decimal_point"])
```

`locales` *Query or Set Aspects of the Locale*

Description

Get details of or set aspects of the locale for the R process.

Usage

```
Sys.getlocale(category = "LC_ALL")
Sys.setlocale(category = "LC_ALL", locale = "")
```

Arguments

`category`	character string. The following categories should always be supported: `"LC_ALL"`, `"LC_COLLATE"`, `"LC_CTYPE"`, `"LC_MONETARY"`, `"LC_NUMERIC"` and `"LC_TIME"`. Some systems will also support `"LC_MESSAGES"`, `"LC_PAPER"` and `"LC_MEASUREMENT"`.
`locale`	character string. A valid locale name on the system in use. Normally `""` (the default) will pick up the default locale for the system.

Details

The locale describes aspects of the internationalization of a program. Initially most aspects of the locale of R are set to `"C"` (which is the default for the C language and reflects North-American usage). R sets `"LC_CTYPE"` and `"LC_COLLATE"`, which allow the use of a different character set and alphabetic comparisons in that character set (including the use of `sort`), `"LC_MONETARY"` (for use by `Sys.localeconv`) and `"LC_TIME"` may affect the behaviour of `as.POSIXlt` and `strptime` and functions which use them (but not `date`).

R can be built with no support for locales, but it is normally available on Unix and is available on Windows.

The first seven categories described here are those specified by POSIX. `"LC_MESSAGES"` will be `"C"` on systems that do not support message translation, and is not supported on Windows. Trying to use an unsupported category is an error for `Sys.setlocale`.

Note that setting `"LC_ALL"` sets only `"LC_COLLATE"`, `"LC_CTYPE"`, `"LC_MONETARY"` and `"LC_TIME"`.

Attempts to set an invalid locale are ignored. There may or may not be a warning, depending on the OS.

Attempts to change the character set (by `Sys.setlocale("LC_TYPE", )`, if that implies a different character set) during a session may not work and are likely to lead to some confusion.

Value

A character string of length one describing the locale in use (after setting for `Sys.setlocale`), or an empty character string if the current locale settings are invalid or `NULL` if locale information is unavailable.

For `category = "LC_ALL"` the details of the string are system-specific: it might be a single locale name or a set of locale names separated by `"/"` (Solaris, Mac OS X) or `";"` (Windows, Linux). For portability, it is best to query categories individually: it is not necessarily the case that the result of `foo <- Sys.getlocale()` can be used in `Sys.setlocale("LC_ALL", locale = foo)`.

Warning

Setting `"LC_NUMERIC"` may cause R to function anomalously, so gives a warning. As from R 2.7.0 input conversions in R itself are unaffected, but the reading and writing of ASCII `save` files will be, as may packages. Setting it temporarily to produce graphical or text output may work well enough, but `options(OutDec)` is often preferable.

See Also

`strptime` for uses of `category = "LC_TIME"`. `Sys.localeconv` for details of numerical and monetary representations.

`l10n_info` gives some summary facts about the locale and its encoding.

Examples

```
Sys.getlocale()
Sys.getlocale("LC_TIME")
## Not run:
Sys.setlocale("LC_TIME", "de")      # Solaris 7: details are OS-dependent
Sys.setlocale("LC_TIME", "de_DE.utf8")  # Modern Linux etc.
Sys.setlocale("LC_TIME", "German") # Windows
## End(Not run)
Sys.getlocale("LC_PAPER")           # may or may not be set

Sys.setlocale("LC_COLLATE", "C")    # turn off locale-specific sorting
```

---

log *Logarithms and Exponentials*

Description

`log` computes logarithms, by default natural logarithms, `log10` computes common (i.e., base 10) logarithms, and `log2` computes binary (i.e., base 2) logarithms. The general form `log(x, base)` computes logarithms with base `base`.

`log1p(x)` computes $\log(1 + x)$ accurately also for $|x| \ll 1$ (and less accurately when $x \approx -1$).

`exp` computes the exponential function.

`expm1(x)` computes $\exp(x) - 1$ accurately also for $|x| \ll 1$.

Usage

```
log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)

log1p(x)

exp(x)
expm1(x)
```

Arguments

`x`	a numeric or complex vector.
`base`	a positive or complex number: the base with respect to which logarithms are computed. Defaults to e=`exp(1)`.

Details

All except `logb` are generic functions: methods can be defined for them individually or via the `Math` group generic.

`log10` and `log2` are only convenience wrappers, but logs to bases 10 and 2 (whether computed *via* `log` or the wrappers) will be computed more efficiently and accurately where supported by the OS. Methods can be set for them individually (and otherwise methods for `log` will be used).

`logb` is a wrapper for `log` for compatibility with S. If (S3 or S4) methods are set for `log` they will be dispatched. Do not set S4 methods on `logb` itself.

Value

A vector of the same length as `x` containing the transformed values. `log(0)` gives `-Inf`, and negative values give `NaN`.

S4 methods

`exp`, `expm1`, `log`, `log10`, `log2` and `log1p` are S4 generic and are members of the `Math` group generic.

Note that this means that the S4 generic for `log` has a signature with only one argument, `x`, but that `base` can be passed to methods (but will not be used for method selection). On the other hand, if you only set a method for the `Math` group generic then `base` argument of `log` will be ignored for your class.

Note

`log` and `logb` are the same thing in R, but `logb` is preferred if `base` is specified, for S-PLUS compatibility.

Source

`log1p` and `expm1` may be taken from the operating system, but if not available there are based on the Fortran subroutine `dlnrel` by W. Fullerton of Los Alamos Scientific Laboratory (see `http://www.netlib.org/slatec/fnlib/dlnrel.f` and (for small x) a single Newton step for the solution of `log1p(y) = x` respectively.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (for `log`, `log10` and `exp`.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (for `logb`.)

See Also

`Trig`, `sqrt`, `Arithmetic`.

Examples

```
log(exp(3))
log10(1e7)# = 7

x <- 10^-(1+2*1:9)
cbind(x, log(1+x), log1p(x), exp(x)-1, expm1(x))
```

`Logic` *Logical Operators*

Description

These operators act on logical vectors.

Usage

```
! x
x & y
x && y
x | y
x || y
xor(x, y)

isTRUE(x)
```

Arguments

`x, y`	logical vectors, or objects which can be coerced to such or for which methods have been written.

Details

`!` indicates logical negation (NOT).

`&` and `&&` indicate logical AND and `|` and `||` indicate logical OR. The shorter form performs elementwise comparisons in much the same way as arithmetic operators. The longer form evaluates left to right examining only the first element of each vector. Evaluation proceeds only until the result is determined. The longer form is appropriate for programming control-flow and typically preferred in `if` clauses.

`xor` indicates elementwise exclusive OR.

`isTRUE(x)` is an abbreviation of `identical(TRUE, x)`, and so is true if and only if `x` is a length-one logical vector with no attributes (not even names).

Numeric and complex vectors will be coerced to logical values, with zero being false and all non-zero values being true. Raw vectors are handled without any coercion for `!`, `&` and `|`, with these operators being applied bitwise (so `!` is the 1-complement).

The operators `!`, `&` and `|` are generic functions: methods can be written for them individually or via the `Ops` (or S4 `Logic`, see below) group generic function. (See `Ops` for how dispatch is computed.)

`NA` is a valid logical object. Where a component of `x` or `y` is `NA`, the result will be `NA` if the outcome is ambiguous. In other words `NA & TRUE` evaluates to `NA`, but `NA & FALSE` evaluates to `FALSE`. See the examples below.

See Syntax for the precedence of these operators: unlike many other languages (including S) the AND and OR operators do not have the same precedence (the AND operators are higher than the OR operators).

Value

For `!`, a logical or raw vector of the same length as `x`.

For `|`, `&` and `xor` a logical or raw vector. The elements of shorter vectors are recycled as necessary (with a `warning` when they are recycled only *fractionally*). The rules for determining the attributes of the result are rather complicated. Most attributes are taken from the longer argument, the first if they are of the same length. Names will be copied from the first if it is the same length as the answer, otherwise from the second if that is. For time series, these operations are allowed only if the series are compatible, when the class and `tsp` attribute of whichever is a time series (the same, if both are) are used. For arrays (and an array result) the dimensions and dimnames are taken from first argument if it is an array, otherwise the second.

For `||`, `&&` and `isTRUE`, a length-one logical vector.

S4 methods

`!`, `&` and `|` are S4 generics, the latter two part of the `Logic` group generic (and hence methods need argument names `e1, e2`).

Prior to R 2.6.0 S4 methods for `!` needed argument name `e1`, but now `x` is correct.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`TRUE` or `logical`.

`any` and `all` for OR and AND on many scalar arguments.

`Syntax` for operator precedence.

Examples

```
y <- 1 + (x <- stats::rpois(50, lambda=1.5) / 4 - 1)
x[(x > 0) & (x < 1)]    # all x values between 0 and 1
if (any(x == 0) || any(y == 0)) "zero encountered"

## construct truth tables :

x <- c(NA, FALSE, TRUE)
names(x) <- as.character(x)
outer(x, x, "&")## AND table
outer(x, x, "|")## OR  table
```

`logical` *Logical Vectors*

Description

Create or test for objects of type `"logical"`, and the basic logical constants.

Usage

```
TRUE
FALSE
T; F

logical(length = 0)
as.logical(x, ...)
is.logical(x)
```

Arguments

`length`	desired length.
`x`	object to be coerced or tested.
`...`	further arguments passed to or from other methods.

Details

`TRUE` and `FALSE` are reserved words denoting logical constants in the R language, whereas `T` and `F` are global variables whose initial values set to these. All four are `logical(1)` vectors.

Value

`logical` creates a logical vector of the specified length. Each element of the vector is equal to `FALSE`.

`as.logical` attempts to coerce its argument to be of logical type. For `factor`s, this uses the `levels` (labels). Like `as.vector` it strips attributes including names.

`is.logical` returns `TRUE` or `FALSE` depending on whether its argument is of logical type or not.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`NA`, the other logical constant.

---

lower.tri *Lower and Upper Triangular Part of a Matrix*

Description

Returns a matrix of logicals the same size of a given matrix with entries `TRUE` in the lower or upper triangle.

Usage

```
lower.tri(x, diag = FALSE)
upper.tri(x, diag = FALSE)
```

Arguments

`x`	a matrix.
`diag`	logical. Should the diagonal be included?

See Also

`diag`, `matrix`.

Examples

```
(m2 <- matrix(1:20, 4, 5))
lower.tri(m2)
m2[lower.tri(m2)] <- NA
m2
```

`ls` *List Objects*

Description

`ls` and `objects` return a vector of character strings giving the names of the objects in the specified environment. When invoked with no argument at the top level prompt, `ls` shows what data sets and functions a user has defined. When invoked with no argument inside a function, `ls` returns the names of the functions local variables. This is useful in conjunction with `browser`.

Usage

```
ls(name, pos = -1, envir = as.environment(pos),
   all.names = FALSE, pattern)
objects(name, pos= -1, envir = as.environment(pos),
        all.names = FALSE, pattern)
```

Arguments

`name`	which environment to use in listing the available objects. Defaults to the *current* environment. Although called `name` for back compatibility, in fact this argument can specify the environment in any form; see the details section.
`pos`	an alternative argument to `name` for specifying the environment as a position in the search list. Mostly there for back compatibility.
`envir`	an alternative argument to `name` for specifying the environment evaluation environment. Mostly there for back compatibility.
`all.names`	a logical value. If `TRUE`, all object names are returned. If `FALSE`, names which begin with a '`.`' are omitted.
`pattern`	an optional regular expression. Only names matching `pattern` are returned. `glob2rx` can be used to convert wildcard patterns to regular expressions.

Details

The `name` argument can specify the environment from which object names are taken in one of several forms: as an integer (the position in the `search` list); as the character string name of an element in the search list; or as an explicit `environment` (including using `sys.frame` to access the currently active function calls). By default, the environment of the call to `ls` or `objects` is used. The `pos` and `envir` arguments are an alternative way to specify an environment, but are primarily there for back compatibility.

Note that the *order* of the resulting strings is locale dependent, see `Sys.getlocale`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

glob2rx for converting wildcard patterns to regular expressions.

ls.str for a long listing based on str. apropos (or find) for finding objects in the whole search path; grep for more details on 'regular expressions'; class, methods, etc., for object-oriented programming.

Examples

```
.Ob <- 1
ls(pattern = "O")
ls(pattern= " O", all.names = TRUE)     # also shows ".[foo]"

# shows an empty list because inside myfunc no variables are defined
myfunc <- function() {ls()}
myfunc()

# define a local variable inside myfunc
myfunc <- function() {y <- 1; ls()}
myfunc()                # shows "y"
```

---

make.names *Make Syntactically Valid Names*

Description

Make syntactically valid names out of character vectors.

Usage

```
make.names(names, unique = FALSE, allow_ = TRUE)
```

Arguments

names	character vector to be coerced to syntactically valid names. This is coerced to character if necessary.
unique	logical; if TRUE, the resulting elements are unique. This may be desired for, e.g., column names.
allow_	logical. For compatibility with R prior to 1.9.0.

Details

A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as ".2way" are not valid, and neither are the reserved words.

The definition of a *letter* depends on the current locale, but only ASCII digits are considered to be digits.

The character "X" is prepended if necessary. All invalid characters are translated to ".". A missing value is translated to "NA". Names which match R keywords have a dot appended to them. Duplicated values are altered by make.unique.

Value

A character vector of same length as `names` with each changed to a syntactically valid name, in the current locale's encoding.

Note

Prior to R version 1.9.0, underscores were not valid in variable names, and code that relies on them being converted to dots will no longer work. Use `allow_ = FALSE` for back-compatibility.

`allow_ = FALSE` is also useful when creating names for export to applications which do not allow underline in names (for example, S-PLUS and some DBMSs).

See Also

`make.unique`, `names`, `character`, `data.frame`.

Examples

```
make.names(c("a and b", "a-and-b"), unique=TRUE)
# "a.and.b"  "a.and.b.1"
make.names(c("a and b", "a_and_b"), unique=TRUE)
# "a.and.b"  "a_and_b"
make.names(c("a and b", "a_and_b"), unique=TRUE, allow_=FALSE)
# "a.and.b"  "a.and.b.1"

state.name[make.names(state.name) != state.name] # those 10 with a space
```

`make.unique` *Make Character Strings Unique*

Description

Makes the elements of a character vector unique by appending sequence numbers to duplicates.

Usage

```
make.unique(names, sep = ".")
```

Arguments

`names`	a character vector
`sep`	a character string used to separate a duplicate name from its sequence number.

Details

The algorithm used by `make.unique` has the property that `make.unique(c(A, B)) == make.unique(c(make.unique(A), B))`.

In other words, you can append one string at a time to a vector, making it unique each time, and get the same result as applying `make.unique` to all of the strings at once.

If character vector `A` is already unique, then `make.unique(c(A, B))` preserves `A`.

Value

A character vector of same length as `names` with duplicates changed, in the current locale's encoding.

Author(s)

Thomas P. Minka

See Also

`make.names`

Examples

```
make.unique(c("a", "a", "a"))
make.unique(c(make.unique(c("a", "a")), "a"))

make.unique(c("a", "a", "a.2", "a"))
make.unique(c(make.unique(c("a", "a")), "a.2", "a"))

rbind(data.frame(x=1), data.frame(x=2), data.frame(x=3))
rbind(rbind(data.frame(x=1), data.frame(x=2)), data.frame(x=3))
```

`manglePackageName` *Mangle the Package Name*

Description

This function takes the package name and the package version number and pastes them together with a separating underscore.

Usage

```
manglePackageName(pkgName, pkgVersion)
```

Arguments

`pkgName`	The package name, as a character string.
`pkgVersion`	The package version, as a character string.

Value

A character string with the two inputs pasted together.

Examples

```
manglePackageName("foo", "1.2.3")
```

mapply *Apply a function to multiple list or vector arguments*

Description

mapply is a multivariate version of sapply. mapply applies FUN to the first elements of each ... argument, the second elements, the third elements, and so on. Arguments are recycled if necessary.

Vectorize returns a new function that acts as if mapply was called.

Usage

```
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
       USE.NAMES = TRUE)

Vectorize(FUN, vectorize.args = arg.names, SIMPLIFY = TRUE,
          USE.NAMES = TRUE)
```

Arguments

FUN	function to apply, found via match.fun.
...	arguments to vectorize over (list or vector).
MoreArgs	a list of other arguments to FUN.
SIMPLIFY	logical; attempt to reduce the result to a vector or matrix?
USE.NAMES	logical; use names if the first ... argument has names, or if it is a character vector, use that character vector as the names.
vectorize.args	a character vector of arguments which should be vectorized. Defaults to all arguments to FUN.

Details

The arguments named in the vectorize.args argument to Vectorize correspond to the arguments passed in the ... list to mapply. However, only those that are actually passed will be vectorized; default values will not. See the example below.

Vectorize cannot be used with primitive functions as they have no formal list.

Value

mapply returns a list, vector, or matrix.

Vectorize returns a function with the same arguments as FUN, but wrapping a call to mapply.

See Also

sapply, outer

Examples

```
require(graphics)

mapply(rep, 1:4, 4:1)

mapply(rep, times=1:4, x=4:1)

mapply(rep, times=1:4, MoreArgs=list(x=42))

# Repeat the same using Vectorize: use rep.int as rep is primitive
vrep <- Vectorize(rep.int)
vrep(1:4, 4:1)
vrep(times=1:4, x=4:1)

vrep <- Vectorize(rep.int, "times")
vrep(times=1:4, x=42)

mapply(function(x,y) seq_len(x) + y,
       c(a= 1, b=2, c= 3),  # names from first
       c(A=10, B=0, C=-10))

word <- function(C,k) paste(rep.int(C,k), collapse='')
utils::str(mapply(word, LETTERS[1:6], 6:1, SIMPLIFY = FALSE))

f <- function(x=1:3, y) c(x,y)
vf <- Vectorize(f, SIMPLIFY = FALSE)
f(1:3,1:3)
vf(1:3,1:3)
vf(y=1:3) # Only vectorizes y, not x

# Nonlinear regression contour plot, based on nls() example

SS <- function(Vm, K, resp, conc) {
    pred <- (Vm * conc)/(K + conc)
    sum((resp - pred)^2 / pred)
}
vSS <- Vectorize(SS, c("Vm", "K"))
Treated <- subset(Puromycin, state == "treated")

Vm <- seq(140, 310, len=50)
K <- seq(0, 0.15, len=40)
SSvals <- outer(Vm, K, vSS, Treated$rate, Treated$conc)
contour(Vm, K, SSvals, levels=(1:10)^2, xlab="Vm", ylab="K")
```

---

margin.table *Compute table margin*

Description

For a contingency table in array form, compute the sum of table entries for a given index.

Usage

```
margin.table(x, margin=NULL)
```

Arguments

x	an array
margin	index number (1 for rows, etc.)

Details

This is really just `apply(x, margin, sum)` packaged up for newbies, except that if `margin` has length zero you get `sum(x)`.

Value

The relevant marginal table. The class of `x` is copied to the output table, except in the summation case.

Author(s)

Peter Dalgaard

See Also

`prop.table` and `addmargins`.

Examples

```
m <- matrix(1:4,2)
margin.table(m,1)
margin.table(m,2)
```

---

mat.or.vec *Create a Matrix or a Vector*

Description

`mat.or.vec` creates an `nr` by `nc` zero matrix if `nc` is greater than 1, and a zero vector of length `nr` if `nc` equals 1.

Usage

```
mat.or.vec(nr, nc)
```

Arguments

nr, nc	numbers of rows and columns.

Examples

```
mat.or.vec(3, 1)
mat.or.vec(3, 2)
```

match *Value Matching*

Description

match returns a vector of the positions of (first) matches of its first argument in its second.

%in% is a more intuitive interface as a binary operator, which returns a logical vector indicating if there is a match or not for its left operand.

Usage

```
match(x, table, nomatch = NA_integer_, incomparables = NULL)

x %in% table
```

Arguments

x	vector or NULL: the values to be matched.
table	vector or NULL: the values to be matched against.
nomatch	the value to be returned in the case when no match is found. Note that it is coerced to integer.
incomparables	a vector of values that cannot be matched. Any value in x matching a value in this vector is assigned the nomatch value. For historical reasons, FALSE is equivalent to NULL.

Details

%in% is currently defined as

```
"%in%" <- function(x, table) match(x, table, nomatch = 0) > 0
```

Factors, raw vectors and lists are converted to character vectors, and then x and table are coerced to a common type (the later of the two types in R's ordering, logical < integer < numeric < complex < character) before matching. If incomparables has positive length it is coerced to the common type.

Matching for lists is potentially very slow and best avoided except in simple cases.

Exactly what matches what is to some extent a matter of definition. For all types, NA matches NA and no other value. For real and complex values, NaN values are regarded as matching any other NaN value, but not matching NA.

Value

A vector of the same length as x.

match: An integer vector giving the position in table of the first match if there is a match, otherwise nomatch.

If x[i] is found to equal table[j] then the value returned in the i-th position of the return value is j, for the smallest possible j. If no match is found, the value is nomatch.

%in%: A logical vector, indicating if a match was located for each element of x.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`pmatch` and `charmatch` for (*partial*) string matching, `match.arg`, etc for function argument matching. `findInterval` similarly returns a vector of positions, but finds numbers within intervals, rather than exact matches.

`is.element` for an S-compatible equivalent of `%in%`.

Examples

```
## The intersection of two sets can be defined via match():
## Simple version:  intersect <- function(x, y) y[match(x, y, nomatch = 0)]
intersect # the R function in base, slightly more careful
intersect(1:10,7:20)

1:10 %in% c(1,3,5,9)
sstr <- c("c","ab","B","bba","c","@","bla","a","Ba","%")
sstr[sstr %in% c(letters,LETTERS)]

"%w/o%" <- function(x,y) x[!x %in% y] #--  x without y
(1:10) %w/o% c(3,7,12)
```

`match.arg` *Argument Verification Using Partial Matching*

Description

`match.arg` matches `arg` against a table of candidate values as specified by `choices`, where `NULL` means to take the first one.

Usage

```
match.arg(arg, choices, several.ok = FALSE)
```

Arguments

`arg`	a character vector (of length one unless `several.ok` is `TRUE`) or `NULL`.
`choices`	a character vector of candidate values
`several.ok`	logical specifying if `arg` should be allowed to have more than one element.

Details

In the one-argument form `match.arg(arg)`, the choices are obtained from a default setting for the formal argument `arg` of the function from which `match.arg` was called. (Since default argument matching will set `arg` to `choices`, this is allowed as an exception to the 'length one unless `several.ok` is `TRUE`' rule, and returns the first element.)

Matching is done using `pmatch`, so `arg` may be abbreviated.

Value

The unabbreviated version of the exact or unique partial match if there is one; otherwise, an error is signalled if `several.ok` is false, as per default. When `several.ok` is true and more than one element of `arg` has a match, all unabbreviated versions of matches are returned.

See Also

`pmatch`, `match.fun`, `match.call`.

Examples

```
require(stats)
## Extends the example for 'switch'
center <- function(x, type = c("mean", "median", "trimmed")) {
  type <- match.arg(type)
  switch(type,
         mean = mean(x),
         median = median(x),
         trimmed = mean(x, trim = .1))
}
x <- rcauchy(10)
center(x, "t")       # Works
center(x, "med")     # Works
try(center(x, "m"))  # Error
stopifnot(identical(center(x),       center(x, "mean")),
          identical(center(x, NULL), center(x, "mean")) )

## Allowing more than one match:
match.arg(c("gauss", "rect", "ep"),
          c("gaussian", "epanechnikov", "rectangular", "triangular"),
          several.ok = TRUE)
```

`match.call` *Argument Matching*

Description

`match.call` returns a call in which all of the specified arguments are specified by their full names.

Usage

```
match.call(definition = NULL, call = sys.call(sys.parent()),
           expand.dots = TRUE)
```

Arguments

`definition`	a function, by default the function from which `match.call` is called. See details.
`call`	an unevaluated call to the function specified by `definition`, as generated by `call`.
`expand.dots`	logical. Should arguments matching `...` in the call be included or left as a `...` argument?

Details

'function' on this help page means an interpreted function (also known as a 'closure'): `match.call` does not support primitive functions (where argument matching is normally positional).

`match.call` is most commonly used in two circumstances:

- To record the call for later re-use: for example most model-fitting functions record the call as element `call` of the list they return. Here the default `expand.dots = TRUE` is appropriate.
- To pass most of the call to another function, often `model.frame`. Here the common idiom is that `expand.dots = FALSE` is used, and the `...` elememt of the matched call is removed. An alternative is to explicitly select the arguments to be passed on, as is done in `lm`.

Calling `match.call` outside a function without specifying `definition` is an error.

Value

An object of class `call`.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`sys.call()` is similar, but does *not* expand the argument names; `call`, `pmatch`, `match.arg`, `match.fun`.

Examples

```
match.call(get, call("get", "abc", i = FALSE, p = 3))
## -> get(x = "abc", pos = 3, inherits = FALSE)
fun <- function(x, lower = 0, upper = 1) {
  structure((x - lower) / (upper - lower), CALL = match.call())
}
fun(4 * atan(1), u = pi)
```

`match.fun` *Function Verification for "Function Variables"*

Description

When called inside functions that take a function as argument, extract the desired function object while avoiding undesired matching to objects of other types.

Usage

```
match.fun(FUN, descend = TRUE)
```

Arguments

`FUN`	item to match as function: a function, symbol or character string. See 'Details'.
`descend`	logical; control whether to search past non-function objects.

Details

match.fun is not intended to be used at the top level since it will perform matching in the *parent* of the caller.

If FUN is a function, it is returned. If it is a symbol (for example, enclosed in backquotes) or a character vector of length one, it will be looked up using get in the environment of the parent of the caller. If it is of any other mode, it is attempted first to get the argument to the caller as a symbol (using substitute twice), and if that fails, an error is declared.

If descend = TRUE, match.fun will look past non-function objects with the given name; otherwise if FUN points to a non-function object then an error is generated.

This is used in base functions such as apply, lapply, outer, and sweep.

Value

A function matching FUN or an error is generated.

Bugs

The descend argument is a bit of misnomer and probably not actually needed by anything. It may go away in the future.

It is impossible to fully foolproof this. If one attaches a list or data frame containing a length-one character vector with the same name as a function, it may be used (although name spaces will help).

Author(s)

Peter Dalgaard and Robert Gentleman, based on an earlier version by Jonathan Rougier.

See Also

match.arg, get

Examples

```
# Same as get("*"):
match.fun("*")
# Overwrite outer with a vector
outer <- 1:5
## Not run:
match.fun(outer, descend = FALSE) #-> Error:  not a function
## End(Not run)
match.fun(outer) # finds it anyway
is.function(match.fun("outer")) # as well
```

MathFun *Miscellaneous Mathematical Functions*

Description

These functions compute miscellaneous mathematical functions. The naming follows the standard for computer languages such as C or Fortran.

Usage

```
abs(x)
sqrt(x)
```

Arguments

x	a numeric or complex vector or array.

Details

These are generic functions: methods can be defined for them individually or via the Math group generic. For complex arguments (and the default method), z, abs(z) == Mod(z) and sqrt(z) == z^0.5.

abs(x) returns an integer vector when x is integer or logical.

S4 methods

Both are S4 generic and members of the Math group generic.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

Arithmetic for simple, log for logarithmic, sin for trigonometric, and Special for special mathematical functions.

'plotmath' for the use of sqrt in plot annotation.

Examples

```
require(stats) # for spline
require(graphics)
xx <- -9:9
plot(xx, sqrt(abs(xx)),  col = "red")
lines(spline(xx, sqrt(abs(xx)), n=101), col = "pink")
```

---

matmult *Matrix Multiplication*

Description

Multiplies two matrices, if they are conformable. If one argument is a vector, it will be promoted to either a row or column matrix to make the two arguments conformable. If both are vectors it will return the inner product (as a matrix).

Usage

```
x %*% y
```

Arguments

`x, y`	numeric or complex matrices or vectors.

Details

When a vector is promoted to a matrix, its names are not promoted to row or column names, unlike `as.matrix`.

This operator is S4 generic but not S3 generic. S4 methods need to be written for a function of two arguments named `x` and `y`.

Value

A double or complex matrix product. Use `drop` to remove dimensions which have only one level.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`matrix`, `Arithmetic`, `diag`.

Examples

```
x <- 1:4
(z <- x %*% x)     # scalar ("inner") product (1 x 1 matrix)
drop(z)              # as scalar

y <- diag(x)
z <- matrix(1:12, ncol = 3, nrow = 4)
y %*% z
y %*% x
x %*% z
```

`matrix` *Matrices*

Description

`matrix` creates a matrix from the given set of values.

`as.matrix` attempts to turn its argument into a matrix.

`is.matrix` tests if its argument is a (strict) matrix.

Usage

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
       dimnames = NULL)

as.matrix(x, ...)
## S3 method for class 'data.frame':
as.matrix(x, rownames.force = NA, ...)

is.matrix(x)
```

Arguments

data	an optional data vector.
nrow	the desired number of rows.
ncol	the desired number of columns.
byrow	logical. If FALSE (the default) the matrix is filled by columns, otherwise the matrix is filled by rows.
dimnames	A dimnames attribute for the matrix: NULL or a list of length 2 giving the row and column names respectively. An empty list is treated as NULL, and a list of length one as row names. The list can be named, and the list names will be used as names for the dimensions.
x	an R object.
...	additional arguments to be passed to or from methods.
rownames.force	logical indicating if the resulting matrix should have character (rather than NULL) rownames. The default, NA, uses NULL rownames if the data frame has 'automatic' row.names or for a zero-row data frame.

Details

If one of nrow or ncol is not given, an attempt is made to infer it from the length of data and the other parameter. If neither is given, a one-column matrix is returned.

If there are too few elements in data to fill the array, then the elements in data are recycled. If data has length zero, NA of an appropriate type is used for atomic vectors (0 for raw vectors) and NULL for lists.

is.matrix returns TRUE if x is a matrix and has a dim attribute of length 2) and FALSE otherwise. Note that a data.frame is **not** a matrix by this test. It is generic: you can write methods to handle specific classes of objects, see InternalMethods.

as.matrix is a generic function. The method for data frames will return a character matrix if there is any non-(numeric/logical/complex) column, applying format to non-character columns. Otherwise, the usual coercion hierarchy (logical < integer < double < complex) will be used, e.g., all-logical data frames will be coerced to a logical matrix, mixed logical-integer will give a integer matrix, etc.

When coercing a vector, it produces a one-column matrix, and promotes the names (if any) of the vector to the rownames of the matrix.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

data.matrix, which attempts to convert to a numeric matrix.

Examples

```
is.matrix(as.matrix(1:10))
!is.matrix(warpbreaks)# data.frame, NOT matrix!
warpbreaks[1:10,]
as.matrix(warpbreaks[1:10,]) #using as.matrix.data.frame(.) method

# Example of setting row and column names
mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol=3, byrow=TRUE,
               dimnames = list(c("row1", "row2"),
                               c("C.1", "C.2", "C.3")))
mdat
```

---

maxCol *Find Maximum Position in Matrix*

Description

Find the maximum position for each row of a matrix, breaking ties at random.

Usage

```
max.col(m, ties.method=c("random", "first", "last"))
```

Arguments

m	numerical matrix
ties.method	a character string specifying how ties are handled, "random" by default; can be abbreviated; see 'Details'.

Details

When ties.method = "random", as per default, ties are broken at random. In this case, the determination of a tie assumes that the entries are probabilities: there is a relative tolerance of 10^{-5}, relative to the largest (in magnitude, omitting infinity) entry in the row.

If ties.method = "first", max.col returns the column number of the *first* of several maxima in every row, the same as unname(unname(m, 1, unname)).
Correspondingly, ties.method = "last" returns the *last* of possibly several indices.

Value

index of a maximal value for each row, an integer vector of length nrow(m).

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* New York: Springer (4th ed).

See Also

`which.max` for vectors.

Examples

```
table(mc <- max.col(swiss))# mostly "1" and "5", 5 x "2" and once "4"
swiss[unique(print(mr <- max.col(t(swiss)))) , ] # 3 33 45 45 33 6

set.seed(1)# reproducible example:
(mm <- rbind(x = round(2*stats::runif(12)),
             y = round(5*stats::runif(12)),
             z = round(8*stats::runif(12))))
## Not run:
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    1    1    2    0    2    2    1    1     0     0     0
y    3    2    4    2    4    5    2    4    5     1     3     1
z    2    3    0    3    7    3    4    5    4     1     7     5
## End(Not run)
## column indices of all row maxima :
utils::str(lapply(1:3, function(i) which(mm[i,] == max(mm[i,]))))
max.col(mm) ; max.col(mm) # "random"
max.col(mm, "first")# -> 4 6 5
max.col(mm, "last") # -> 7 9 11
```

`mean` *Arithmetic Mean*

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`	An R object. Currently there are methods for numeric data frames, numeric vectors and dates. A complex vector is allowed for `trim = 0`, only.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
`na.rm`	a logical value indicating whether `NA` values should be stripped before the computation proceeds.
`...`	further arguments passed to or from other methods.

Value

For a data frame, a named vector with the appropriate method being applied column by column.

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), integer, numeric or complex, `NA` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`weighted.mean`, `mean.POSIXct`, `colMeans` for row and column means.

Examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))

mean(USArrests, trim = 0.2)
```

`Memory` *Memory Available for Data Storage*

Description

Use command line options to control the memory available for R.

Usage

```
R --min-vsize=vl --max-vsize=vu --min-nsize=nl --max-nsize=nu --max-ppsize=N

mem.limits(nsize = NA, vsize = NA)
```

Arguments

`vl, vu, vsize`	Heap memory in bytes.
`nl, nu, nsize`	Number of cons cells.
`N`	Number of nested `PROTECT` calls.

.

Details

R has a variable-sized workspace. There is much less need to set memory options than prior to R 1.2.0, and most users will never need to set these. They are provided both as a way to control the overall memory usage (which can also be done by operating-system facilities such as `limit` on Unix), and since setting larger values of the minima will make R slightly more efficient on large tasks.

To understand the options, one needs to know that R maintains separate areas for fixed and variable sized objects. The first of these is allocated as an array of *cons cells* (Lisp programmers will know what they are, others may think of them as the building blocks of the language itself, parse trees, etc.), and the second are thrown on a *heap* of 'Vcells' of 8 bytes each. Effectively, the inputs `vl` and `vu` are rounded up to the next multiple of 8.

Each cons cell occupies 28 bytes on a 32-bit machine, (usually) 56 bytes on a 64-bit machine.

The '`--*-nsize`' options can be used to specify the number of cons cells and the '`--*-vsize`' options specify the size of the vector heap in bytes. Both options must be integers or integers followed by `G`, `M`, `K`, or `k` meaning `Giga` ($2^{30} = 1073741824$) *Mega* ($2^{20} = 1048576$), (computer) *Kilo* ($2^{10} = 1024$), or regular *kilo* (1000).

The '`--min-*`' options set the minimal sizes for the number of cons cells and for the vector heap. These values are also the initial values, but thereafter R will grow or shrink the areas depending on usage, but never exceeding the limits set by the '`--max-*`' options nor decreasing below the initial values.

The default values are currently minima of 350k cons cells, 6Mb of vector heap and no maxima (other than machine resources). The maxima can be changed during an R session by calling `mem.limits`. (If this is called with the default values, it reports the current settings.)

You can find out the current memory consumption (the heap and cons cells used as numbers and megabytes) by typing `gc()` at the R prompt. Note that following `gcinfo(TRUE)`, automatic garbage collection always prints memory use statistics. Maxima will never be reduced below the current values for triggering garbage collection, and attempts to do so will be silently ignored.

The command-line option '`--max-ppsize`' controls the maximum size of the pointer protection stack. This defaults to 50000, but can be increased to allow deep recursion or large and complicated calculations to be done. *Note* that parts of the garbage collection process goes through the full reserved pointer protection stack and hence becomes slower when the size is increased. Currently the maximum value accepted is 500000.

Value

`mem.limits()` returns an integer vector giving the current settings of the maxima, possibly `NA`.

See Also

An Introduction to R for more command-line options

`Memory-limits` for the design limitations.

`gc` for information on the garbage collector and total memory usage, `object.size(a)` for the (approximate) size of R object `a`. `memory.profile` for profiling the usage of cons cells.

Examples

```
# Start R with 10MB of heap memory and 500k cons cells, limit to
# 100Mb and 1M cells
## Not run:
## Unix
```

```
R --min-vsize=10M --max-vsize=100M --min-nsize=500k --max-nsize=1M
## End(Not run)
```

`Memory-limits` *Memory Limits in R*

Description

R holds objects it is using in memory. This help file documents the current design limitations on large objects: these differ between 32-bit and 64-bit builds of R.

Details

R holds all objects in memory, and there are limits based on the amount of memory that can be used by all objects:

- There may be limits on the size of the heap and the number of cons cells allowed – see `Memory` – but these are usually not imposed.
- There is a limit on the address space of a single process such as the R executable. This is system-specific, but 32-bit OSes imposes a limit of no more than 4Gb: it is often 3Gb or less.
- The environment may impose limitations on the resources available to a single process – see the OS/shell's help on commands such as `limit` or `ulimit`.

Error messages beginning `cannot allocate vector of size` indicate a failure to obtain memory, either because the size exceeded the address-space limit for a process or, more likely, because the system was unable to provide the memory. Note that on a 32-bit OS there may well be enough free memory available, but not a large enough contiguous block of address space into which to map it.

There are also limits on individual objects. On all versions of R, the maximum length (number of elements) of a vector is $2^{31} - 1 \approx 2\,10^9$, as lengths are stored as signed integers. In addition, the storage space cannot exceed the address limit, and if you try to exceed that limit, the error message begins `cannot allocate vector of length`. The number of characters in a character string is in theory only limited by the address space.

See Also

`object.size(a)` for the (approximate) size of R object `a`.

`memory.profile` *Profile the Usage of Cons Cells*

Description

Lists the usage of the cons cells by `SEXPREC` type.

Usage

```
memory.profile()
```

Details

The current types and their uses are listed in the include file 'Rinternals.h'.

Value

A vector of counts, named by the types. See `typeof` for an explanation of types.

See Also

`gc` for the overall usage of cons cells. `Rprofmem` and `tracemem` allow memory profiling of specific code or objects, but need to be enabled at compile time.

Examples

```
memory.profile()
```

---

merge *Merge Two Data Frames*

Description

Merge two data frames by common columns or row names, or do other versions of database *join* operations.

Usage

```
merge(x, y, ...)

## Default S3 method:
merge(x, y, ...)

## S3 method for class 'data.frame':
merge(x, y, by = intersect(names(x), names(y)),
      by.x = by, by.y = by, all = FALSE, all.x = all, all.y = all,
      sort = TRUE, suffixes = c(".x",".y"), incomparables = NULL, ...)
```

Arguments

`x, y`	data frames, or objects to be coerced to one.
`by, by.x, by.y`	specifications of the common columns. See 'Details'.
`all`	logical; `all = L` is shorthand for `all.x = L` and `all.y = L`.
`all.x`	logical; if `TRUE`, then extra rows will be added to the output, one for each row in `x` that has no matching row in `y`. These rows will have `NA`s in those columns that are usually filled with values from `y`. The default is `FALSE`, so that only rows with data from both `x` and `y` are included in the output.
`all.y`	logical; analogous to `all.x` above.
`sort`	logical. Should the results be sorted on the `by` columns?
`suffixes`	character(2) specifying the suffixes to be used for making non-`by` `names()` unique.

`incomparables`
: values which cannot be matched. See `match`.

`...`
: arguments to be passed to or from methods.

Details

By default the data frames are merged on the columns with names they both have, but separate specifications of the columns can be given by `by.x` and `by.y`. Columns can be specified by name, number or by a logical vector: the name `"row.names"` or the number `0` specifies the row names. The rows in the two data frames that match on the specified columns are extracted, and joined together. If there is more than one match, all possible matches contribute one row each. For the precise meaning of 'match', see `match`.

If `by` or both `by.x` and `by.y` are of length 0 (a length zero vector or `NULL`), the result, `r`, is the *Cartesian product* of `x` and `y`, i.e., `dim(r) = c(nrow(x)*nrow(y), ncol(x) + ncol(y))`.

If `all.x` is true, all the non matching cases of `x` are appended to the result as well, with `NA` filled in the corresponding columns of `y`; analogously for `all.y`.

If the remaining columns in the data frames have any common names, these have `suffixes` (`".x"` and `".y"` by default) appended to make the names of the result unique.

The complexity of the algorithm used is proportional to the length of the answer.

In SQL database terminology, the default value of `all = FALSE` gives a *natural join*, a special case of an *inner join*. Specifying `all.x = TRUE` gives a *left (outer) join*, `all.y = TRUE` a *right (outer) join*, and both (`all=TRUE` a *(full) outer join*. DBMSes do not match `NULL` records, equivalent to `incomparables = NA` in R.

Value

A data frame. The rows are by default lexicographically sorted on the common columns, but for `sort = FALSE` are in an unspecified order. The columns are the common columns followed by the remaining columns in `x` and then those in `y`. If the matching involved row names, an extra character column called `Row.names` is added at the left, and in all cases the result has 'automatic' row names.

See Also

`data.frame`, `by`, `cbind`

Examples

```
## use character columns of names to get sensible sort order
authors <- data.frame(
    surname = I(c("Tukey", "Venables", "Tierney", "Ripley", "McNeil")),
    nationality = c("US", "Australia", "US", "UK", "Australia"),
    deceased = c("yes", rep("no", 4)))
books <- data.frame(
    name = I(c("Tukey", "Venables", "Tierney",
             "Ripley", "Ripley", "McNeil", "R Core")),
    title = c("Exploratory Data Analysis",
              "Modern Applied Statistics ...",
              "LISP-STAT",
              "Spatial Statistics", "Stochastic Simulation",
              "Interactive Data Analysis",
              "An Introduction to R"),
```

```
    other.author = c(NA, "Ripley", NA, NA, NA, NA,
                     "Venables & Smith"))

(m1 <- merge(authors, books, by.x = "surname", by.y = "name"))
(m2 <- merge(books, authors, by.x = "name", by.y = "surname"))
stopifnot(as.character(m1[,1]) == as.character(m2[,1]),
          all.equal(m1[, -1], m2[, -1][ names(m1)[-1] ]),
          dim(merge(m1, m2, by = integer(0))) == c(36, 10))

## "R core" is missing from authors and appears only here :
merge(authors, books, by.x = "surname", by.y = "name", all = TRUE)

## example of using 'incomparables'
x <- data.frame(k1=c(NA,NA,3,4,5), k2=c(1,NA,NA,4,5), data=1:5)
y <- data.frame(k1=c(NA,2,NA,4,5), k2=c(NA,NA,3,4,5), data=1:5)
merge(x, y, by=c("k1","k2")) # NA's match
merge(x, y, by=c("k1","k2"), incomparables=NA)
merge(x, y, by="k1") # NA's match, so 6 rows
merge(x, y, by="k2", incomparables=NA) # 2 rows
```

`message` *Diagnostic Messages*

Description

Generate a diagnostic message from its arguments.

Usage

```
message(..., domain = NULL, appendLF = TRUE)
suppressMessages(expr)

packageStartupMessage(..., domain = NULL, appendLF = TRUE)
suppressPackageStartupMessages(expr)

.makeMessage(..., domain = NULL, appendLF = FALSE)
```

Arguments

`...`	zero or more objects which can be coerced to character (and which are pasted together with no separator) or (for `message` only) a single condition object.
`domain`	see `gettext`. If `NA`, messages will not be translated.
`appendLF`	logical: should messages given as a character string have a newline appended?
`expr`	expression to evaluate.

Details

`message` is used for generating 'simple' diagnostic messages which are neither warnings nor errors, but nevertheless represented as conditions. Unlike warnings and errors, a final newline is regarded as part of the message, and is optional. The default handler sends the message to the `stderr()` connection.

If a condition object is supplied to `message` it should be the only argument, and further arguments will be ignored, with a warning.

While the message is being processed, a `muffleMessage` restart is available.

`suppressMessages` evaluates its expression in a context that ignores all 'simple' diagnostic messages.

`packageStartupMessage` is a variant whose messages can be suppressed separately by `suppressPackageStartupMessages`. (They are still messages, so can be suppressed by `suppressMessages`.)

`.makeMessage` is a utility used by `message`, `warning` and `stop` to generate a text message from the `...` arguments by possible translation (see `gettext`) and concatenation (with no separator).

See Also

`warning` and `stop` for generating warnings and errors; `conditions` for condition handling and recovery.

`gettext` for the mechanisms for the automated translation of text.

Examples

```
message("ABC", "DEF")
suppressMessages(message("ABC"))

testit <- function() {
  message("testing package startup messages")
  packageStartupMessage("initializing ...", appendLF = FALSE)
  Sys.sleep(1)
  packageStartupMessage(" done")
}

testit()
suppressPackageStartupMessages(testit())
suppressMessages(testit())
```

`missing` *Does a Formal Argument have a Value?*

Description

`missing` can be used to test whether a value was specified as an argument to a function.

Usage

```
missing(x)
```

Arguments

`x`	a formal argument.

Details

`missing(x)` is only reliable if `x` has not been altered since entering the function: in particular it will *always* be false after `x <- match.arg(x)`.

The example shows how a plotting function can be written to work with either a pair of vectors giving x and y coordinates of points to be plotted or a single vector giving y values to be plotted against their indexes.

Currently `missing` can only be used in the immediate body of the function that defines the argument, not in the body of a nested function or a `local` call. This may change in the future.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`substitute` for argument expression; `NA` for missing values in data.

Examples

```
myplot <- function(x,y) {
                if(missing(y)) {
                        y <- x
                        x <- 1:length(y)
                }
                plot(x,y)
        }
```

---

mode *The (Storage) Mode of an Object*

Description

Get or set the type or storage mode of an object.

Usage

```
mode(x)
mode(x) <- value
storage.mode(x)
storage.mode(x) <- value
```

Arguments

`x`	any R object.
`value`	a character string giving the desired mode or 'storage mode' (type) of the object.

Details

Both `mode` and `storage.mode` return a character string giving the (storage) mode of the object — often the same — both relying on the output of `typeof(x)`, see the example below.

`mode(x) <- "newmode"` changes the `mode` of object `x` to `newmode`. This is only supported if there is an appropriate `as.newmode` function, for example `"logical"`, `"integer"`, `"double"`, `"complex"`, `"raw"`, `"character"`, `"list"`, `"expression"`, `"name"`, `"symbol"` and `"function"`. Attributes are preserved (but see below).

`storage.mode(x) <- "newmode"` is a more efficient internal version of `mode<-`, which works for `"newmode"` which is one of the internal types (see `typeof`), but not for `"single"`. Attributes are preserved.

As storage mode `"single"` is only a pseudo-mode in R, it will not be reported by `mode` or `storage.mode`: use `attr(object, "Csingle")` to examine this. However, `mode<-` can be used to set the mode to `"single"`, which sets the real mode to `"double"` and the `"Csingle"` attribute to `TRUE`. Setting any other mode will remove this attribute.

Note (in the examples below) that some `call`s have mode `"("` which is S compatible.

Mode names

Modes have the same set of names as types (see `typeof`) except that

- types `"integer"` and `"double"` are returned as `"numeric"`.
- types `"special"` and `"builtin"` are returned as `"function"`.
- type `"symbol"` is called mode `"name"`.
- type `"language"` is returned as `"("` or `"call"`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`typeof` for the R-internal 'mode', `attributes`.

Examples

```
require(stats)

sapply(options(),mode)

cex3 <- c("NULL","1","1:1","1i","list(1)","data.frame(x=1)",
  "pairlist(pi)", "c", "lm", "formals(lm)[[1]]",  "formals(lm)[[2]]",
  "y~x","expression((1))[[1]]", "(y~x)[[1]]",
  "expression(x <- pi)[[1]][[1]]")
lex3 <- sapply(cex3, function(x) eval(parse(text=x)))
mex3 <- t(sapply(lex3,
                 function(x) c(typeof(x), storage.mode(x), mode(x))))
dimnames(mex3) <- list(cex3, c("typeof(.)","storage.mode(.)","mode(.)"))
mex3

## This also makes a local copy of 'pi':
storage.mode(pi) <- "complex"
```

```
storage.mode(pi)
rm(pi)
```

NA *Not Available / "Missing" Values*

Description

NA is a logical constant of length 1 which contains a missing value indicator. NA can be freely coerced to any other vector type except raw. There are also constants NA_integer_, NA_real_, NA_complex_ and NA_character_ of the other atomic vector types which support missing values: all of these are reserved words in the R language.

The generic function is.na indicates which elements are missing.

The generic function is.na<- sets elements to NA.

Usage

```
NA
is.na(x)
## S3 method for class 'data.frame':
is.na(x)

is.na(x) <- value
```

Arguments

x	an R object to be tested.
value	a suitable index vector for use with x.

Details

The NA of character type is distinct from the string "NA". Programmers who need to specify an explicit string NA should use NA_character_ rather than "NA", or set elements to NA using is.na<-.

is.na(x) works elementwise when x is a list. It is generic: you can write methods to handle specific classes of objects, see InternalMethods. A complex value is regarded as NA if either its real or imaginary part is NA or NaN.

Function is.na<- may provide a safer way to set missingness. It behaves differently for factors, for example.

Value

The default method for is.na returns a logical vector of the same length as its argument x, containing TRUE for those elements marked NA or NaN (!) and FALSE otherwise. dim, dimnames and names attributes are preserved.

The method is.na.data.frame returns a logical matrix with the same dimensions as the data frame, and with dimnames taken from the row and column names of the data frame.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`NaN`, `is.nan`, etc., and the utility function `complete.cases`.

`na.action`, `na.omit`, `na.fail` on how methods can be tuned to deal with missing values.

Examples

```
is.na(c(1, NA))         #> FALSE  TRUE
is.na(paste(c(1, NA))) #> FALSE FALSE

(xx <- c(0:4))
is.na(xx) <- c(2, 4)
xx                      #> 0 NA  2 NA  4
```

`name` *Names and Symbols*

Description

A 'name' (also known as a 'symbol') is a way to refer to R objects by name (rather than the value of the object, if any, bound to that name).

`as.name` and `as.symbol` are identical: they attempt to coerce the argument to a name.

`is.symbol` and the identical `is.name` return `TRUE` or `FALSE` depending on whether the argument is a name or not.

Usage

```
as.symbol(x)
is.symbol(x)

as.name(x)
is.name(x)
```

Arguments

`x`	object to be coerced or tested.

Details

`as.name` first coerces its argument internally to a character vector (so methods for `as.character` are not used). It then takes the first element and provided it is not `""`, returns a symbol of that name (and if the element is `NA_character_`, the name is `'NA'`.

`as.name` is implemented as `as.vector(x, "symbol")`, and hence will dispatch methods for the generic function `as.vector`.

Value

For `as.name` and `as.symbol`, an R object of type `"symbol"` (see `typeof`).

For `is.name` and `is.symbol`, a length-one logical vector with value `TRUE` or `FALSE`.

Note

The term 'symbol' is from the LISP background of R, whereas 'name' has been the standard S term for this.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`call`, `is.language`. For the internal object mode, `typeof`.

`plotmath` for another use of 'symbol'.

Examples

```
an <- as.name("arrg")
is.name(an) # TRUE
mode(an)    # name
typeof(an) # symbol
```

`names` *The Names of an Object*

Description

Functions to get or set the names of an object.

Usage

```
names(x)
names(x) <- value
```

Arguments

`x`	an R object.
`value`	a character vector of up to the same length as `x`, or `NULL`.

Details

`names` is a generic accessor function, and `names<-` is a generic replacement function. The default methods get and set the `"names"` attribute of a vector (including a list) or pairlist.

If `value` is shorter than `x`, it is extended by character `NA`s to the length of `x`.

It is possible to update just part of the names attribute via the general rules: see the examples. This works because the expression there is evaluated as `z <- "names<-"(z, "[<-"(names(z), 3, "c2"))`.

The name `""` is special: it is used to indicate that there is no name associated with an element of a (atomic or generic) vector. Subscripting by `""` will match nothing (not even elements which have no name).

A name can be character `NA`, but such a name will never be matched and is likely to lead to confusion.

Value

For `names`, `NULL` or a character vector of the same length as `x`. (`NULL` is given if the object has no names, including for objects of types which cannot have names.)

For `names<-`, the updated object. (Note that the value of `names(x) <- value` is that of the assignment, `value`, not the return value from the left-hand side.)

Note

For vectors, the names are one of the attributes with restrictions on the possible values. For pairlists, the names are the tags and converted to and from a character vector.

For a one-dimensional array the `names` attribute really is `dimnames[[1]]`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
# print the names attribute of the islands data set
names(islands)

# remove the names attribute
names(islands) <- NULL
islands
rm(islands) # remove the copy made

z <- list(a=1, b="c", c=1:3)
names(z)
# change just the name of the third element.
names(z)[3] <- "c2"
z

z <- 1:3
names(z)
## assign just one name
names(z)[2] <- "b"
z
```

nargs *The Number of Arguments to a Function*

Description

When used inside a function body, `nargs` returns the number of arguments supplied to that function, *including* positional arguments left blank.

Usage

```
nargs()
```

Details

The count includes empty (missing) arguments, so that `foo(x,,z)` will be considered to have three arguments (see 'Examples'). This can occur in rather indirect ways, so for example `x[]` might dispatch a call to `` `[.some_method`(x, ) `` which is considered to have two arguments.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`args`, `formals` and `sys.call`.

Examples

```
tst <- function(a, b = 3, ...) {nargs()}
tst() # 0
tst(clicketyclack) # 1 (even non-existing)
tst(c1, a2, rr3) # 3

foo <- function(x, y, z, w) {
   cat("call was", deparse(match.call()), "\n")
   nargs()
}
foo()    # 0
foo(,,3) # 3
foo(z=3) # 1, even though this is the same call

nargs()# not really meaningful
```

nchar *Count the Number of Characters (or Bytes or Width)*

Description

nchar takes a character vector as an argument and returns a vector whose elements contain the sizes of the corresponding elements of x.

nzchar is a fast way to find out if elements of a character vector are non-empty strings.

Usage

```
nchar(x, type = "chars", allowNA = FALSE)

nzchar(x)
```

Arguments

x	character vector, or a vector to be coerced to a character vector.
type	character string: partial matching to one of c("bytes", "chars", "width"). See 'Details'.
allowNA	logical: show NA be returned for invalid multibyte strings (rather than throwing an error)?

Details

The 'size' of a character string can be measured in one of three ways

bytes The number of bytes needed to store the string (plus in C a final terminator which is not counted).

chars The number of human-readable characters.

width The number of columns cat will use to print the string in a monospaced font. The same as chars if this cannot be calculated.

These will often be the same, and almost always will be in single-byte locales. There will be differences between the first two with multibyte character sequences, e.g. in UTF-8 locales.

The internal equivalent of the default method of as.character is performed on x (so there is no method dispatch). If you want to operate on non-vector objects passing them through deparse first will be required.

Value

For nchar, an integer vector giving the sizes of each element, currently always 2 for missing values (for NA).

If allowNA = TRUE and an element is invalid in a multi-byte character set such as UTF-8, its number of characters and the width will be NA. Otherwise the number of characters will be non-negative, so !is.na(nchar(x, "chars", TRUE)) is a test of validity.

Names, dims and dimnames are copied from the input.

For nzchar, a logical vector of the same length as x, true if and only if the element has non-zero length.

Note

This does **not** by default give the number of characters that will be used to `print()` the string. Use `encodeString` to find the characters used to print the string. Where character strings have been marked as UTF-8, the number of characters and widths will be computed in UTF-8, even though printing may use escapes such as `<U+2642>` in a non-UTF-8 locale.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`strwidth` giving width of strings for plotting; `paste`, `substr`, `strsplit`

Examples

```
x <- c("asfef", "qwerty", "yuiop[", "b", "stuff.blah.yech")
nchar(x)
# 5  6  6  1 15

nchar(deparse(mean))
# 18 17
```

---

nlevels *The Number of Levels of a Factor*

Description

Return the number of levels which its argument has.

Usage

```
nlevels(x)
```

Arguments

`x`	an object, usually a factor.

Details

This is usually applied to a factor, but other objects can have levels.

The actual factor levels (if they exist) can be obtained with the `levels` function.

Value

The length of `levels(x)`, which is zero if `x` has no levels.

See Also

`levels`, `factor`.

Examples

```
nlevels(gl(3,7)) # = 3
```

`noquote` *Class for "no quote" Printing of Character Strings*

Description

Print character strings without quotes.

Usage

```
noquote(obj)

## S3 method for class 'noquote':
print(x, ...)

## S3 method for class 'noquote':
c(..., recursive = FALSE)
```

Arguments

`obj`	any R object, typically a vector of `character` strings.
`x`	an object of class `"noquote"`.
`...`	further options passed to next methods, such as `print`.
`recursive`	for compatibility with the generic `c` function.

Details

`noquote` returns its argument as an object of class `"noquote"`. There is a method for `c()` and subscript method (`"[.noquote"`) which ensures that the class is not lost by subsetting. The print method (`print.noquote`) prints character strings *without* quotes (`"..."`).

These functions exist both as utilities and as an example of using (S3) `class` and object orientation.

Author(s)

Martin Maechler ⟨maechler@stat.math.ethz.ch⟩

See Also

`methods`, `class`, `print`.

Examples

```
letters
nql <- noquote(letters)
nql
nql[1:4] <- "oh"
nql[1:12]

cmp.logical <- function(log.v)
```

```
{
  ## Purpose: compact printing of logicals
  log.v <- as.logical(log.v)
  noquote(if(length(log.v)==0)"()" else c(".","|")[1+log.v])
}
cmp.logical(stats::runif(20) > 0.8)
```

NotYet *Not Yet Implemented Functions and Unused Arguments*

Description

In order to pinpoint missing functionality, the R core team uses these functions for missing R functions and not yet used arguments of existing R functions (which are typically there for compatibility purposes).

You are very welcome to contribute your code ...

Usage

```
.NotYetImplemented()
.NotYetUsed(arg, error = TRUE)
```

Arguments

arg	an argument of a function that is not yet used.
error	a logical. If TRUE, an error is signalled; if FALSE; only a warning is given.

See Also

the contrary, Deprecated and Defunct for outdated code.

Examples

```
require(graphics)
require(stats)
plot.mlm         # to see how the "NotYetImplemented"
                 # reference is made automagically
try(plot.mlm())

barplot(1:5, inside = TRUE) # 'inside' is not yet used
```

`nrow` *The Number of Rows/Columns of an Array*

Description

`nrow` and `ncol` return the number of rows or columns present in `x`. `NCOL` and `NROW` do the same treating a vector as 1-column matrix.

Usage

```
nrow(x)
ncol(x)
NCOL(x)
NROW(x)
```

Arguments

`x`	a vector, array or data frame

Value

an `integer` of length 1 or `NULL`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (`ncol` and `nrow`.)

See Also

`dim` which returns *all* dimensions; `array`, `matrix`.

Examples

```
ma <- matrix(1:12, 3, 4)
nrow(ma)   # 3
ncol(ma)   # 4

ncol(array(1:24, dim = 2:4)) # 3, the second dimension
NCOL(1:12) # 1
NROW(1:12) # 12
```

ns-dblcolon *Double Colon and Triple Colon Operators*

Description

Accessing exported and internal variables in a name space.

Usage

```
pkg::name
pkg:::name
```

Arguments

pkg	package name symbol or literal character string.
name	variable name symbol or literal character string.

Details

The expression `pkg::name` returns the value of the exported variable `name` in package `pkg` if the package has a name space. The expression `pkg:::name` returns the value of the internal variable `name` in package `pkg` if the package has a name space. The package will be loaded if it was not loaded already before the call. Assignment into name spaces is not supported.

Note that it is typically a design mistake to use `:::` in your code since the corresponding object has probably been kept namespace-internal for a good reason. Consider contacting the package maintainer if you feel the need to access the object for anything but mere inspection.

If the package `pkg` does not have a name space but is on the search path then `pkg::name` returns the value of `name` in the package environment.

See Also

`get` to access an object masked by another of the same name.

Examples

```
base::log
base::"+"

## Beware --  use ':::' at your own risk! (see "Details")
stats:::coef.default
```

`ns-hooks` *Hooks for Name Space events*

Description

Packages with name spaces can supply functions to be called when loaded, attached or unloaded.

Usage

```
.onLoad(libname, pkgname)
.onAttach(libname, pkgname)
.onUnload(libpath)
```

Arguments

`libname`	a character string giving the library directory where the package defining the namespace was found.
`pkgname`	a character string giving the name of the package, including the version number if the package was installed with `--with-package-versions` (deprecated).
`libpath`	a character string giving the complete path to the package.

Details

These functions apply only to packages with name spaces.

After loading, `loadNamespace` looks for a hook function named `.onLoad` and runs it before sealing the namespace and processing exports.

If a name space is unloaded (via `unloadNamespace`), a hook function `.onUnload` is run before final unloading.

Note that the code in `.onLoad` and `.onUnload` is run without the package being on the search path, but (unless circumvented) lexical scope will make objects in the namespace and its imports visible. (Do not use the double colon operator in `.onLoad` as exports have not been processed at the point it is run.)

When the package is attached (via `library`), the hook function `.onAttach` is looked for and if found is called after the exported functions are attached and before the package environment is sealed. This is less likely to be useful than `.onLoad`, which should be seen as the analogue of `.First.lib` (which is only used for packages without a name space).

`.onLoad`, `.onUnload` and `.onAttach` are looked for as internal variables in the name space and should not be exported.

If a function `.Last.lib` is visible in the package, it will be called when the package is detached: this does need to be exported.

Anything needed for the functioning of the name space should be handled at load/unload times by the `.onLoad` and `.onUnload` hooks. For example, DLLs can be loaded (unless done by a `useDynLib` directive in the 'NAMESPACE' file) and initialized in `.onLoad` and unloaded in `.onUnload`. Use `.onAttach` only for actions that are needed only when the package becomes visible to the user, for example a start-up message.

If a package was installed with `--with-package-versions`, the `pkgname` supplied will be something like `tree_1.0-16`.

See Also

`setHook` shows how users can set hooks on the same events.

`ns-load` *Loading and Unloading Name Spaces*

Description

Functions to load and unload namespaces.

Usage

```
attachNamespace(ns, pos = 2, dataPath = NULL)
loadNamespace(package, lib.loc = NULL,
              keep.source = getOption("keep.source.pkgs"),
              partial = FALSE, declarativeOnly = FALSE)
loadedNamespaces()
unloadNamespace(ns)
```

Arguments

`ns`	string or namespace object.
`pos`	integer specifying position to attach.
`dataPath`	path to directory containing a database of datasets to be lazy-loaded into the attached environment.
`package`	string naming the package/name space to load.
`lib.loc`	character vector specifying library search path.
`keep.source`	logical specifying whether to retain source. This applies only to the specified name space, and not to other name spaces which might be loaded to satisfy imports. For more details see this argument to `library`.
`partial`	logical; if true, stop just after loading code.
`declarativeOnly`	logical; disables `.Import`, etc, if true.

Details

The functions `loadNamespace` and `attachNamespace` are usually called implicitly when `library` is used to load a name space and any imports needed. However it may be useful to call these functions directly at times.

`loadNamespace` loads the specified name space and registers it in an internal data base. A request to load a name space that is already loaded has no effect. The arguments have the same meaning as the corresponding arguments to `library`, whose help page explains the details of how a particular installed package comes to be chosen. After loading, `loadNamespace` looks for a hook function named `.onLoad` as an internal variable in the name space (it should not be exported). This function is called with the same arguments as `.First.lib`. Partial loading is used to support installation with the '`--save`' and '`--lazy`' options.

loadNamespace does not attach the name space it loads to the search path. attachNamespace can be used to attach a frame containing the exported values of a name space to the search path. The hook function .onAttach is run after the name space exports are attached.

loadedNamespaces returns a character vector of the names of the loaded name spaces.

unloadNamespace can be used to force a name space to be unloaded. An error is signaled if the name space is imported by other loaded name spaces. If defined, a hook function .onUnload is run before removing the name space from the internal registry. unloadNamespace will first detach a package of the same name if one is on the path, thereby running a .Last.lib function in the package if one is exported.

Author(s)

Luke Tierney

---

ns-topenv *Top Level Environment*

Description

Finding the top level environment.

Usage

```
topenv(envir = parent.frame(),
       matchThisEnv = getOption("topLevelEnvironment"))
```

Arguments

envir	environment.
matchThisEnv	return this environment, if it matches before any other criterion is satisfied. The default, the option 'topLevelEnvironment', is set by sys.source, which treats a specific environment as the top level environment. Supplying the argument as NULL means it will never match.

Details

topenv returns the first top level environment found when searching envir and its parent environments. An environment is considered top level if it is the internal environment of a name space, a package environment in the search path, or .GlobalEnv.

Examples

```
topenv(.GlobalEnv)
topenv(new.env())
```

NULL *The Null Object*

Description

`NULL` represents the null object in R: it is a reserved word. NULL is often returned by expressions and functions whose value is undefined: it is also used as the empty pairlist.

`as.null` ignores its argument and returns the value `NULL`.

`is.null` returns `TRUE` if its argument is `NULL` and `FALSE` otherwise.

Usage

```
NULL
as.null(x, ...)
is.null(x)
```

Arguments

`x`	an object to be tested or coerced.
`...`	ignored.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
is.null(list())    # FALSE (on purpose!)
is.null(integer(0))# F
is.null(logical(0))# F
as.null(list(a=1,b='c'))
```

numeric *Numeric Vectors*

Description

Creates or coerces objects of type `"numeric"`. `is.numeric` is a more general test of an object being interpretable as numbers.

Usage

```
numeric(length = 0)
as.numeric(x, ...)
is.numeric(x)
```

Arguments

`length`	desired length.
`x`	object to be coerced or tested.
`...`	further arguments passed to or from other methods.

Details

`numeric` is identical to `double` (and `real`). It creates a double-precision vector of the specified length with each element equal to `0`.

`as.numeric` is a generic function, but S3 methods must be written for `as.double`. It is identical to `as.double` (and `as.real`).

`is.numeric` is generic: you can write methods to handle specific classes of objects, see Internal-Methods. It is **not** the same as `is.double`. Factors are handled by the default method, and there are methods for classes `"Date"` and `"POSIXt"` (in all three cases the result is false). Methods for `is.numeric` should only return true if the base type of the class is `double` or `integer` *and* values can reasonably be regarded as numeric (e.g. arithmetic on them makes sense).

Value

for `numeric` and `as.numeric` see `double`.

The default method for `is.numeric` returns `TRUE` if its argument is of mode `"numeric"` (type `"double"` or type `"integer"`) and not a factor, and `FALSE` otherwise. That is, `is.integer(x) || is.double(x)`, or `(mode(x) == "numeric") && !is.factor(x)`.

S4 methods

`as.numeric` and `is.numeric` are internally S4 generic and so methods can be set for them *via* `setMethod`.

To ensure that `as.numeric`, `as.double` and `as.real` remain identical, S4 methods can only be set for `as.numeric`.

Note on names

It is a historical anomaly that R has three names for its floating-point vectors, `double`, `numeric` and `real`.

`double` is the name of the type. `numeric` is the name of the mode and also of the implicit class. As an S4 formal class, use `"numeric"` (there was a formal class `"double"` prior to R 2.7.0).

`real` is deprecated and should not be used in new code.

The potential confusion is that R has used *mode* `"numeric"` to mean 'double or integer', which conflicts with the S4 usage. Thus `is.numeric` tests the mode, not the class, but `as.numeric` (which is identical to `as.double`) coerces to the class.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`double`, `integer`, `storage.mode`.

Examples

```
as.numeric(c("-.1"," 2.7 ","B")) # (-0.1, 2.7, NA)  +  warning
as.numeric(factor(5:10))
```

`NumericConstants` *Numeric Constants*

Description

How R parses numeric constants.

Details

R parses numeric constants in its input in a very similar way to C99 floating-point constants.

`Inf` and `NaN` are numeric constants (with `typeof(.)` `"double"`). These are recognized ignoring case, as is `infinity` as an alternative to `Inf`. `NA_real_` and `NA_integer_` are constants of types `"double"` and `"integer"` representing missing values. All other numeric constants start with a digit or period and are either a decimal or hexadecimal constant optionally followed by `L`.

Hexadecimal constants start with `0x` or `0X` followed by a nonempty sequence from `0-9 a-f A-F .` which is interpreted as a hexadecimal number, optionally followed by a binary exponent. A binary exponent consists of an `P` or `p` followed by an optional plus or minus sign followed by a non-empty sequence of (decimal) digits, and indicates multiplication by a power of two. Thus `0x123p456` is 291×2^{456}.

Decimal constants consist of a nonempty sequence of digits possibly containing a period (the decimal point), optionally followed by a decimal exponent. A decimal exponent consists of an `E` or `e` followed by an optional plus or minus sign followed by a non-empty sequence of digits, and indicates multiplication by a power of ten.

Values which are too large or too small to be representable will overflow to `Inf` or underflow to `0.0`.

A numeric constant immediately followed by `i` is regarded as an imaginary complex number.

An numeric constant immediately followed by `L` is regarded as an `integer` number when possible (and with a warning if it contains a `"."`).

Only the ASCII digits 0–9 are recognized as digits, even in languages which have other representations of digits. The 'decimal separator' is always a period and never a comma.

Note that a leading plus or minus is not regarded by the parser as part of a numeric constant but as a unary operator applied to the constant.

See Also

`Syntax`.

`Quotes` for the parsing of character constants,

Examples

```
2.1
typeof(2)
sqrt(1i) # remember elementary math?
utils::str(0xA0)
identical(1L, as.integer(1))

## You can combine the "0x" prefix with the "L" suffix :
identical(0xFL, as.integer(15)) # (with a regard to Fritz :-)
```

---

numeric_version *Numeric Versions*

Description

A simple S3 class for representing numeric versions including package versions, and associated methods.

Usage

```
numeric_version(x, strict = TRUE)
package_version(x, strict = TRUE)
R_system_version(x, strict = TRUE)
getRversion()
```

Arguments

x	a character vector with suitable numeric version strings (see 'Details'); for package_version, alternatively an R version object as obtained by R.version.
strict	a logical indicating whether invalid numeric versions should results in an error (default) or not.

Details

Numeric versions are sequences of one or more non-negative integers, usually (e.g., in package 'DESCRIPTION' files) represented as character strings with the elements of the sequence concatenated and separated by single '.' or '-' characters. R package versions consist of at least two such integers, an R system version of exactly three (major, minor and patchlevel).

Functions numeric_version, package_version and R_system_version create a representation from such strings (if suitable) which allows for coercion and testing, combination, comparison, summaries (min/max), inclusion in data frames, subscripting, and printing. The classes can hold a vector of such representations.

getRversion returns the version of the running R as an R system version object.

The [[operator extracts or replaces a single version. To access the integers of a version use two indices: see the examples.

See Also

compareVersion

Examples

```
x <- package_version(c("1.2-4", "1.2-3", "2.1"))
x < "1.4-2.3"
c(min(x), max(x))
x[2, 2]
x$major
x$minor

if(getRversion() <= "2.5.0") { ## work around missing feature
  cat("Your version of R, ", as.character(getRversion()),
      ", is outdated.\n",
      "Now trying to work around that ...\n", sep = "")
}

x[[c(1,3)]]  # '4' as a numeric vector, same as x[1, 3]
x[1, 3]      # 4 as an integer
x[[2, 3]] <- 0   # zero the patchlevel
x[[c(2,3)]] <- 0 # same
x
x[[3]] <- "2.2.3"; x
```

---

octmode *Display Numbers in Octal*

Description

Convert or print integers in octal format, with as many digits as are needed to display the largest, using leading zeroes as necessary.

Usage

```
as.octmode(x)

## S3 method for class 'octmode':
as.character(x, ...)

## S3 method for class 'octmode':
format(x, ...)

## S3 method for class 'octmode':
print(x, ...)
```

Arguments

x	An object, for the methods inheriting from class "octmode".
...	further arguments passed to or from other methods.

Details

Class "octmode" consists of integer vectors with that class attribute, used merely to ensure that they are printed in octal notation, specifically for Unix-like file permissions such as 755. Subsetting ([) works too.

as.octmode can convert integers (of type "integer" or "double") and character strings to class "octmode"

See Also

These are auxiliary functions for file.info.

hexmode

Examples

```
(on <- structure(c(16,32, 127:129),  class = "octmode")) #-> print.*()
##-> "020" "040" "177" "200" "201"
unclass(on[3:4]) # subsetting
```

---

on.exit *Function Exit Code*

Description

on.exit records the expression given as its argument as needing to be executed when the current function exits (either naturally or as the result of an error). This is useful for resetting graphical parameters or performing other cleanup actions.

If no expression is provided, i.e., the call is on.exit(), then the current on.exit code is removed.

on.exit is a primitive function so positional matching is used and names of supplied arguments are ignored.

Usage

```
on.exit(expr, add = FALSE)
```

Arguments

expr	an expression to be executed.
add	if TRUE, add expr to be executed after any previously set expressions; otherwise (the default) expr will overwrite any previously set expressions.

Details

Where expr was evaluated changed in R 2.8.0, and the following applies only to that and later versions.

The expr argument passed to on.exit is recorded without evaluation. If it is not subsequently removed/replaced by another on.exit call in the same function, it is evaluated in the evaluation frame of the function when it exits (including during standard error handling). Thus any functions or variables in the expression will be looked for in the function and its environment at the time of exit: to capture the current value in expr use substitute or similar.

Value

Invisible NULL.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`sys.on.exit` which returns the expression stored for use by `on.exit()` in the function in which `sys.on.exit()` is evaluated.

Examples

```
require(graphics)

opar <- par(mai = c(1,1,1,1))
on.exit(par(opar))
```

`Ops.Date` *Operators on the Date Class*

Description

Operators for the "`Date`" class.

There is an `Ops` method and specific methods for + and − for the `Date` class.

Usage

```
date + x
date - x
date1 lop date2
```

Arguments

`date`	date objects
`date1, date2`	date objects or character vectors. (Character vectors are converted by `as.Date`.)
`x`	a numeric vector (in days) *or* an object of class "`difftime`".
`lop`	One of ==, !=, <, <=, > or >=.

Examples

```
(z <- Sys.Date())
z + 10
z < c("2006-06-01", "2007-01-01", "2010-01-01")
```

`options` *Options Settings*

Description

Allow the user to set and examine a variety of global *options* which affect the way in which R computes and displays its results.

Usage

```
options(...)

getOption(x)

.Options
```

Arguments

`...`	any options can be defined, using `name = value` or by passing a list of such tagged values. However, only the ones below are used in base R. Further, `options('name') == options()['name']`, see the example.
`x`	a character string holding an option name.

Details

Invoking `options()` with no arguments returns a list with the current values of the options. Note that not all options listed below are set initially. To access the value of a single option, one should use `getOption("width")`, e.g., rather than `options("width")` which is a *list* of length one.

`.Options` also always contains the `options()` list (as a pairlist, unsorted), for S compatibility. Assigning to it will make a local copy and not change the original.

Value

For `getOption`, the current value set for option `x`, or `NULL` if the option is unset.

For `options()`, a list of all set options sorted by name. For `options(name)`, a list of length one containing the set value, or `NULL` if it is unset. For uses setting one or more options, a list with the previous values of the options changed (returned invisibly).

Options used in base R

`add.smooth`: typically logical, defaulting to `TRUE`. Could also be set to an integer for specifying how many (simulated) smooths should be added. This is currently only used by `plot.lm`.

`check.bounds`: logical, defaulting to `FALSE`. If true, a warning is produced whenever a vector (atomic or `list`) is extended, by something like `x <- 1:3; x[5] <- 6`.

`continue`: a non-empty string setting the prompt used for lines which continue over one line.

`defaultPackages`: the packages that are attached by default when R starts up. Initially set from value of the environment variable `R_DEFAULT_PACKAGES`, or if that is unset to `c("datasets", "utils", "grDevices", "graphics", "stats",`

"methods"). (Set R_DEFAULT_PACKAGES to NULL or a comma-separated list of package names.) A call to options should be in your '.Rprofile' file to ensure that the change takes effect before the base package is initialized (see Startup).

deparse.max.lines: controls the number of lines used when deparsing in traceback and browser. Initially unset, and only used if set to a positive integer.

digits: controls the number of digits to print when printing numeric values. It is a suggestion only. Valid values are 1...22 with default 7. See the warning in print.default about values greater than 15.

digits.secs: controls the maximum number of digits to print when formatting time values in seconds. Valid values are 0...6 with default 0. See strftime.

download.file.method: Method to be used for download.file. Currently download methods "internal", "wget" and "lynx" are available. There is no default for this option, when method = "auto" is chosen: see download.file.

echo: logical. Only used in non-interactive mode, when it controls whether input is echoed. Command-line option -slave sets this to FALSE, but otherwise it starts the session as TRUE.

encoding: The name of an encoding, default "native.enc"). See connections.

error: either a function or an expression governing the handling of non-catastrophic errors such as those generated by stop as well as by signals and internally detected errors. If the option is a function, a call to that function, with no arguments, is generated as the expression. The default value is NULL: see stop for the behaviour in that case. The functions dump.frames and recover provide alternatives that allow post-mortem debugging. Note that these need to specified as e.g. options=utils::recover in startup files such as '.Rprofile'.

expressions: sets a limit on the number of nested expressions that will be evaluated. Valid values are 25...500000 with default 5000. If you increase it, you may also want to start R with a larger protection stack; see -max-ppsize in Memory. Note too that you may cause a segfault from overflow of the C stack, and on OSes where it is possible you may want to increase that.

keep.source: When TRUE, the source code for functions (newly defined or loaded) is stored in their "source" attribute (see attr) allowing comments to be kept in the right places.

The default is interactive(), i.e., TRUE for interactive use.

keep.source.pkgs: As for keep.source, for functions in packages loaded by library or require. Defaults to FALSE unless the environment variable R_KEEP_PKG_SOURCE is set to yes.

Note this does not apply to packages using lazy-loading or saved images. Whether they have kept source is determined when they are installed (and is almost certainly false).

mailer: default mailer used by bug.report(). Can be "none".

max.contour.segments: positive integer, defaulting to 250000 and usually not set. A limit on the number of segments in a single contour line in contour or contourLines.

max.print: integer, defaulting to 99999. print or show methods can make use of this option, to limit the amount of information that is printed, to something in the order of (and typically slightly less than) max.print *entries*.

OutDec: one-character string. The character to be used as the decimal point in output conversions, that is in printing, plotting and as.character but not deparsing.

pager: the command used for displaying text files by file.show. Defaults to '*R_HOME*/bin/pager', which selects a pager via the PAGER environment variable (and that usually defaults to less). Can be a character string or an R function, in which case it needs to accept the same first four arguments as file.show.

papersize: the default paper format used by `postscript`; set by environment variable `R_PAPERSIZE` when R is started: if that is unset or invalid it defaults to a value derived from the locale category `LC_PAPER`, or if that is unavailable to a default set when R was built.

printcmd: the command used by `postscript` for printing; set by environment variable `R_PRINTCMD` when R is started. This should be a command that expects either input to be piped to 'stdin' or to be given a single filename argument.

prompt: a non-empty string to be used for R's prompt; should usually end in a blank (`"  "`).

rl_word_breaks: Used for the readline-based terminal interface. Default value `" \t\n\"\\'` `` `><=%;,|&{()}" ``. This is the set of characters use to break the input line up into tokens for object- and file-name completion. Those who do not use spaces around operators may prefer `" \t\n\"\\'` `` `><=+-*%;,|&{()}" ``. which was the default in R 2.5.0. (The default in pre-2.5.0 versions of R was `" \t\n\"\\'` `` `@$><=;|&{(" ``.)

save.defaults, save.image.defaults: see `save`.

scipen: integer. A penalty to be applied when deciding to print numeric values in fixed or exponential notation. Positive values bias towards fixed and negative towards scientific notation: fixed notation will be preferred unless it is more than `scipen` digits wider.

showWarnCalls, showErrorCalls: a logical. Should warning and error messages show a summary of the call stack? By default error calls are shown in non-interactive sessions.

showNCalls: a integer. Controls how long the sequence of calls must be (in bytes) before ellipses are used. Defaults to 40 and should be at least 30 and no more than 500.

show.error.messages: a logical. Should error messages be printed? Intended for use with `try` or a user-installed error handler.

stringsAsFactors: The default setting for arguments of `data.frame` and `read.table`.

texi2dvi: used by function `texi2dvi` in package **tools**.

timeout: integer. The timeout for some Internet operations, in seconds. Default 60 seconds. See `download.file` and `connections`.

topLevelEnvironment: see `topenv` and `sys.source`.

useFancyQuotes: see `sQuote`.

verbose: logical. Should R report extra information on progress? Set to `TRUE` by the command-line option '`-verbose`'.

warn: sets the handling of warning messages. If `warn` is negative all warnings are ignored. If `warn` is zero (the default) warnings are stored until the top–level function returns. If fewer than 10 warnings were signalled they will be printed otherwise a message saying how many (max 50) were signalled. An object called `last.warning` is created and can be printed through the function `warnings`. If `warn` is one, warnings are printed as they occur. If `warn` is two or larger all warnings are turned into errors.

warnEscapes: logical. If true (the default) the parser warns on unrecognized (backslash) escape sequences and there is a truncation warning on any attempt to create a character string containing an embedded nul.

warnPartialMatchArgs: logical. If true, warns if partial matching is used in argument matching.

warnPartialMatchAttr: logical. If true, warns if partial matching is used in extracting attributes via `attr`.

warnPartialMatchDollar: logical. If true, warns if partial matching is used for extraction by `$`.

warning.expression: an R code expression to be called if a warning is generated, replacing the standard message. If non-null it is called irrespective of the value of option `warn`.

warning.length: sets the truncation limit for error and warning messages. A non-negative integer, with allowed values 100...8170, default 1000.

width: controls the maximum number of columns on a line used in printing vectors, matrices and arrays, and when filling by `cat`.

Columns are normally the same as characters except in CJK languages.

You may want to change this if you re-size the window that R is running in. Valid values are 10...10000 with default normally 80. (The limits on valid values are in file 'Print.h' and can be changed by re-compiling R.) Some R consoles automatically change the value when they are resized.

The 'factory-fresh' default settings of some of these options are

```
add.smooth            TRUE
check.bounds          FALSE
continue              "+ "
digits                7
echo                  TRUE
encoding              "native.enc"
error                 NULL
expressions           5000
keep.source           interactive()
keep.source.pkgs      FALSE
max.print             99999
OutDec                "."
prompt                "> "
scipen                0
show.error.messages   TRUE
timeout               60
verbose               FALSE
warn                  0
warnings.length       1000
width                 80
```

Others are set from environment variables or are platform-dependent.

Options set in package grDevices

These will be set when package **grDevices** (or its name space) is loaded if not already set.

device: a character string giving the name of a function, or the function object itself, which when called creates a new graphics device of the default type for that session. The value of this option defaults to the normal screen device (e.g., `X11`, `windows` or `quartz`) for an interactive session, and `pdf` in batch use or if a screen is not available. If set to the name of a device, the device is looked for first from the global environment (that is down the usual search path) and then in the **grDevices** namespace.

The default values in interactive and non-interactive sessions are configurable via environment variables `R_INTERACTIVE_DEVICE` and `R_DEFAULT_DEVICE` respectively.

locatorBell: logical. Should selection in `locator` and `identify` be confirmed by a bell? Default `TRUE`. Honoured at least on `X11` and `windows` devices.

device.ask.default: logical. The default for devAskNewPage("ask") when a device is opened.

bitmapType: character. The default type for the bitmap devices such as png. Defaults to "cairo" on systems where that is available, or to "quartz" on Mac OS X where that is available.

Options set in package stats

These will be set when package **stats** (or its name space) is loaded if not already set.

contrasts: the default contrasts used in model fitting such as with aov or lm. A character vector of length two, the first giving the function to be used with unordered factors and the second the function to be used with ordered factors. By default the elements are named c("unordered", "ordered"), but the names are unused.

na.action: the name of a function for treating missing values (NA's) for certain situations.

show.coef.Pvalues: logical, affecting whether P values are printed in summary tables of coefficients. See printCoefmat.

show.signif.stars: logical, should stars be printed on summary tables of coefficients? See printCoefmat.

ts.eps: the relative tolerance for certain time series (ts) computations. Default 1e-05.

ts.S.compat: logical. Used to select S compatibility for plotting time-series spectra. See the description of argument log in plot.spec.

Options set in package utils

These will be set when package **utils** (or its name space) is loaded if not already set.

browser: default HTML browser used by help.start() and browseURL on UNIX, or a non-default browser on Windows. Alternatively, an R function that is called with a URL as its argument.

de.cellwidth: integer: the cell widths (number of characters) to be used in the data editor dataentry. If this is unset (the default), 0, negative or NA, variable cell widths are used.

editor: a non-empty string, or a function that is called with a file path as argument. Sets the default text editor, e.g., for edit. Set from the environment variable VISUAL on UNIX.

example.ask: default for the ask argument of example.

help.try.all.packages: default for an argument of help.

HTTPUserAgent: string used as the user agent in HTTP requests. If NULL, HTTP requests will be made without a user agent header. The default is R (<version> <platform> <arch> <os>)

internet.info: The minimum level of information to be printed on URL downloads etc. Default is 2, for failure causes. Set to 1 or 0 to get more information.

menu.graphics: Logical: should graphical menus be used if available?. Defaults to TRUE. Currently applies to chooseCRANmirror, setRepositories and to select from multiple help files in help.

pkgType: The default type of packages to be downloaded and installed – see install.packages. Possible values are "source" (the default except under the CRAN Mac OS X build) and "mac.binary". The latter can have a suffix if supported by a special build, such as "mac.binary.leopard" to access the "leopard" tree of repositories instead of the default "universal".

repos: URLs of the repositories for use by `update.packages`. Defaults to `c(CRAN="@CRAN@")`, a value that causes some utilities to prompt for a CRAN mirror. To avoid this do set the CRAN mirror, by something like `local({r <- getOption("repos"); r["CRAN"] <- "http://my.local.cran"; options(repos=r)})`.

Note that you can add more repositories (Bioconductor and Omegahat, notably) using `setRepositories()`.

SweaveHooks, SweaveSyntax: see `Sweave`.

unzip: a character string, the path of the command used for unzipping help files, or `"internal"`. Defaults to the value of `R_UNZIPCMD`, which is set in 'etc/Renviron' if an `unzip` command was found during configuration.

Options used on Unix only

These will be `""` if no suitable path is known.

latexcmd, dvipscmd: character strings giving commands to be used in off-line printing of help pages.

pdfviewer: default PDF viewer. Set from the environment variable `R_PDFVIEWER`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
op <- options(); utils::str(op) # op() may contain functions.

getOption("width") == options()$width # the latter needs more memory
options(digits = 15)
pi

# set the editor, and save previous value
old.o <- options(editor = "nedit")
old.o

options(check.bounds = TRUE, warn = 1)
x <- NULL; x[4] <- "yes" # gives a warning

options(digits=5)
print(1e5)
options(scipen=3); print(1e5)

options(op)     # reset (all) initial options
options("digits")

## Not run:
## set contrast handling to be like S
options(contrasts = c("contr.helmert", "contr.poly"))
## End(Not run)

## Not run:
## on error, terminate the R session with error status 66
options(error = quote(q("no", status=66, runLast=FALSE)))
```

```
stop("test it")
## End(Not run)

## Not run:
## Set error actions for debugging:
## enter browser on error, see ?recover:
options(error = recover)
## allows to call debugger() afterwards, see ?debugger:
options(error = dump.frames)
## A possible setting for non-interactive sessions
options(error = quote({dump.frames(to.file=TRUE); q()}))
## End(Not run)
```

`order` *Ordering Permutation*

Description

`order` returns a permutation which rearranges its first argument into ascending or descending order, breaking ties by further arguments. `sort.list` is the same, using only one argument. See the examples for how to use these functions to sort data frames, etc.

Usage

```
order(..., na.last = TRUE, decreasing = FALSE)

sort.list(x, partial = NULL, na.last = TRUE, decreasing = FALSE,
          method = c("shell", "quick", "radix"))
```

Arguments

`...`	a sequence of numeric, complex, character or logical vectors, all of the same length.
`x`	a vector.
`partial`	vector of indices for partial sorting. (Non-`NULL` values are not implemented.)
`decreasing`	logical. Should the sort order be increasing or decreasing?
`na.last`	for controlling the treatment of `NA`s. If `TRUE`, missing values in the data are put last; if `FALSE`, they are put first; if `NA`, they are removed.
`method`	the method to be used: partial matches are allowed.

Details

In the case of ties in the first vector, values in the second are used to break the ties. If the values are still tied, values in the later arguments are used to break the tie (see the first example). The sort used is *stable* (except for `method = "quick"`), so any unresolved ties will be left in their original ordering.

Complex values are sorted first by the real part, then the imaginary part.

The sort order for character vectors will depend on the collating sequence of the locale in use: see `Comparison`.

The default method for `sort.list` is a good compromise. Method `"quick"` is only supported for numeric `x` with `na.last=NA`, and is not stable, but will be faster for long vectors. Method `"radix"` is only implemented for integer `x` with a range of less than 100,000. For such `x` it is very fast (and stable), and hence is ideal for sorting factors.

`partial` is supplied for compatibility with other implementations of S, but no other values are accepted and ordering is always complete.

Note that these functions are only defined for vectors, so any class of the object supplied is ignored: this means factors are sorted on their internal codes and not their printed representation.

Note

`sort.list` can get called by mistake as a method for `sort` with a list argument, and gives a suitable error message for list `x`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`sort`, `rank`, `xtfrm`.

Examples

```
require(stats)

(ii <- order(x <- c(1,1,3:1,1:4,3), y <- c(9,9:1), z <-c(2,1:9)))
## 6  5  2  1  7  4 10  8  3  9
rbind(x,y,z)[,ii] # shows the reordering (ties via 2nd & 3rd arg)

## Suppose we wanted descending order on y.
## A simple solution for numeric 'y' is
rbind(x,y,z)[, order(x, -y, z)]
## More generally we can make use of xtfrm
cy <- as.character(y)
rbind(x,y,z)[, order(x, -xtfrm(cy), z)]

## Sorting data frames:
dd <- transform(data.frame(x,y,z),
                z = factor(z, labels=LETTERS[9:1]))
## Either as above {for factor 'z' : using internal coding}:
dd[ order(x, -y, z) ,]
## or along 1st column, ties along 2nd, ... *arbitrary* no.{columns}:
dd[ do.call(order, dd) ,]

set.seed(1)# reproducible example:
d4 <- data.frame(x = round(   rnorm(100)), y = round(10*runif(100)),
                 z = round( 8*rnorm(100)), u = round(50*runif(100)))
(d4s <- d4[ do.call(order, d4) ,])
(i <- which(diff(d4s[,3]) == 0))
#   in 2 places, needed 3 cols to break ties:
d4s[ rbind(i,i+1), ]

## rearrange matched vectors so that the first is in ascending order
```

```
x <- c(5:1, 6:8, 12:9)
y <- (x - 5)^2
o <- order(x)
rbind(x[o], y[o])

## tests of na.last
a <- c(4, 3, 2, NA, 1)
b <- c(4, NA, 2, 7, 1)
z <- cbind(a, b)
(o <- order(a, b)); z[o, ]
(o <- order(a, b, na.last = FALSE)); z[o, ]
(o <- order(a, b, na.last = NA)); z[o, ]

## Not run:
##  speed examples for long vectors:
x <- factor(sample(letters, 1e6, replace=TRUE))
system.time(o <- sort.list(x)) ## 1.2 secs
stopifnot(!is.unsorted(x[o]))
system.time(o <- sort.list(x, method="quick", na.last=NA)) # 0.15 sec
stopifnot(!is.unsorted(x[o]))
system.time(o <- sort.list(x, method="radix")) # 0.02 sec
stopifnot(!is.unsorted(x[o]))
xx <- sample(1:26, 1e7, replace=TRUE)
system.time(o <- sort.list(xx, method="radix")) # 0.2 sec
xx <- sample(1:100000, 1e7, replace=TRUE)
system.time(o <- sort.list(xx, method="radix")) # 0.8 sec
system.time(o <- sort.list(xx, method="quick", na.last=NA)) # 1.4 sec
## End(Not run)
```

`outer` *Outer Product of Arrays*

Description

The outer product of the arrays `X` and `Y` is the array `A` with dimension `c(dim(X), dim(Y))` where element `A[c(arrayindex.x, arrayindex.y)] = FUN(X[arrayindex.x], Y[arrayindex.y], ...)`.

Usage

```
outer(X, Y, FUN="*", ...)
X %o% Y
```

Arguments

`X, Y`	First and second arguments for function `FUN`. Typically a vector or array.
`FUN`	a function to use on the outer products, found *via* `match.fun` (except for the special case `"*"`).
`...`	optional arguments to be passed to `FUN`.

Details

X and Y must be suitable arguments for FUN. Each will be extended by rep to length the products of the lengths of X and Y before FUN is called.

FUN is called with these two extended vectors as arguments. Therefore, it must be a vectorized function (or the name of one), expecting at least two arguments.

Where they exist, the [dim]names of X and Y will be copied to the answer, and a dimension assigned which is the concatenation of the dimensions of X and Y (or lengths if dimensions do not exist).

FUN = "*" is handled internally as a special case, *via* as.vector(X) %*% t(as.vector(Y)), and is intended only for numeric vectors and arrays.

%o% is binary operator providing a wrapper for outer(x, y, "*").

Author(s)

Jonathan Rougier

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

%*% for usual (*inner*) matrix vector multiplication; kronecker which is based on outer; Vectorize for vectorizing a non-vectorized function.

Examples

```
x <- 1:9; names(x) <- x
# Multiplication & Power Tables
x %o% x
y <- 2:8; names(y) <- paste(y,":",sep="")
outer(y, x, "^")

outer(month.abb, 1999:2003, FUN = "paste")

## three way multiplication table:
x %o% x %o% y[1:3]
```

Paren *Parentheses and Braces*

Description

Open parenthesis, (, and open brace, {, are .Primitive functions in R.

Effectively, (is semantically equivalent to the identity function(x) x, whereas { is slightly more interesting, see examples.

Usage

```
( ... )

{ ... }
```

Value

For `(`, the result of evaluating the argument. This has visibility set, so will auto-print if used at top-level.

For `{`, the result of the last expression evaluated. This has the visibility of the last evaluation.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`if`, `return`, etc for other objects used in the R language itself.

`Syntax` for operator precedence.

Examples

```
f <- get("(")
e <- expression(3 + 2 * 4)
identical(f(e), e)

do <- get("{")
do(x <- 3, y <- 2*x-3, 6-x-y); x; y

## note the differences
(2+3)
{2+3; 4+5}
(invisible(2+3))
{invisible(2+3)}
```

`parse` *Parse Expressions*

Description

`parse` returns the parsed but unevaluated expressions in a list.

Usage

```
parse(file = "", n = NULL, text = NULL, prompt = "?", srcfile,
      encoding = "unknown")
```

Arguments

`file`	a connection, or a character string giving the name of a file or a URL to read the expressions from. If `file` is `""` and `text` is missing or `NULL` then input is taken from the console.
`n`	integer (or coerced to integer). The maximum number of expressions to parse. If `n` is `NULL` or negative or `NA` the input is parsed in its entirety.
`text`	character vector. The text to parse. Elements are treated as if they were lines of a file. Other R objects will be coerced to character (without method dispatch) if possible.

prompt	the prompt to print when parsing from the keyboard. NULL means to use R's prompt, getOption("prompt").
srcfile	NULL, or a srcfile object. See the 'Details' section.
encoding	encoding to be assumed for input strings. If the value is "latin1" or "UTF-8" it is used to mark character strings as known to be in Latin-1 or UTF-8: it is not used to re-encode the input. To do the latter, specify the encoding as part of the connection con or *via* options(encoding=): see the example under file.

Details

If text has length greater than zero (after coercion) it is used in preference to file.

All versions of R accept input from a connection with end of line marked by LF (as used on Unix), CRLF (as used on DOS/Windows) or CR (as used on classic MacOS). The final line can be incomplete, that is missing the final EOL marker.

See source for the limits on the size of functions that can be parsed (by default).

When input is taken from the console, n = NULL is equivalent to n = 1, and n < 0 will read until an EOF character is read.

The default for srcfile is set as follows. If options("keep.source") is FALSE, srcfile defaults to NULL. Otherwise, if text is used, srcfile will be set to a srcfilecopy containing the text. If a character string is used for file, a srcfile object referring to that file will be used.

Value

An object of type "expression", with up to n elements if specified as a non-negative integer.

When srcfile is non-NULL, a "srcref" attribute will be attached to the result containing a list of srcref records corresponding to each element, and a "srcfile" attribute will be attached containing a copy of srcfile.

A syntax error (including an incomplete expression) will throw an error.

Character strings in the result will have a declared encoding if encoding is "latin1" or "UTF-8".

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

scan, source, eval, deparse.

Examples

```
cat("x <- c(1,4)\n  x ^ 3 -10 ; outer(1:7,5:9)\n", file="xyz.Rdmped")
# parse 3 statements from the file "xyz.Rdmped"
parse(file = "xyz.Rdmped", n = 3)
unlink("xyz.Rdmped")
```

paste *Concatenate Strings*

Description

Concatenate vectors after converting to character.

Usage

```
paste(..., sep = " ", collapse = NULL)
```

Arguments

...	one or more R objects, to be converted to character vectors.
sep	a character string to separate the terms. Not NA_character_.
collapse	an optional character string to separate the results. Not NA_character_.

Details

paste converts its arguments (*via* as.character) to character strings, and concatenates them (separating them by the string given by sep). If the arguments are vectors, they are concatenated term-by-term to give a character vector result. Vector arguments are recycled as needed, with zero-length arguments being recycled to "".

Note that paste() coerces NA_character_, the character missing value, to "NA" which may seem undesirable, e.g., when pasting two character vectors, or very desirable, e.g. in paste("the value of p is ", p).

If a value is specified for collapse, the values in the result are then concatenated into a single string, with the elements being separated by the value of collapse.

Value

A character vector of the concatenated values. This will be of length zero if all the objects are, unless collapse is non-NULL in which case it is a single empty string.

If any input into an element of the result is in UTF-8, that element will be in UTF-8, otherwise in the current encoding in which case the encoding of an element of the element is declared if the current locale is either Latin-1 or UTF-8, at least one of the corresponding inputs (including separators) had a declared encoding and all inputs were either ASCII or declared.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

String manipulation with as.character, substr, nchar, strsplit; further, cat which concatenates and writes to a file, and sprintf for C like string construction.

'plotmath' for the use of paste in plot annotation.

Examples

```
paste(1:12) # same as as.character(1:12)
paste("A", 1:6, sep = "")
paste("Today is", date())
```

---

path.expand *Expand File Paths*

Description

Expand a path name, for example by replacing a leading tilde by the user's home directory (if defined on that platform).

Usage

```
path.expand(path)
```

Arguments

path	character vector containing one or more path names.

Details

On *some* Unix versions, a leading `~user` will expand to the home directory of `user`, but not on Unix versions without `readline` installed, nor if R is invoked with '`--no-readline`'.

See Also

basename

Examples

```
path.expand("~/foo")
```

---

pmatch *Partial String Matching*

Description

`pmatch` seeks matches for the elements of its first argument among those of its second.

Usage

```
pmatch(x, table, nomatch = NA_integer_, duplicates.ok = FALSE)
```

Arguments

x	the values to be matched: converted to a character vector by as.character.
table	the values to be matched against: converted to a character vector.
nomatch	the value to be returned at non-matching or multiply partially matching positions. Note that it is coerced to integer.
duplicates.ok	should elements be in table be used more than once?

Details

The behaviour differs by the value of duplicates.ok. Consider first the case if this is true. First exact matches are considered, and the positions of the first exact matches are recorded. Then unique partial matches are considered, and if found recorded. (A partial match occurs if the whole of the element of x matches the beginning of the element of table.) Finally, all remaining elements of x are regarded as unmatched. In addition, an empty string can match nothing, not even an exact match to an empty string. This is the appropriate behaviour for partial matching of character indices, for example.

If duplicates.ok is FALSE, values of table once matched are excluded from the search for subsequent matches. This behaviour is equivalent to the R algorithm for argument matching, except for the consideration of empty strings (which in argument matching are matched after exact and partial matching to any remaining arguments).

charmatch is similar to pmatch with duplicates.ok true, the differences being that it differentiates between no match and an ambiguous partial match, it does match empty strings, and it does not allow multiple exact matches.

NA values are treated as if they were the string constant "NA".

Value

An integer vector (possibly including NA if nomatch = NA) of the same length as x, giving the indices of the elements in table which matched, or nomatch.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

match, charmatch and match.arg, match.fun, match.call, for function argument matching etc., grep etc for more general (regexp) matching of strings.

Examples

```
pmatch("", "")                              # returns NA
pmatch("m",   c("mean", "median", "mode")) # returns NA
pmatch("med", c("mean", "median", "mode")) # returns 2

pmatch(c("", "ab", "ab"), c("abc", "ab"), dup=FALSE)
pmatch(c("", "ab", "ab"), c("abc", "ab"), dup=TRUE)
## compare
charmatch(c("", "ab", "ab"), c("abc", "ab"))
```

`polyroot` *Find Zeros of a Real or Complex Polynomial*

Description

Find zeros of a real or complex polynomial.

Usage

```
polyroot(z)
```

Arguments

`z`	the vector of polynomial coefficients in increasing order.

Details

A polynomial of degree $n - 1$,

$$p(x) = z_1 + z_2 x + \cdots + z_n x^{n-1}$$

is given by its coefficient vector `z[1:n]`. `polyroot` returns the $n - 1$ complex zeros of $p(x)$ using the Jenkins-Traub algorithm.

If the coefficient vector `z` has zeroes for the highest powers, these are discarded.

As from R 2.8.0 there is no maximum degree, but numerical stability may be an issue for all but low-degree polynomials.

Value

A complex vector of length $n - 1$, where n is the position of the largest non-zero element of `z`.

References

Jenkins and Traub (1972) TOMS Algorithm 419. *Comm. ACM*, **15**, 97–99.

See Also

`uniroot` for numerical root finding of arbitrary functions; `complex` and the `zero` example in the demos directory.

Examples

```
polyroot(c(1, 2, 1))
round(polyroot(choose(8, 0:8)), 11) # guess what!
for (n1 in 1:4) print(polyroot(1:n1), digits = 4)
polyroot(c(1, 2, 1, 0, 0)) # same as the first
```

pos.to.env *Convert Positions in the Search Path to Environments*

Description

Returns the environment at a specified position in the search path.

Usage

```
pos.to.env(x)
```

Arguments

x	an integer between 1 and `length(search())`, the length of the search path.

Details

Several R functions for manipulating objects in environments (such as `get` and `ls`) allow specifying environments via corresponding positions in the search path. `pos.to.env` is a convenience function for programmers which converts these positions to corresponding environments; users will typically have no need for it.

Examples

```
pos.to.env(1) # R_GlobalEnv
# the next returns the base environment
pos.to.env(length(search()))
```

pretty *Pretty Breakpoints*

Description

Compute a sequence of about `n+1` equally spaced 'round' values which cover the range of the values in `x`. The values are chosen so that they are 1, 2 or 5 times a power of 10.

Usage

```
pretty(x, n = 5, min.n = n %/% 3,  shrink.sml = 0.75,
       high.u.bias = 1.5, u5.bias = .5 + 1.5*high.u.bias,
       eps.correct = 0)
```

Arguments

x	an object coercible to numeric by `as.numeric`.
n	integer giving the *desired* number of intervals. Non-integer values are rounded down.
min.n	nonnegative integer giving the *minimal* number of intervals. If `min.n == 0`, `pretty(.)` may return a single value.

`shrink.sml` positive numeric by a which a default scale is shrunk in the case when `range(x)` is very small (usually 0).

`high.u.bias` non-negative numeric, typically > 1. The interval unit is determined as {1,2,5,10} times `b`, a power of 10. Larger `high.u.bias` values favor larger units.

`u5.bias` non-negative numeric multiplier favoring factor 5 over 2. Default and 'optimal': `u5.bias = .5 + 1.5*high.u.bias`.

`eps.correct` integer code, one of {0,1,2}. If non-0, an *epsilon correction* is made at the boundaries such that the result boundaries will be outside `range(x)`; in the *small* case, the correction is only done if `eps.correct >=2`.

Details

`pretty` ignores non-finite values in `x`.

Let `d <- max(x) - min(x)` ≥ 0. If `d` is not (very close) to 0, we let `c <- d/n`, otherwise more or less `c <- max(abs(range(x)))*shrink.sml / min.n`. Then, the *10 base* `b` is $10^{\lfloor \log_{10}(c) \rfloor}$ such that $b \leq c < 10b$.

Now determine the basic *unit* u as one of $\{1, 2, 5, 10\}b$, depending on $c/b \in [1, 10)$ and the two '*bias*' coefficients, $h =$`high.u.bias` and $f =$`u5.bias`.

.....

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`axTicks` for the computation of pretty axis tick locations in plots, particularly on the log scale.

Examples

```
pretty(1:15)      # 0  2  4  6  8 10 12 14 16
pretty(1:15, h=2)# 0  5 10 15
pretty(1:15, n=4)# 0  5 10 15
pretty(1:15 * 2) # 0  5 10 15 20 25 30
pretty(1:20)       # 0  5 10 15 20
pretty(1:20, n=2) # 0 10 20
pretty(1:20, n=10)# 0  2  4 ... 20

for(k in 5:11) {
  cat("k=",k,": "); print(diff(range(pretty(100 + c(0, pi*10^-k)))))}

##-- more bizarre, when  min(x) == max(x):
pretty(pi)

add.names <- function(v) { names(v) <- paste(v); v}
utils::str(lapply(add.names(-10:20), pretty))
utils::str(lapply(add.names(0:20),   pretty, min.n = 0))
sapply(    add.names(0:20),   pretty, min.n = 4)

pretty(1.234e100)
pretty(1001.1001)
```

```
pretty(1001.1001, shrink = .2)
for(k in -7:3)
  cat("shrink=", formatC(2^k, width=9),":",
      formatC(pretty(1001.1001, shrink.sml = 2^k), width=6),"\n")
```

`Primitive` *Call a "Primitive" Internal Function*

Description

`.Primitive` returns an entry point to a 'primitive' (internally implemented) function.

Usage

```
.Primitive(name)
```

Arguments

`name`	name of the R function.

Details

The advantage of `.Primitive` over `.Internal` functions is the potential efficiency of argument passing. However, this is done by ignoring argument names and using positional matching of arguments (unless arranged differently for specific primitives such as `rep`), so this is discouraged for functions of more than one argument.

All primitive functions are in the base name space.

See Also

`.Internal`.

Examples

```
mysqrt <- .Primitive("sqrt")
c
.Internal # this one *must* be primitive!
get("if") # just 'if' or 'print(if)' are not syntactically ok.
```

`print` *Print Values*

Description

`print` prints its argument and returns it *invisibly* (via `invisible(x)`). It is a generic function which means that new printing methods can be easily added for new `class`es.

Usage

```
print(x, ...)

## S3 method for class 'factor':
print(x, quote = FALSE, max.levels = NULL,
      width = getOption("width"), ...)

## S3 method for class 'table':
print(x, digits = getOption("digits"), quote = FALSE,
      na.print = "", zero.print = "0", justify = "none", ...)
```

Arguments

x	an object used to select a method.
...	further arguments passed to or from other methods.
quote	logical, indicating whether or not strings should be printed with surrounding quotes.
max.levels	integer, indicating how many levels should be printed for a factor; if 0, no extra "Levels" line will be printed. The default, NULL, entails choosing max.levels such that the levels print on one line of width width.
width	only used when max.levels is NULL, see above.
digits	minimal number of *significant* digits, see print.default.
na.print	character string (or NULL) indicating NA values in printed output, see print.default.
zero.print	character specifying how zeros (0) should be printed; for sparse tables, using "." can produce stronger results.
justify	character indicating if strings should left- or right-justified or left alone, passed to format.

Details

The default method, print.default has its own help page. Use methods("print") to get all the methods for the print generic.

print.factor allows some customization and is used for printing ordered factors as well.

print.table for printing tables allows other customization.

See noquote as an example of a class whose main purpose is a specific print method.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

The default method print.default, and help for the methods above; further options, noquote.

For more customizable (but cumbersome) printing, see cat, format or also write.

Examples

```
require(stats)

ts(1:20)#-- print is the "Default function" --> print.ts(.) is called
rr <- for(i in 1:3) print(1:i)
rr

## Printing of factors
attenu$station ## 117 levels -> 'max.levels' depending on width

## ordered factors: levels  "l1 < l2 < .."
esoph$agegp[1:12]
esoph$alcgp[1:12]

## Printing of sparse (contingency) tables
set.seed(521)
t1 <- round(abs(rt(200, df=1.8)))
t2 <- round(abs(rt(200, df=1.4)))
table(t1,t2) # simple
print(table(t1,t2), zero.print = ".")# nicer to read
```

---

print.data.frame *Printing Data Frames*

Description

Print a data frame.

Usage

```
## S3 method for class 'data.frame':
print(x, ..., digits = NULL,
      quote = FALSE, right = TRUE, row.names = TRUE)
```

Arguments

x	object of class data.frame.
...	optional arguments to print or plot methods.
digits	the minimum number of significant digits to be used: see print.default.
quote	logical, indicating whether or not entries should be printed with surrounding quotes.
right	logical, indicating whether or not strings should be right-aligned. The default is right-alignment.
row.names	logical (or character vector), indicating whether (or what) row names should be printed.

Details

This calls format which formats the data frame column-by-column, then converts to a character matrix and dispatches to the print method for matrices.

When quote = TRUE only the entries are quoted not the row names nor the column names.

See Also

data.frame.

Examples

```
(dd <- data.frame(x=1:8, f=gl(2,4), ch=I(letters[1:8])))
     # print() with defaults
print(dd, quote = TRUE, row.names = FALSE)
     # suppresses row.names and quotes all entries
```

---

print.default *Default Printing*

Description

print.default is the *default* method of the generic print function which prints its argument.

Usage

```
## Default S3 method:
print(x, digits = NULL, quote = TRUE,
      na.print = NULL, print.gap = NULL, right = FALSE,
      max = NULL, useSource = TRUE, ...)
```

Arguments

x	the object to be printed.
digits	a non-null value for digits specifies the minimum number of significant digits to be printed in values. The default, NULL, uses getOption(digits). (For the interpretation for complex numbers see signif.) Non-integer values will be rounded down, and only values greater than or equal to 1 and no greater than 22 are accepted.
quote	logical, indicating whether or not strings (characters) should be printed with surrounding quotes.
na.print	a character string which is used to indicate NA values in printed output, or NULL (see 'Details').
print.gap	a non-negative integer $\leq$ 1024, or NULL (meaning 1), giving the spacing between adjacent columns in printed vectors, matrices and arrays.
right	logical, indicating whether or not strings should be right aligned. The default is left alignment.
max	a non-null value for max specifies the approximate maximum number of entries to be printed. The default, NULL, uses getOption(max.print); see that help page for more details.
useSource	logical, indicating whether to use source references or copies rather than deparsing language objects. The default is to use the original source if it is available.
...	further arguments to be passed to or from other methods. They are ignored in this function.

Details

The default for printing NAs is to print NA (without quotes) unless this is a character NA *and* quote = FALSE, when <NA> is printed.

The same number of decimal places is used throughout a vector. This means that digits specifies the minimum number of significant digits to be used, and that at least one entry will be encoded with that minimum number. However, if all the encoded elements then have trailing zeroes, the number of decimal places is reduced until at least one element has a non-zero final digit. Decimal points are only included if at least one decimal place is selected.

Attributes are printed respecting their class(es), using the values of digits to print.default, but using the default values (for the methods called) of the other arguments.

When the **methods** package is attached, print will call show for R objects with formal classes if called with no optional arguments.

Warning

Using too large a value of digits may lead to representation errors in the calculation of the number of significant digits and the decimal representation: these are likely for digits >= 16, and these possible errors are taken into account in assessing the numher of significant digits to be printed in that case.

Whereas earlier versions of R might have printed further digits for digits >= 16 on some platforms, they were not necessarily reliable.

Single-byte locales

If a non-printable character is encountered during output, it is represented as one of the ANSI escape sequences (\a, \b, \f, \n, \r, \t, \v, \\ and \0: see Quotes), or failing that as a 3-digit octal code: for example the UK currency pound sign in the C locale (if implemented correctly) is printed as \243. Which characters are non-printable depends on the locale.

Unicode and other multi-byte locales

In all locales, the characters in the ASCII range (0x00 to 0x7f) are printed in the same way, as-is if printable, otherwise via ANSI escape sequences or 3-digit octal escapes as described for single-byte locales.

Multi-byte non-printing characters are printed as an escape sequence of the form \uxxxx or \Uxxxxxxxx (in hexadecimal). This is the internal code for the wide-character representation of the character. If this is not known to be the Unicode point, a warning is issued. The only known exceptions are certain Japanese ISO2022 locales on commercial Unixes, which use a concatenation of the bytes: it is unlikely that R compiles on such a system.

It is possible to have a character string in a character vector that is not valid in the current locale. If a byte is encountered that is not part of a valid character it is printed in hex in the form \xab and this is repeated until the start of a valid character. (This will rapidly recover from minor errors in UTF-8.)

See Also

The generic print, options. The "noquote" class and print method.

encodeString, which encodes a character vector the way it would be printed.

Examples

```
pi
print(pi, digits = 16)
LETTERS[1:16]
print(LETTERS, quote = FALSE)

M <- cbind(I = 1, matrix(1:10000, ncol = 10,
                         dimnames = list(NULL, LETTERS[1:10])))
utils::head(M) # makes more sense than
print(M, max = 1000)# prints 90 rows and a message about omitting 910
```

---

prmatrix *Print Matrices, Old-style*

Description

An earlier method for printing matrices, provided for S compatibility.

Usage

```
prmatrix(x, rowlab =, collab =,
         quote = TRUE, right = FALSE, na.print = NULL, ...)
```

Arguments

x	numeric or character matrix.
rowlab,collab	(optional) character vectors giving row or column names respectively. By default, these are taken from `dimnames(x)`.
quote	logical; if `TRUE` and `x` is of mode `"character"`, *quotes* (`"`) are used.
right	if `TRUE` and `x` is of mode `"character"`, the output columns are *right*-justified.
na.print	how `NA`s are printed. If this is non-null, its value is used to represent `NA`.
...	arguments for `print` methods.

Details

`prmatrix` is an earlier form of `print.matrix`, and is very similar to the S function of the same name.

Value

Invisibly returns its argument, `x`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`print.default`, and other `print` methods.

Examples

```
prmatrix(m6 <- diag(6), rowlab = rep("",6), collab =rep("",6))

chm <- matrix(scan(system.file("help", "AnIndex", package = "splines"),
                   what = ""), , 2, byrow = TRUE)
chm  # uses print.matrix()
prmatrix(chm, collab = paste("Column",1:3), right=TRUE, quote=FALSE)
```

---

proc.time *Running Time of R*

Description

`proc.time` determines how much real and CPU time (in seconds) the currently running R process has already taken.

Usage

```
proc.time()
```

Details

`proc.time` returns five elements for backwards compatibility, but its `print` method prints a named vector of length 3. The first two entries are the total user and system CPU times of the current R process and any child processes on which it has waited, and the third entry is the 'real' elapsed time since the process was started.

Value

An object of class `"proc_time"` which is a numeric vector of length 5, containing the user, system, and total elapsed times for the currently running R process, and the cumulative sum of user and system times of any child processes spawned by it on which it has waited. (The `print` method combines the child times with those of the main process.)

The definition of 'user' and 'system' times is from your OS. Typically it is something like

The 'user time' is the CPU time charged for the execution of user instructions of the calling process. The 'system time' is the CPU time charged for execution by the system on behalf of the calling process.

The resolution of the times will be system-specific and on Unix-alikes times are rounded to the nearest 1ms. On modern systems they will be that accurate, but on older systems they might be accurate to 1/100 or 1/60 sec.

Note

It is possible to compile R without support for `proc.time`, when the function will throw an error.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

system.time for timing a valid R expression, gc.time for how much of the time was spent in garbage collection.

Examples

```
## Not run:
## a way to time an R expression: system.time is preferred
ptm <- proc.time()
for (i in 1:50) mad(stats::runif(500))
proc.time() - ptm
## End(Not run)
```

---

prod *Product of Vector Elements*

Description

prod returns the product of all the values present in its arguments.

Usage

```
prod(..., na.rm = FALSE)
```

Arguments

...	numeric or complex or logical vectors.
na.rm	logical. Should missing values be removed?

Details

If na.rm is FALSE an NA value in any of the arguments will cause a value of NA to be returned, otherwise NA values are ignored.

This is a generic function: methods can be defined for it directly or via the Summary group generic. For this to work properly, the arguments ... should be unnamed, and dispatch is on the first argument.

Logical true values are regarded as one, false values as zero. For historical reasons, NULL is accepted and treated as if it were numeric(0).

Value

The product, a numeric (of type "double") or complex vector of length one. **NB:** the product of an empty set is one, by definition.

S4 methods

This is part of the S4 Summary group generic. Methods for it must use the signature x, ..., na.rm.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`sum`, `cumprod`, `cumsum`.

'plotmath' for the use of `prod` in plot annotation.

Examples

```
print(prod(1:7)) == print(gamma(8))
```

---

prop.table *Express Table Entries as Fraction of Marginal Table*

Description

This is really `sweep(x, margin, margin.table(x, margin), "/")` for newbies, except that if `margin` has length zero, then one gets `x/sum(x)`.

Usage

```
prop.table(x, margin=NULL)
```

Arguments

`x`	table
`margin`	index, or vector of indices to generate margin for

Value

Table like `x` expressed relative to `margin`

Author(s)

Peter Dalgaard

See Also

`margin.table`

Examples

```
m <- matrix(1:4,2)
m
prop.table(m,1)
```

pushBack *Push Text Back on to a Connection*

Description

Functions to push back text lines onto a connection, and to enquire how many lines are currently pushed back.

Usage

```
pushBack(data, connection, newLine = TRUE)
pushBackLength(connection)
```

Arguments

data	a character vector.
connection	A connection.
newLine	logical. If true, a newline is appended to each string pushed back.

Details

Several character strings can be pushed back on one or more occasions. The occasions form a stack, so the first line to be retrieved will be the first string from the last call to pushBack. Lines which are pushed back are read prior to the normal input from the connection, by the normal text-reading functions such as readLines and scan.

Pushback is only allowed for readable connections in text mode.

Not all uses of connections respect pushbacks, in particular the input connection is still wired directly, so for example parsing commands from the console and scan("") ignore pushbacks on stdin.

Value

pushBack returns nothing.

pushBackLength returns number of lines currently pushed back.

See Also

connections, readLines.

Examples

```
zz <- textConnection(LETTERS)
readLines(zz, 2)
pushBack(c("aa", "bb"), zz)
pushBackLength(zz)
readLines(zz, 1)
pushBackLength(zz)
readLines(zz, 1)
readLines(zz, 1)
close(zz)
```

qr *The QR Decomposition of a Matrix*

Description

qr computes the QR decomposition of a matrix. It provides an interface to the techniques used in the LINPACK routine DQRDC or the LAPACK routines DGEQP3 and (for complex matrices) ZGEQP3.

Usage

```
qr(x, ...)
## Default S3 method:
qr(x, tol = 1e-07 , LAPACK = FALSE, ...)

qr.coef(qr, y)
qr.qy(qr, y)
qr.qty(qr, y)
qr.resid(qr, y)
qr.fitted(qr, y, k = qr$rank)
qr.solve(a, b, tol = 1e-7)
## S3 method for class 'qr':
solve(a, b, ...)

is.qr(x)
as.qr(x)
```

Arguments

x	a matrix whose QR decomposition is to be computed.
tol	the tolerance for detecting linear dependencies in the columns of x. Only used if LAPACK is false and x is real.
qr	a QR decomposition of the type computed by qr.
y, b	a vector or matrix of right-hand sides of equations.
a	A QR decomposition or (qr.solve only) a rectangular matrix.
k	effective rank.
LAPACK	logical. For real x, if true use LAPACK otherwise use LINPACK.
...	further arguments passed to or from other methods

Details

The QR decomposition plays an important role in many statistical techniques. In particular it can be used to solve the equation $\boldsymbol{Ax} = \boldsymbol{b}$ for given matrix $\boldsymbol{A}$, and vector $\boldsymbol{b}$. It is useful for computing regression coefficients and in applying the Newton-Raphson algorithm.

The functions qr.coef, qr.resid, and qr.fitted return the coefficients, residuals and fitted values obtained when fitting y to the matrix with QR decomposition qr. (If pivoting is used, some of the coefficients will be NA.) qr.qy and qr.qty return Q %*% y and t(Q) %*% y, where Q is the (complete) $\boldsymbol{Q}$ matrix.

All the above functions keep dimnames (and names) of x and y if there are.

solve.qr is the method for solve for qr objects. qr.solve solves systems of equations via the QR decomposition: if a is a QR decomposition it is the same as solve.qr, but if a is a rectangular matrix the QR decomposition is computed first. Either will handle over- and under-determined systems, providing a least-squares fit if appropriate.

is.qr returns TRUE if x is a list with components named qr, rank and qraux and FALSE otherwise.

It is not possible to coerce objects to mode "qr". Objects either are QR decompositions or they are not.

Value

The QR decomposition of the matrix as computed by LINPACK or LAPACK. The components in the returned value correspond directly to the values returned by DQRDC/DGEQP3/ZGEQP3.

qr	a matrix with the same dimensions as x. The upper triangle contains the $\boldsymbol{R}$ of the decomposition and the lower triangle contains information on the $\boldsymbol{Q}$ of the decomposition (stored in compact form). Note that the storage used by DQRDC and DGEQP3 differs.
qraux	a vector of length ncol(x) which contains additional information on $\boldsymbol{Q}$.
rank	the rank of x as computed by the decomposition: always full rank in the LAPACK case.
pivot	information on the pivoting strategy used during the decomposition.

Non-complex QR objects computed by LAPACK have the attribute "useLAPACK" with value TRUE.

Note

To compute the determinant of a matrix (do you *really* need it?), the QR decomposition is much more efficient than using Eigen values (eigen). See det.

Using LAPACK (including in the complex case) uses column pivoting and does not attempt to detect rank-deficient matrices.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at http://www.netlib.org/lapack/lug/lapack_lug.html.

See Also

qr.Q, qr.R, qr.X for reconstruction of the matrices. lm.fit, lsfit, eigen, svd.

det (using qr) to compute the determinant of a matrix.

Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
h9 <- hilbert(9); h9
qr(h9)$rank           #--> only 7
qrh9 <- qr(h9, tol = 1e-10)
qrh9$rank             #--> 9
##-- Solve linear equation system  H %*% x = y :
y <- 1:9/10
x <- qr.solve(h9, y, tol = 1e-10) # or equivalently :
x <- qr.coef(qrh9, y) #-- is == but much better than
                      #-- solve(h9) %*% y
h9 %*% x              # = y

## overdetermined system
A <- matrix(runif(12), 4)
b <- 1:4
qr.solve(A, b) # or solve(qr(A), b)
solve(qr(A, LAPACK=TRUE), b)
# this is a least-squares solution, cf. lm(b ~ 0 + A)

## underdetermined system
A <- matrix(runif(12), 3)
b <- 1:3
qr.solve(A, b)
solve(qr(A, LAPACK=TRUE), b)
# solutions will have one zero, not necessarily the same one
```

QR.Auxiliaries *Reconstruct the Q, R, or X Matrices from a QR Object*

Description

Returns the original matrix from which the object was constructed or the components of the decomposition.

Usage

```
qr.X(qr, complete = FALSE, ncol =)
qr.Q(qr, complete = FALSE, Dvec =)
qr.R(qr, complete = FALSE)
```

Arguments

qr	object representing a QR decomposition. This will typically have come from a previous call to qr or lsfit.
complete	logical expression of length 1. Indicates whether an arbitrary orthogonal completion of the $\boldsymbol{Q}$ or $\boldsymbol{X}$ matrices is to be made, or whether the $\boldsymbol{R}$ matrix is to be completed by binding zero-value rows beneath the square upper triangle.
ncol	integer in the range 1:nrow(qr$qr). The number of columns to be in the reconstructed $\boldsymbol{X}$. The default when complete is FALSE is the first min(ncol(X), nrow(X)) columns of the original $\boldsymbol{X}$ from which the qr

object was constructed. The default when `complete` is `TRUE` is a square matrix with the original $\boldsymbol{X}$ in the first `ncol(X)` columns and an arbitrary orthogonal completion (unitary completion in the complex case) in the remaining columns.

`Dvec` vector (not matrix) of diagonal values. Each column of the returned $\boldsymbol{Q}$ will be multiplied by the corresponding diagonal value. Defaults to all `1`s.

Value

`qr.X` returns $\boldsymbol{X}$, the original matrix from which the qr object was constructed, provided `ncol(X) <= nrow(X)`. If `complete` is `TRUE` or the argument `ncol` is greater than `ncol(X)`, additional columns from an arbitrary orthogonal (unitary) completion of `X` are returned.

`qr.Q` returns part or all of **Q**, the order-nrow(X) orthogonal (unitary) transformation represented by `qr`. If `complete` is `TRUE`, **Q** has `nrow(X)` columns. If `complete` is `FALSE`, **Q** has `ncol(X)` columns. When `Dvec` is specified, each column of **Q** is multiplied by the corresponding value in `Dvec`.

`qr.R` returns **R**. The number of rows of **R** is either `nrow(X)` or `ncol(X)` (and may depend on whether `complete` is `TRUE` or `FALSE`.

See Also

`qr`, `qr.qy`.

Examples

```
p <- ncol(x <- LifeCycleSavings[,-1]) # not the 'sr'
qrstr <- qr(x)   # dim(x) == c(n,p)
qrstr $ rank # = 4 = p
Q <- qr.Q(qrstr) # dim(Q) == dim(x)
R <- qr.R(qrstr) # dim(R) == ncol(x)
X <- qr.X(qrstr) # X == x
range(X - as.matrix(x))# ~ < 6e-12
## X == Q %*% R if there has been no pivoting, as here.
Q %*% R
```

`quit` *Terminate an R Session*

Description

The function `quit` or its alias `q` terminate the current R session.

Usage

```
quit(save = "default", status = 0, runLast = TRUE)
   q(save = "default", status = 0, runLast = TRUE)
.Last <- function(x) { ..... }
```

Arguments

`save`	a character string indicating whether the environment (workspace) should be saved, one of `"no"`, `"yes"`, `"ask"` or `"default"`.
`status`	the (numerical) error status to be returned to the operating system, where relevant. Conventionally `0` indicates successful completion.
`runLast`	should `.Last()` be executed?

Details

`save` must be one of `"no"`, `"yes"`, `"ask"` or `"default"`. In the first case the workspace is not saved, in the second it is saved and in the third the user is prompted and can also decide *not* to quit. The default is to ask in interactive use but may be overridden by command-line arguments (which must be supplied in non-interactive use).

Immediately *before* terminating, the function `.Last()` is executed if it exists and `runLast` is true. If in interactive use there are errors in the `.Last` function, control will be returned to the command prompt, so do test the function thoroughly. There is a system analogue, `.Last.sys()`, which is run after `.Last()` if `runLast` is true.

Some error statuses are used by R itself. The default error handler for non-interactive use effectively calls `q("no", 1, FALSE)` and returns error code 1. Error status 2 is used for R 'suicide', that is a catastrophic failure, and other small numbers are used by specific ports for initialization failures. It is recommended that users choose statuses of 10 or more.

Valid values of `status` are system-dependent, but `0:255` are normally valid. (Many OSes will report the last byte of the value, that is report the number modulo 256. But not all.)

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`.First` for setting things on startup.

Examples

```
## Not run:
## Unix-flavour example
.Last <- function() {
  cat("Now sending PostScript graphics to the printer:\n")
  system("lpr Rplots.ps")
  cat("bye bye...\n")
}
quit("yes")
## End(Not run)
```

Quotes *Quotes*

Description

Descriptions of the various uses of quoting in R.

Details

Three types of quote are part of the syntax of R: single and double quotation marks and the backtick (or back quote, `). In addition, backslash is used for quoting the following characters inside character constants.

Character constants

Single and double quotes delimit character constants. They can be used interchangeably but double quotes are preferred (and character constants are printed using double quotes), so single quotes are normally only used to delimit character constants containing double quotes.

Backslash is used to start an escape sequence inside character constants. Unless specified in the following table, an escaped character is interpreted as the character itself. (Note that the parser will warn about most such uses, as they are most often erroneous, e.g. using `\.` where `\\.` was intended.)

Single quotes need to be escaped by backslash in single-quoted strings, and double quotes in double-quoted strings.

`\n`	newline
`\r`	carriage return
`\t`	tab
`\b`	backspace
`\a`	alert (bell)
`\f`	form feed
`\v`	vertical tab
`\\`	backslash `\`
`\nnn`	character with given octal code (1, 2 or 3 digits)
`\xnn`	character with given hex code (1 or 2 hex digits)
`\unnnn`	Unicode character with given code (1–4 hex digits)
`\Unnnnnnnn`	Unicode character with given code (1–8 hex digits)

The last two are only supported on versions of R built with MBCS support: they are an error on other versions. Alternative forms are `\u{nnnn}` and `\U{nnnnnnnn}`. All except the Unicode escape sequences are also supported when reading character strings by `scan` and `read.table` if `allowEscapes = TRUE`.

These forms will also be used by `print.default` when outputting non-printable characters (including backslash).

Note that as from R 2.8.0 embedded nuls are not allowed in character strings, so using escapes (such as `\0`) for a nul will result in the string being truncated at that point (usually with a warning).

Names and Identifiers

Identifiers consist of a sequence of letters, digits, the period (`.`) and the underscore. They must not start with a digit nor underscore, nor with a period followed by a digit. Reserved words are not valid identifiers.

The definition of a *letter* depends on the current locale, but only ASCII digits are considered to be digits.

Such identifiers are also known as *syntactic names* and may be used directly in R code. Almost always, other names can be used provided they are quoted. The preferred quote is the backtick (`` ` ``), and `deparse` will normally use it, but under many circumstances single or double quotes can be used (as a character constant will often be converted to a name). One place where backticks may be essential is to delimit variable names in formulae: see `formula`.

See Also

`Syntax` for other aspects of the syntax.

`sQuote` for quoting English text.

`shQuote` for quoting OS commands.

The *R Language Definition* manual.

`R.Version` *Version Information*

Description

`R.Version()` provides detailed information about the version of R running.

`R.version` is a variable (a `list`) holding this information (and `version` is a copy of it for S compatibility).

Usage

```
R.Version()
R.version
R.version.string
version
```

Value

`R.Version` returns a list with character-string components

`platform`	the platform for which R was built. A triplet of the form CPU-VENDOR-OS, as determined by the configure script. E.g, `"i586-unknown-linux"` or `"i386-pc-mingw32"`.
`arch`	the architecture (CPU) R was built on/for.
`os`	the underlying operating system
`system`	CPU and OS, separated by a comma.
`status`	the status of the version (e.g., `"Alpha"`)
`major`	the major version number

minor	the minor version number, including the patchlevel
year	the year the version was released
month	the month the version was released
day	the day the version was released
svn rev	the Subversion revision number, which should be either "unknown" or a single number. (A range of numbers or a number with 'M' or 'S' appended indicates inconsistencies in the sources used to build this version of R.)
language	always "R".
version.string	a character string concatenating some of the info above, useful for plotting, etc.

R.version and version are lists of class "simple.list" which has a print method.

Note

Do *not* use R.version$os to test the platform the code is running on: use .Platform$OS.type instead. Slightly different versions of the OS may report different values of R.version$os, as may different versions of R.

R.version.string is a copy of R.version$version.string for simplicity and backwards compatibility.

See Also

sessionInfo which provides additional information; getRversion typically used inside R code, .Platform.

Examples

```
require(graphics)

R.version$os # to check how lucky you are ...
plot(0) # any plot
mtext(R.version.string, side=1,line=4,adj=1)# a useful bottom-right note
```

Random *Random Number Generation*

Description

.Random.seed is an integer vector, containing the random number generator (RNG) **state** for random number generation in R. It can be saved and restored, but should not be altered by the user.

RNGkind is a more friendly interface to query or set the kind of RNG in use.

RNGversion can be used to set the random generators as they were in an earlier R version (for reproducibility).

set.seed is the recommended way to specify seeds.

Usage

```
.Random.seed <- c(rng.kind, n1, n2, ...)
save.seed <- .Random.seed

RNGkind(kind = NULL, normal.kind = NULL)
RNGversion(vstr)
set.seed(seed, kind = NULL, normal.kind = NULL)
```

Arguments

`kind`	character or `NULL`. If `kind` is a character string, set R's RNG to the kind desired. Use `"default"` to return to the R default. See 'Details' for the interpretation of `NULL`.
`normal.kind`	character string or `NULL`. If it is a character string, set the method of Normal generation. Use `"default"` to return to the R default. `NULL` makes no change.
`seed`	a single value, interpreted as an integer.
`vstr`	a character string containing a version number, e.g., `"1.6.2"`
`rng.kind`	integer code in `0:k` for the above `kind`.
`n1, n2, ...`	integers. See the details for how many are required (which depends on `rng.kind`).

Details

The currently available RNG kinds are given below. `kind` is partially matched to this list. The default is `"Mersenne-Twister"`.

`"Wichmann-Hill"` The seed, `.Random.seed[-1] == r[1:3]` is an integer vector of length 3, where each `r[i]` is in `1:(p[i] - 1)`, where `p` is the length 3 vector of primes, `p = (30269, 30307, 30323)`. The Wichmann–Hill generator has a cycle length of 6.9536×10^{12} (= `prod(p-1)/4`, see *Applied Statistics* (1984) **33**, 123 which corrects the original article).

`"Marsaglia-Multicarry"`: A *multiply-with-carry* RNG is used, as recommended by George Marsaglia in his post to the mailing list 'sci.stat.math'. It has a period of more than 2^{60} and has passed all tests (according to Marsaglia). The seed is two integers (all values allowed).

`"Super-Duper"`: Marsaglia's famous Super-Duper from the 70's. This is the original version which does *not* pass the MTUPLE test of the Diehard battery. It has a period of $\approx 4.6 \times 10^{18}$ for most initial seeds. The seed is two integers (all values allowed for the first seed: the second must be odd).

We use the implementation by Reeds et al. (1982–84).

The two seeds are the Tausworthe and congruence long integers, respectively. A one-to-one mapping to S's `.Random.seed[1:12]` is possible but we will not publish one, not least as this generator is **not** exactly the same as that in recent versions of S-PLUS.

`"Mersenne-Twister"`: From Matsumoto and Nishimura (1998). A twisted GFSR with period $2^{19937} - 1$ and equidistribution in 623 consecutive dimensions (over the whole period). The 'seed' is a 624-dimensional set of 32-bit integers plus a current position in that set.

`"Knuth-TAOCP-2002"`: A 32-bit integer GFSR using lagged Fibonacci sequences with subtraction. That is, the recurrence used is

$$X_j = (X_{j-100} - X_{j-37}) \bmod 2^{30}$$

and the 'seed' is the set of the 100 last numbers (actually recorded as 101 numbers, the last being a cyclic shift of the buffer). The period is around 2^{129}.

"Knuth-TAOCP": An earlier version from Knuth (1997).

The 2002 version was not backwards compatible with the earlier version: the initialization of the GFSR from the seed was altered. R did not allow you to choose consecutive seeds, the reported 'weakness', and already scrambled the seeds.

Initialization of this generator is done in interpreted R code and so takes a short but noticeable time.

"user-supplied": Use a user-supplied generator. See `Random.user` for details.

`normal.kind` can be `"Kinderman-Ramage"`, `"Buggy Kinderman-Ramage"` (not for `set.seed`), `"Ahrens-Dieter"`, `"Box-Muller"`, `"Inversion"` (the default), or `"user-supplied"`. (For inversion, see the reference in `qnorm`.) The Kinderman-Ramage generator used in versions prior to 1.7.1 (now called `"Buggy"` had several approximation errors and should only be used for reproduction of older results. The `"Box-Muller"` generator is stateful as pairs of normals are generated and returned sequentially. The state is reset whenever it is selected (even if it is the current normal generator) and when `kind` is changed.

`set.seed` uses its single integer argument to set as many seeds as are required. It is intended as a simple way to get quite different seeds by specifying small integer arguments, and also as a way to get valid seed sets for the more complicated methods (especially `"Mersenne-Twister"` and `"Knuth-TAOCP"`).

The use of `kind=NULL` or `normal.kind=NULL` in `RNGkind` or `set.seed` selects the currently-used generator (including as from R 2.8.0 that used in the previous session if the workspace has been restored): if no generator has been used it selects `"default"`.

Value

`.Random.seed` is an `integer` vector whose first element *codes* the kind of RNG and normal generator. The lowest two decimal digits are in `0:(k-1)` where `k` is the number of available RNGs. The hundreds represent the type of normal generator (starting at `0`).

In the underlying C, `.Random.seed[-1]` is `unsigned`; therefore in R `.Random.seed[-1]` can be negative, due to the representation of an unsigned integer by a signed integer.

`RNGkind` returns a two-element character vector of the RNG and normal kinds selected *before* the call, invisibly if either argument is not `NULL`. A type starts a session as the default, and is selected either by a call to `RNGkind` or by setting `.Random.seed` in the workspace.

`RNGversion` returns the same information as `RNGkind` about the defaults in a specific R version.

`set.seed` returns `NULL`, invisibly.

Note

Initially, there is no seed; a new one is created from the current time when one is required. Hence, different sessions started at (sufficiently) different times will give different simulation results, by default. However, the seed might be restored from a previous session if a previously saved workspace is restored.

`.Random.seed` saves the seed set for the uniform random-number generator, at least for the system generators. It does not necessarily save the state of other generators, and in particular does not save the state of the Box–Muller normal generator. If you want to reproduce work later, call `set.seed` (preferably with explicit values for `kind` and `normal.kind`) rather than set `.Random.seed`.

The object `.Random.seed` is only looked for in the user's workspace.

Do not rely on randomness of low-order bits from RNGs. Most of the supplied uniform generators return 32-bit integer values that are converted to doubles, so they take at most 2^{32} distinct values and long runs will return duplicated values (Wichmann-Hill is the exception, and all give at least 30 varying bits.)

Author(s)

of RNGkind: Martin Maechler. Current implementation, B. D. Ripley

References

Ahrens, J. H. and Dieter, U. (1973) Extensions of Forsythe's method for random sampling from the normal distribution. *Mathematics of Computation* **27**, 927-937.

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`set.seed`, storing in `.Random.seed`.)

Box, G. E. P. and Muller, M. E. (1958) A note on the generation of normal random deviates. *Annals of Mathmatical Statistics* **29**, 610–611.

De Matteis, A. and Pagnutti, S. (1993) *Long-range Correlation Analysis of the Wichmann-Hill Random Number Generator*, Statist. Comput., **3**, 67–70.

Kinderman, A. J. and Ramage, J. G. (1976) Computer generation of normal random variables. *Journal of the American Statistical Association* **71**, 893-896.

Knuth, D. E. (1997) *The Art of Computer Programming.* Volume 2, third edition.
Source code at http://www-cs-faculty.stanford.edu/~knuth/taocp.html.

Knuth, D. E. (2002) *The Art of Computer Programming.* Volume 2, third edition, ninth printing.
See http://Sunburn.Stanford.EDU/~knuth/news02.html.

Marsaglia, G. (1997) *A random number generator for C.* Discussion paper, posting on Usenet newsgroup `sci.stat.math` on September 29, 1997.

Marsaglia, G. and Zaman, A. (1994) Some portable very-long-period random number generators. *Computers in Physics*, **8**, 117–121.

Matsumoto, M. and Nishimura, T. (1998) Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Transactions on Modeling and Computer Simulation*, **8**, 3–30.
Source code at http://www.math.keio.ac.jp/~matumoto/emt.html.

Reeds, J., Hubert, S. and Abrahams, M. (1982–4) C implementation of SuperDuper, University of California at Berkeley. (Personal communication from Jim Reeds to Ross Ihaka.)

Wichmann, B. A. and Hill, I. D. (1982) *Algorithm AS 183: An Efficient and Portable Pseudo-random Number Generator*, Applied Statistics, **31**, 188–190; Remarks: **34**, 198 and **35**, 89.

See Also

`runif`, `rnorm`, ...

Examples

```
require(stats)

## the default random seed is 626 integers, so only print a few
runif(1); .Random.seed[1:6]; runif(1); .Random.seed[1:6]
## If there is no seed, a "random" new one is created:
rm(.Random.seed); runif(1); .Random.seed[1:6]
```

```
ok <- RNGkind()
RNGkind("Wich")# (partial string matching on 'kind')

## This shows how 'runif(.)' works for Wichmann-Hill,
## using only R functions:

p.WH <- c(30269, 30307, 30323)
a.WH <- c(  171,   172,   170)
next.WHseed <- function(i.seed = .Random.seed[-1])
  { (a.WH * i.seed) %% p.WH }
my.runif1 <- function(i.seed = .Random.seed)
  { ns <- next.WHseed(i.seed[-1]); sum(ns / p.WH) %% 1 }
rs <- .Random.seed
(WHs <- next.WHseed(rs[-1]))
u <- runif(1)
stopifnot(
 next.WHseed(rs[-1]) == .Random.seed[-1],
 all.equal(u, my.runif1(rs))
)

## ----
.Random.seed
RNGkind("Super")#matches  "Super-Duper"
RNGkind()
.Random.seed # new, corresponding to  Super-Duper

## Reset:
RNGkind(ok[1])

## ----
sum(duplicated(runif(1e6))) # around 110 for default generator
## and we would expect about almost sure duplicates beyond about
qbirthday(1-1e-6, classes=2e9) # 235,000
```

`Random.user` *User-supplied Random Number Generation*

Description

Function `RNGkind` allows user-coded uniform and normal random number generators to be supplied. The details are given here.

Details

A user-specified uniform RNG is called from entry points in dynamically-loaded compiled code. The user must supply the entry point `user_unif_rand`, which takes no arguments and returns a *pointer to* a double. The example below will show the general pattern.

Optionally, the user can supply the entry point `user_unif_init`, which is called with an `unsigned int` argument when `RNGkind` (or `set.seed`) is called, and is intended to be used to initialize the user's RNG code. The argument is intended to be used to set the 'seeds'; it is the `seed` argument to `set.seed` or an essentially random seed if `RNGkind` is called.

If only these functions are supplied, no information about the generator's state is recorded in `.Random.seed`. Optionally, functions `user_unif_nseed` and `user_unif_seedloc` can

be supplied which are called with no arguments and should return pointers to the number of seeds and to an integer array of seeds. Calls to `GetRNGstate` and `PutRNGstate` will then copy this array to and from `.Random.seed`.

A user-specified normal RNG is specified by a single entry point `user_norm_rand`, which takes no arguments and returns a *pointer to* a double.

Warning

As with all compiled code, mis-specifying these functions can crash R. Do include the 'R_ext/Random.h' header file for type checking.

Examples

```
## Not run:
##  Marsaglia's congruential PRNG
#include <R_ext/Random.h>

static Int32 seed;
static double res;
static int nseed = 1;

double * user_unif_rand()
{
    seed = 69069 * seed + 1;
    res = seed * 2.32830643653869e-10;
    return &res;
}

void  user_unif_init(Int32 seed_in) { seed = seed_in; }
int * user_unif_nseed() { return &nseed; }
int * user_unif_seedloc() { return (int *) &seed; }

/*  ratio-of-uniforms for normal  */
#include <math.h>
static double x;

double * user_norm_rand()
{
    double u, v, z;
    do {
        u = unif_rand();
        v = 0.857764 * (2. * unif_rand() - 1);
        x = v/u; z = 0.25 * x * x;
        if (z < 1. - u) break;
        if (z > 0.259/u + 0.35) continue;
    } while (z > -log(u));
    return &x;
}

## Use under Unix:
R CMD SHLIB urand.c
R
> dyn.load("urand.so")
> RNGkind("user")
> runif(10)
> .Random.seed
```

```
> RNGkind(, "user")
> rnorm(10)
> RNGkind()
[1] "user-supplied" "user-supplied"
## End(Not run)
```

`range` *Range of Values*

Description

`range` returns a vector containing the minimum and maximum of all the given arguments.

Usage

```
range(..., na.rm = FALSE)

## Default S3 method:
range(..., na.rm = FALSE, finite = FALSE)
```

Arguments

`...`	any `numeric` or character objects.
`na.rm`	logical, indicating if `NA`'s should be omitted.
`finite`	logical, indicating if all non-finite elements should be omitted.

Details

`range` is a generic function: methods can be defined for it directly or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

If `na.rm` is `FALSE`, `NA` and `NaN` values in any of the arguments will cause `NA` values to be returned, otherwise `NA` values are ignored.

If `finite` is `TRUE`, the minimum and maximum of all finite values is computed, i.e., `finite=TRUE` *includes* `na.rm=TRUE`.

A special situation occurs when there is no (after omission of `NA`s) nonempty argument left, see `min`.

S4 methods

This is part of the S4 `Summary` group generic. Methods for it must use the signature `x, ..., na.rm`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

The `extendrange()` utility; `min`, `max`, `Methods`.

Examples

```
(r.x <- range(stats::rnorm(100)))
diff(r.x) # the SAMPLE range

x <- c(NA, 1:3, -1:1/0); x
range(x)
range(x, na.rm = TRUE)
range(x, finite = TRUE)
```

`rank` *Sample Ranks*

Description

Returns the sample ranks of the values in a vector. Ties (i.e., equal values) and missing values can be handled in several ways.

Usage

```
rank(x, na.last = TRUE,
     ties.method = c("average", "first", "random", "max", "min"))
```

Arguments

`x`	a numeric, complex, character or logical vector.
`na.last`	for controlling the treatment of NAs. If `TRUE`, missing values in the data are put last; if `FALSE`, they are put first; if `NA`, they are removed; if `"keep"` they are kept with rank `NA`.
`ties.method`	a character string specifying how ties are treated, see 'Details'; can be abbreviated.

Details

If all components are different (and no `NA`s), the ranks are well defined, with values in `seq_len(x)`. With some values equal (called 'ties'), the argument `ties.method` determines the result at the corresponding indices. The `"first"` method results in a permutation with increasing values at each index set of ties. The `"random"` method puts these in random order whereas the default, `"average"`, replaces them by their mean, and `"max"` and `"min"` replaces them by their maximum and minimum respectively, the latter being the typical sports ranking.

`NA` values are never considered to be equal: for `na.last = TRUE` and `na.last = FALSE` they are given distinct ranks in the order in which they occur in `x`.

NB: `rank` is not itself generic but `xtfrm` is, and `rank(xtfrm(x), ....)` will have the desired result if there is a `xtfrm` method. Otherwise, `rank` will make use of ==, > and `is.na` methods for classed objects, possibly rather slowly.

Value

A numeric vector of the same length as `x` with names copied from `x` (unless `na.last = NA`, when missing values are removed). The vector is of integer type unless `ties.method = "average"` when it is of double type (whether or not there are any ties).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`order` and `sort`.

Examples

```
(r1 <- rank(x1 <- c(3, 1, 4, 15, 92)))
x2 <- c(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)
names(x2) <- letters[1:11]
(r2 <- rank(x2)) # ties are averaged

## rank() is "idempotent": rank(rank(x)) == rank(x) :
stopifnot(rank(r1) == r1, rank(r2) == r2)

## ranks without averaging
rank(x2, ties.method= "first")  # first occurrence wins
rank(x2, ties.method= "random") # ties broken at random
rank(x2, ties.method= "random") # and again

## keep ties ties, no average
(rma <- rank(x2, ties.method= "max"))  # as used classically
(rmi <- rank(x2, ties.method= "min"))  # as in Sports
stopifnot(rma + rmi == round(r2 + r2))
```

---

rapply *Recursively Apply a Function to a List*

Description

`rapply` is a recursive version of `lapply`.

Usage

```
rapply(object, f, classes = "ANY", deflt = NULL,
       how = c("unlist", "replace", "list"), ...)
```

Arguments

`object`	A list.
`f`	A function of a single argument.
`classes`	A character vector of `class` names, or `"ANY"` to match any class.
`deflt`	The default result (not used if `how = "replace"`).
`how`	A character string matching the three possibilities given: see 'Details'.
`...`	additional arguments passed to the call to `f`.

Details

This function has two basic modes. If `how = "replace"`, each element of the list which is not itself a list and has a class included in `classes` is replaced by the result of applying `f` to the element.

If the mode is `how = "list"` or `how = "unlist"`, the list is copied, all non-list elements which have a class included in `classes` are replaced by the result of applying `f` to the element and all others are replaced by `deflt`. Finally, if `how = "unlist"`, `unlist(recursive = TRUE)` is called on the result.

The semantics differ in detail from `lapply`: in particular the arguments are evaluated before calling the C code.

Value

If `how = "unlist"`, a vector, otherwise a list of similar structure to `object`.

References

Chambers, J. A. (1998) *Programming with Data*. Springer.
(`rapply` is only described briefly there.)

See Also

`lapply`, `dendrapply`.

Examples

```
X <- list(list(a=pi, b=list(c=1:1)), d="a test")
rapply(X, function(x) x, how="replace")
rapply(X, sqrt, classes="numeric", how="replace")
rapply(X, nchar, classes="character",
       deflt = as.integer(NA), how="list")
rapply(X, nchar, classes="character",
       deflt = as.integer(NA), how="unlist")
rapply(X, nchar, classes="character", how="unlist")
rapply(X, log, classes="numeric", how="replace", base=2)
```

---

raw *Raw Vectors*

Description

Creates or tests for objects of type `"raw"`.

Usage

```
raw(length = 0)
as.raw(x)
is.raw(x)
```

Arguments

length	desired length.
x	object to be coerced.

Details

The raw type is intended to hold raw bytes. It is possible to extract subsequences of bytes, and to replace elements (but only by elements of a raw vector). The relational operators (see Comparison) work, as do the logical operators (see Logic) with a bitwise interpretation.

A raw vector is printed with each byte separately represented as a pair of hex digits. If you want to see a character representation (with escape sequences for non-printing characters) use `rawToChar`.

Coercion to raw treats the input values as representing a small (decimal) integers, so the input is first coerced to integer, and then values which are outside the range `[0 ... 255]` or are `NA` are set to `0` (the `nul` byte).

Value

`raw` creates a raw vector of the specified length. Each element of the vector is equal to `0`. Raw vectors are used to store fixed-length sequences of bytes.

`as.raw` attempts to coerce its argument to be of raw type. The (elementwise) answer will be `0` unless the coercion succeeds.

`is.raw` returns true if and only if `typeof(x) == "raw"`.

See Also

`charToRaw`, `rawShift`, etc.

Examples

```
xx <- raw(2)
xx[1] <- as.raw(40)      # NB, not just 40.
xx[2] <- charToRaw("A")
xx

x <- "A test string"
(y <- charToRaw(x))
is.vector(y) # TRUE
rawToChar(y)
is.raw(x)
is.raw(y)

isASCII <-  function(txt) all(charToRaw(txt) <= as.raw(127))
isASCII(x)  # true
isASCII("\x9c25.63") # false (in Latin-1, this is an amount in UK pounds)
```

rawConnection *Raw Connections*

Description

Input and output raw connections.

Usage

```
rawConnection(object, open = "r")

rawConnectionValue(con)
```

Arguments

object	character or raw vector. A description of the connection. For an input this is an R raw vector object, and for an output connection the name for the connection.
open	character. Any of the standard connection open modes.
con	An output raw connection.

Details

An input raw connection is opened and the raw vector is copied at the time the connection object is created, and `close` destroys the copy.

An output raw connection is opened and creates an R raw vector internally. The raw vector can be retrieved *via* `rawConnectionValue`.

If a connection is open for both input and output the initial raw vector supplied is copied when the connections is open

Value

For `rawConnection`, a connection object of class `"rawConnection"` which inherits from class `"connection"`.

For `rawConnectionValue`, a raw vector.

Note

As output raw connections keep the internal raw vector up to date call-by-call, they are relatively expensive to use (although over-allocation is used), and it may be better to use an anonymous `file()` connection to collect output.

On (rare) platforms where `vsnprintf` does not return the needed length of output there is a 100,000 character limit on the length of line for output connections: longer lines will be truncated with a warning.

See Also

`connections`, `showConnections`.

Examples

```
zz <- rawConnection(raw(0), "r+") # start with empty raw vector
writeBin(LETTERS, zz)
seek(zz, 0)
readLines(zz) # raw vector has embedded nuls
seek(zz, 0)
writeBin(letters[1:3], zz)
rawConnectionValue(zz)
close(zz)
```

---

rawConversion *Convert to or from Raw Vectors*

Description

Conversion and manipulation of objects of type `"raw"`.

Usage

```
charToRaw(x)
rawToChar(x, multiple = FALSE)

rawShift(x, n)

rawToBits(x)
intToBits(x)
packBits(x, type = c("raw", "integer"))
```

Arguments

`x`	object to be converted or shifted.
`multiple`	logical: should the conversion be to a single character string or multiple individual characters?
`n`	the number of bits to shift. Positive numbers shift right and negative numbers shift left: allowed values are `-8 ... 8`.
`type`	the result type.

Details

`packBits` accepts raw, integer or logical inputs, the last two without any NAs.

Note that 'bytes' are not necessarily the same as characters, e.g. in UTF-8 domains.

Value

`charToRaw` converts a length-one character string to raw bytes. It does so without taking into account any declared encoding (see `Encoding`).

`rawToChar` converts raw bytes either to a single character string or a character vector of single bytes (with `""` for `0`). (Note that a single character string could contain embedded nuls, in which case it will be truncated at the first nul with a warning.) In either case it is possible to create a result which is invalid in a multibyte locale, e.g. one using UTF-8.

`rawToBits` returns a raw vector of 8 times the length of a raw vector with entries 0 or 1. `intToBits` returns a raw vector of 32 times the length of an integer vector with entries 0 or 1. (Non-integral numeric values are truncated to integers.) In both cases the unpacking is least-significant bit first.

`packBits` packs its input (using only the lowest bit for raw or integer vectors) least-significant bit first to a raw or integer vector.

Examples

```
x <- "A test string"
(y <- charToRaw(x))
is.vector(y) # TRUE

rawToChar(y)
rawToChar(y, multiple = TRUE)
(xx <- c(y, as.raw(0), charToRaw("more")))
rawToChar(xx)
xxx <- xx
xxx[length(y)+1] <- charToRaw("&")
xxx
rawToChar(xxx)

rawShift(y, 1)
rawShift(y, -2)

rawToBits(y)
```

RdUtils *Utilities for Processing Rd Files*

Description

Utilities for converting files in R documentation (Rd) format to other formats or create indices from them, and for converting documentation in other formats to Rd format.

Usage

```
R CMD Rdconv [options] file
R CMD Rd2dvi [options] files
R CMD Rd2txt [options] file
R CMD Sd2Rd [options] file
```

Arguments

`file`	the path to a file to be processed.
`files`	a list of file names specifying the R documentation sources to use, by either giving the paths to the files, or the path to a directory with the sources of a package.
`options`	further options to control the processing, or for obtaining information about usage and version of the utility.

Details

Rdconv converts Rd format to other formats. Currently, plain text, HTML, LaTeX, S version 3 (Sd), and S version 4 (.sgml) formats are supported. It can also extract the examples for run-time testing.

Rd2dvi and Rd2txt are user-level programs for producing DVI/PDF output or pretty text output from Rd sources. Rd2dvi will make use of environment variables R_PAPERSIZE (set by R CMD, with a default set when R was installed) and xdvi (the DVI previewer, default xdvi), and R_PDFVIEWER (the PDF previwer). (Valid values for R_PAPERSIZE are a4, letter, legal and executive.)

Sd2Rd converts S (version 3 or 4) documentation formats to Rd format.

Use R CMD foo --help to obtain usage information on utility foo.

Note

Conversion to S version 3/4 formats is rough: there are some '.Rd' constructs for which there is no natural analogue. They are intended as a starting point for hand-tuning.

See Also

The chapter "Processing Rd format" in the "Writing R Extensions" manual.

---

readBin *Transfer Binary Data To and From Connections*

Description

Read binary data from a connection, or write binary data to a connection.

Usage

```
readBin(con, what, n = 1, size = NA_integer_, signed = TRUE,
        endian = .Platform$endian)

writeBin(object, con, size = NA_integer_,
         endian = .Platform$endian)
```

Arguments

con	A connection object or a character string naming a file or a raw vector.
what	Either an object whose mode will give the mode of the vector to be read, or a character vector of length one describing the mode: one of "numeric", "double", "integer", "int", "logical", "complex", "character", "raw".
n	integer. The (maximal) number of records to be read. You can use an over-estimate here, but not too large as storage is reserved for n items.
size	integer. The number of bytes per element in the byte stream. The default, NA_integer_, uses the natural size. Size changing is not supported for raw and complex vectors.

signed	logical. Only used for integers of sizes 1 and 2, when it determines if the quantity on file should be regarded as a signed or unsigned integer.
endian	The endian-ness ("big" or "little" of the target system for the file. Using "swap" will force swapping endian-ness.
object	An R object to be written to the connection.

Details

If con is a character string, the functions call file to obtain an file connection which is opened for the duration of the function call.

If the connection is open it is read/written from its current position. If it is not open, it is opened for the duration of the call in an appropriate mode (binary read or write) and then closed again. An open connection must be in binary mode.

If readBin is called with con a raw vector, the data in the vector is used as input. If writeBin is called with con a raw vector, it is just an indication that a raw vector should be returned.

If size is specified and not the natural size of the object, each element of the vector is coerced to an appropriate type before being written or as it is read. Possible sizes are 1, 2, 4 and possibly 8 for integer or logical vectors, and 4, 8 and possibly 12/16 for numeric vectors. (Note that coercion occurs as signed types except if signed = FALSE when reading integers of sizes 1 and 2.) Changing sizes is unlikely to preserve NAs, and the extended precision sizes are unlikely to be portable across platforms.

readBin and writeBin read and write C-style zero-terminated character strings. Input strings are limited to 10000 characters. readChar and writeChar can be used to read and write fixed-length strings. No check is made that the string is valid in the current locale.

Handling R's missing and special (Inf, -Inf and NaN) values is discussed in the *R Data Import/Export* manual.

Value

For readBin, a vector of appropriate mode and length the number of items read (which might be less than n).

For writeBin, a raw vector (if con is a raw vector) or invisibly NULL.

Note

Integer read/writes of size 8 will be available if either C type long is of size 8 bytes or C type long long exists and is of size 8 bytes.

Real read/writes of size sizeof(long double) (usually 12 or 16 bytes) will be available only if that type is available and different from double.

If readBin(what = character()) is used incorrectly on a file which does not contain C-style character strings, warnings (usually many) are given. From a file or connection, the input will be broken into pieces of length 10000 with any final part being discarded.

See Also

The *R Data Import/Export* manual.

readChar to read/write fixed-length strings.

connections, readLines, writeLines.

.Machine for the sizes of long, long long and long double.

Examples

```
zz <- file("testbin", "wb")
writeBin(1:10, zz)
writeBin(pi, zz, endian="swap")
writeBin(pi, zz, size=4)
writeBin(pi^2, zz, size=4, endian="swap")
writeBin(pi+3i, zz)
writeBin("A test of a connection", zz)
z <- paste("A very long string", 1:100, collapse=" + ")
writeBin(z, zz)
if(.Machine$sizeof.long == 8 || .Machine$sizeof.longlong == 8)
    writeBin(as.integer(5^(1:10)), zz, size = 8)
if((s <-.Machine$sizeof.longdouble) > 8)
    writeBin((pi/3)^(1:10), zz, size = s)
close(zz)

zz <- file("testbin", "rb")
readBin(zz, integer(), 4)
readBin(zz, integer(), 6)
readBin(zz, numeric(), 1, endian="swap")
readBin(zz, numeric(), size=4)
readBin(zz, numeric(), size=4, endian="swap")
readBin(zz, complex(), 1)
readBin(zz, character(), 1)
z2 <- readBin(zz, character(), 1)
if(.Machine$sizeof.long == 8 || .Machine$sizeof.longlong == 8)
    readBin(zz, integer(), 10,  size = 8)
if((s <-.Machine$sizeof.longdouble) > 8)
    readBin(zz, numeric(), 10, size = s)
close(zz)
unlink("testbin")
stopifnot(z2 == z)

## signed vs unsigned ints
zz <- file("testbin", "wb")
x <- as.integer(seq(0, 255, 32))
writeBin(x, zz, size=1)
writeBin(x, zz, size=1)
x <- as.integer(seq(0, 60000, 10000))
writeBin(x, zz, size=2)
writeBin(x, zz, size=2)
close(zz)
zz <- file("testbin", "rb")
readBin(zz, integer(), 8, size=1)
readBin(zz, integer(), 8, size=1, signed=FALSE)
readBin(zz, integer(), 7, size=2)
readBin(zz, integer(), 7, size=2, signed=FALSE)
close(zz)
unlink("testbin")

## use of raw
z <- writeBin(pi^{1:5}, raw(), size = 4)
readBin(z, numeric(), 5, size = 4)
z <- writeBin(c("a", "test", "of", "character"), raw())
rawToChar(z)
readBin(z, character(), 4)
```

readChar *Transfer Character Strings To and From Connections*

Description

Transfer character strings to and from connections, without assuming they are null-terminated on the connection.

Usage

```
readChar(con, nchars, useBytes = FALSE)

writeChar(object, con,
          nchars = nchar(object, type="chars"), eos = "")
```

Arguments

con	A connection object, or a character string naming a file, or a raw vector.
nchars	integer, giving the lengths in characters of (unterminated) character strings to be read or written. Must be >= 0 and not missing.
useBytes	logical: should nchars be regarded as a number of bytes not characters in a multi-byte locale?
object	A character vector to be written to the connection, at least as long as nchars.
eos	'end of string': character string . The terminator to be written after each string, followed by an ASCII nul; use NULL for no terminator at all.

Details

These functions complement readBin and writeBin which read and write C-style zero-terminated character strings. They are for strings of known length, and can optionally write an end-of-string mark. They are intended only for character strings valid in the current locale.

If con is a character string, the functions call file to obtain an file connection which is opened for the duration of the function call.

If the connection is open it is read/written from its current position. If it is not open, it is opened for the duration of the call in an appropriate mode (binary read or write) and then closed again. An open connection must be in binary mode.

If readChar is called with con a raw vector, the data in the vector is used as input. If writeChar is called with con a raw vector, it is just an indication that a raw vector should be returned.

Character strings containing ASCII nul(s) will be read correctly by readChar but truncated at the first nul with a warning.

If the character length requested for readChar is longer than the data available on the connection, what is available is returned. For writeChar if too many characters are requested the output is zero-padded, with a warning.

Missing strings are written as NA.

Value

For `readChar`, a character vector of length the number of items read (which might be less than `length(nchars)`).

For `writeChar`, a raw vector (if `con` is a raw vector) or invisibly `NULL`.

Note

Earlier versions of R allowed embedded nul bytes within character strings, but not R >= 2.8.0. `readChar` was commonly used to read fixed-size zero-padded byte fields for which `readBin` was unsuitable. `readChar` can still be used for such fields if there are no embedded nuls: otherwise `readBin(what="raw")` provides an alternative.

`nchars` will be interpreted in bytes not characters in a non-UTF-8 multi-byte locale, with a warning.

There is little validity checking of UTF-8 reads.

See Also

The *R Data Import/Export* manual.

`connections`, `readLines`, `writeLines`, `readBin`

Examples

```
## test fixed-length strings
zz <- file("testchar", "wb")
x <- c("a", "this will be truncated", "abc")
nc <- c(3, 10, 3)
writeChar(x, zz, nc, eos=NULL)
writeChar(x, zz, eos="\r\n")
close(zz)

zz <- file("testchar", "rb")
readChar(zz, nc)
readChar(zz, nchar(x)+3) # need to read the terminator explicitly
close(zz)
unlink("testchar")
```

`readline` *Read a Line from the Terminal*

Description

`readline` reads a line from the terminal

Usage

```
readline(prompt = "")
```

Arguments

`prompt`	the string printed when prompting the user for input. Should usually end with a space `" "`.

Details

The prompt string will be truncated to a maximum allowed length, normally 256 chars (but can be changed in the source code).

Value

A character vector of length one.

See Also

`readLines` for reading text lines of connections, including files.

Examples

```
fun <- function() {
  ANSWER <- readline("Are you a satisfied R user? ")
  if (substr(ANSWER, 1, 1) == "n")
    cat("This is impossible.  YOU LIED!\n")
  else
    cat("I knew it.\n")
}
fun()
```

---

readLines *Read Text Lines from a Connection*

Description

Read some or all text lines from a connection.

Usage

```
readLines(con = stdin(), n = -1, ok = TRUE, warn = TRUE,
          encoding = "unknown")
```

Arguments

`con`	a connection object or a character string.
`n`	integer. The (maximal) number of lines to read. Negative values indicate that one should read up to the end of the connection.
`ok`	logical. Is it OK to reach the end of the connection before `n > 0` lines are read? If not, an error will be generated.
`warn`	logical. Warn if a text file is missing a final EOL.
`encoding`	encoding to be assumed for input strings. It is used to mark character strings as known to be in Latin-1 or UTF-8: it is not used to re-encode the input. To do the latter, specify the encoding as part of the connection `con` or via `options(encoding=)`: see the example under `file`.

Details

If the con is a character string, the function calls file to obtain a file connection which is opened for the duration of the function call.

If the connection is open it is read from its current position. If it is not open, it is opened in "rt" mode for the duration of the call and then closed again.

If the final line is incomplete (no final EOL marker) the behaviour depends on whether the connection is blocking or not. For a non-blocking text-mode connection the incomplete line is pushed back, silently. For all other connections the line will be accepted, with a warning.

Whatever mode the connection is opened in, any of LF, CRLF or CR will be accepted as the EOL marker for a line.

Value

A character vector of length the number of lines read.

The elements of the result have a declared encoding if encoding is "latin1" or "UTF-8",

Note

The default connection, stdin, may be different from con = "stdin": see file.

See Also

connections, writeLines, readBin, scan

Examples

```
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file="ex.data",
    sep="\n")
readLines("ex.data", n=-1)
unlink("ex.data") # tidy up

## difference in blocking
cat("123\nabc", file = "test1")
readLines("test1") # line with a warning

con <- file("test1", "r", blocking = FALSE)
readLines(con) # empty
cat(" def\n", file = "test1", append = TRUE)
readLines(con) # gets both
close(con)

unlink("test1") # tidy up
```

---

real *Real Vectors*

Description

These functions are the same as their double equivalents and are provided for backwards compatibility only.

Usage

```
real(length = 0)
as.real(x, ...)
is.real(x)
```

Arguments

`length`	desired length.
`x`	object to be coerced or tested.
`...`	further arguments passed to or from other methods.

Details

`as.real` is a generic function, but S3 methods must be written for `as.double`.

`Recall` *Recursive Calling*

Description

`Recall` is used as a placeholder for the name of the function in which it is called. It allows the definition of recursive functions which still work after being renamed, see example below.

Usage

```
Recall(...)
```

Arguments

`...`	all the arguments to be passed.

Note

`Recall` will not work correctly when passed as a function argument, e.g. to the `apply` family of functions.

See Also

`do.call` and `call`.

`local` for another way to write anonymous recursive functions.

Examples

```
## A trivial (but inefficient!) example:
fib <- function(n)
   if(n<=2) { if(n>=0) 1 else 0 } else Recall(n-1) + Recall(n-2)
fibonacci <- fib; rm(fib)
## renaming wouldn't work without Recall
fibonacci(10) # 55
```

`reg.finalizer` *Finalization of Objects*

Description

Registers an R function to be called upon garbage collection of object or (optionally) at the end of an R session.

Usage

```
reg.finalizer(e, f, onexit = FALSE)
```

Arguments

`e`	Object to finalize. Must be environment or external pointer.
`f`	Function to call on finalization. Must accept a single argument, which will be the object to finalize.
`onexit`	logical: should the finalizer be run if the object is still uncollected at the end of the R session?

Value

`NULL`.

Note

The purpose of this function is mainly to allow objects that refer to external items (a temporary file, say) to perform cleanup actions when they are no longer referenced from within R. This only makes sense for objects that are never copied on assignment, hence the restriction to environments and external pointers.

See Also

`gc` and `Memory` for garbage collection and memory management.

Examples

```
f <- function(e) print("cleaning....")
g <- function(x){ e <- environment(); reg.finalizer(e,f) }
g()
invisible(gc()) # trigger cleanup
```

regex *Regular Expressions as used in R*

Description

This help page documents the regular expression patterns supported by grep and related functions regexpr, gregexpr, sub and gsub, as well as by strsplit.

Details

A 'regular expression' is a pattern that describes a set of strings. Three types of regular expressions are used in R, *extended* regular expressions, used by grep(extended = TRUE) (its default), *basic* regular expressions, as used by grep(extended = FALSE), and *Perl-like* regular expressions used by grep(perl = TRUE).

Other functions which use regular expressions (often via the use of grep) include apropos, browseEnv, help.search, list.files, ls and strsplit. These will all use *extended* regular expressions, unless strsplit is called with argument extended = FALSE or perl = TRUE.

Patterns are described here as they would be printed by cat: *do remember that backslashes need to be doubled when entering* R *character strings*, e.g. from the keyboard.

Extended Regular Expressions

This section covers the regular expressions allowed if extended = TRUE in grep, regexpr, gregexpr, sub, gsub and strsplit. They use the glibc 2.7 implementation of the POSIX 1003.2 standard.

Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions.

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash. (Escaping other characters with a backslash is undefined in POSIX but gives the character in the R implementation.) The metacharacters in EREs are . \ | ( ) [{ ^ $ * + ?, but note that whether these have a special meaning depends on the context.

A *character class* is a list of characters enclosed between [and] which matches any single character in that list; unless the first character of the list is the caret ^, when it matches any character *not* in the list. For example, the regular expression [0123456789] matches any single digit, and [^abc] matches anything except the characters a, b or c. A range of characters may be specified by giving the first and last characters, separated by a hyphen. (Because their interpretation is so locale-dependent, they are best avoided.)

The precise way character ranges are interpreted depends on the values of perl and ignore.case. For basic and extended regular expressions the collation order is taken from the OS's implementation of the setting of the locale category LC_COLLATE, so [W-Z] may include x and if it does may or may not include w. (In most English locales the collation order is wWxXyYzZ.) For caseless matching the characters in a range are interpreted as if in lower case, so in an English locale [W-z] matches WXYZwxyz. The only portable way to specify all ASCII letters is to list them all as character class, [ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz].

For Perl regexps, the ranges are interpreted in the numerical order of the characters, either as bytes in a single-byte locale or as Unicode points in a UTF-8 locale. So in either case `[A-Za-z]` specifies the set of ASCII letters.

Certain named classes of characters are predefined. Their interpretation depends on the *locale* (see locales); the interpretation below is that of the POSIX locale.

`[:alnum:]` Alphanumeric characters: `[:alpha:]` and `[:digit:]`.

`[:alpha:]` Alphabetic characters: `[:lower:]` and `[:upper:]`.

`[:blank:]` Blank characters: space and tab.

`[:cntrl:]` Control characters. In ASCII, these characters have octal codes 000 through 037, and 177 (`DEL`). In another character set, these are the equivalent characters, if any.

`[:digit:]` Digits: `0 1 2 3 4 5 6 7 8 9`.

`[:graph:]` Graphical characters: `[:alnum:]` and `[:punct:]`.

`[:lower:]` Lower-case letters in the current locale.

`[:print:]` Printable characters: `[:alnum:]`, `[:punct:]` and space.

`[:punct:]` Punctuation characters: `! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~`.

`[:space:]` Space characters: tab, newline, vertical tab, form feed, carriage return, and space.

`[:upper:]` Upper-case letters in the current locale.

`[:xdigit:]` Hexadecimal digits: `0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`.

For example, `[[:alnum:]]` means `[0-9A-Za-z]`, except the latter depends upon the locale and the character encoding, whereas the former is independent of locale and character set. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.) Most metacharacters lose their special meaning inside lists. To include a literal `]`, place it first in the list. Similarly, to include a literal `^`, place it anywhere but first. Finally, to include a literal `-`, place it first or last (or, for `perl = TRUE` only, precede it by a backslash.). (Only these and `\` remain special inside character classes.)

The period `.` matches any single character. The symbol `\w` is documented to be synonym for `[[:alnum:]]` and `\W` is its negation. However, `\w` also matches underscore in the GNU grep code used in R.

The caret `^` and the dollar sign `$` are metacharacters that respectively match the empty string at the beginning and end of a line. The symbols `\<` and `\>` respectively match the empty string at the beginning and end of a word. The symbol `\b` matches the empty string at either edge of a word, and `\B` matches the empty string provided it is not at an edge of a word.

A regular expression may be followed by one of several repetition quantifiers:

`?` The preceding item is optional and will be matched at most once.

`*` The preceding item will be matched zero or more times.

`+` The preceding item will be matched one or more times.

`{n}` The preceding item is matched exactly `n` times.

`{n,}` The preceding item is matched `n` or more times.

`{n,m}` The preceding item is matched at least `n` times, but not more than `m` times.

Repetition is greedy, so the maximal possible number of repeats is used.

Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.

Two regular expressions may be joined by the infix operator `|`; the resulting regular expression matches any string matching either subexpression. For example, `abba|cde` matches either the string `abba` or the string `cde`. Note that alternation does not work inside character classes, where `|` has its literal meaning.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules.

The backreference `\N`, where N is a single digit, matches the substring previously matched by the Nth parenthesized subexpression of the regular expression.

Basic Regular Expressions

This section covers the regular expressions allowed if `extended = FALSE` in `grep`, `regexpr`, `gregexpr`, `sub`, `gsub` and `strsplit`.

In basic regular expressions the metacharacters `?`, `+`, `{`, `|`, `(`, and `)` lose their special meaning; instead use the backslashed versions `\?`, `\+`, `\ {`, `\|`, `\(`, and `\)`. Thus the metacharacters are `. \ [ ^ $ *`.

Perl Regular Expressions

The `perl = TRUE` argument to `grep`, `regexpr`, `gregexpr`, `sub`, `gsub` and `strsplit` switches to the PCRE library that 'implements regular expression pattern matching using the same syntax and semantics as Perl 5.6 or later, with just a few differences'. It adds some features from Perl 5.10.

For complete details please consult the man pages for PCRE, especially `man pcrepattern` and `man pcreapi`), on your system or from the sources at `http://www.pcre.org`. If PCRE support was compiled from the sources within R, the PCRE version is 7.7 as described here (version $\geq$ 7.6 is required if R is configured to use the system's PCRE library).

Perl regular expressions are computed byte-by-byte rather than character-by-character except in UTF-8 locales. Since the only non-UTF-8 multibyte locales in common use are those for CJK languages, they should be used with care in non-UTF-8 CJK locales.

All the regular expressions described for extended regular expressions are accepted except `\<` and `\>`: in Perl all backslashed metacharacters are alphanumeric and backslashed symbols always are interpreted as a literal character. `{` is not special if it would be the start of an invalid interval specification. There can be more than 9 backreferences. In a UTF-8 locale the named character classes only match ASCII characters: see `\p` below for an alternative.

The construct `(?...)` is used for Perl extensions in a variety of ways depending on what immediately follows the `?`.

Perl-like matching can work in several modes, set by the options `(?i)` (caseless, equivalent to Perl's `/i`), `(?m)` (multiline, equivalent to Perl's `/m`), `(?s)` (single line, so a dot matches all characters, even new lines: equivalent to Perl's `/s`) and `(?x)` (extended, whitespace data characters are ignored unless escaped and comments are allowed: equivalent to Perl's `/x`). These can be concatenated, so for example, `(?im)` sets caseless multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and to combine setting and unsetting such as `(?im-sx)`. These settings can be applied within patterns, and then apply to the remainder of the pattern. Additional options not in Perl include `(?U)` to set 'ungreedy' mode (so matching is minimal unless `?` is used, when it is greedy). Initially none of these options are set.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between \Q and \E. This is different from Perl in that $ and @ are handled as literals in \Q...\E sequences in PCRE, whereas in Perl, $ and @ cause variable interpolation.

The escape sequences \d, \s and \w represent any decimal digit, space character and 'word' character (letter, digit or underscore in the current locale, except that in a UTF-8 locale only ASCII letters are considered) respectively, and their upper-case versions represent their negation. Unlike POSIX and earlier versions of Perl and PCRE, vertical tab is not regarded as a whitespace character.

Escape sequence \a is BEL, \e is ESC, \f is FF, \n is LF, \r is CR and \t is TAB. In addition \cx is cntrl-x for any x, \ddd is the octal character ddd (for up to three digits unless interpretable as a backreference, as \1 to \7 always are), and \xhh specifies a character in hex. In a UTF-8 locale, \x{h...} specifies a Unicode point by one or more hex digits.

Outside a character class, \b matches a word boundary, \B is its negation, \A matches at start of a subject (even in multiline mode, unlike ^), \Z matches at end of a subject or before newline at end, \z matches at end of a subject. and \G matches at first matching position in a subject (which is subtly different from Perl's end of the previous match). \C matches a single byte. including a newline. In a UTF-8 locale, \R matches any Unicode newline character (not just CR), and \X matches any number of Unicode characters that form an extended Unicode sequence.

In a UTF-8 locale, some Unicode properties are supported via \p{*xx*} and \P{*xx*} which match characters with and without property *xx* respectively. For a list of supported properties see the PCRE documentation, but for example Lu is 'upper case letter' and Sc is 'currency symbol'.

The same repetition quantifiers as extended POSIX are supported. However, if a quantifier is followed by ?, the match is 'ungreedy', that is as short as possible rather than as long as possible (unless the meanings are reversed by the (?U) option.)

The sequence (?# marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part at all in the pattern matching.

If the extended option is set, an unescaped # character outside a character class introduces a comment that continues up to the next newline character in the pattern.

The pattern (?:...) groups characters just as parentheses do but does not make a backreference.

Patterns (?=...) and (?!...) are zero-width positive and negative lookahead *assertions*: they match if an attempt to match the ... forward from the current position would succeed (or not), but use up no characters in the string being processed. Patterns (?<=...) and (?<!...) are the lookbehind equivalents: they do not allow repetition quantifiers nor \C in ....

Named subpatterns, atomic grouping, possessive qualifiers and conditional and recursive patterns are not covered here.

Author(s)

This help page is based on the documentation of GNU grep 2.4.2 and the pcrepattern man page from PCRE 7.4.

See Also

grep, apropos, browseEnv, glob2rx, help.search, list.files, ls and strsplit.

http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html

The pcrepattern can be found as part of http://www.pcre.org/pcre.txt, and details of Perl's own implementation at http://perldoc.perl.org/perlre.html.

remove *Remove Objects from a Specified Environment*

Description

remove and rm can be used to remove objects. These can be specified successively as character strings, or in the character vector list, or through a combination of both. All objects thus specified will be removed.

If envir is NULL then the currently active environment is searched first.

If inherits is TRUE then parents of the supplied directory are searched until a variable with the given name is encountered. A warning is printed for each variable that is not found.

Usage

```
remove(..., list = character(0), pos = -1,
       envir = as.environment(pos), inherits = FALSE)

rm    (..., list = character(0), pos = -1,
       envir = as.environment(pos), inherits = FALSE)
```

Arguments

...	the objects to be removed, as names (unquoted) or character strings (quoted).
list	a character vector naming objects to be removed.
pos	where to do the removal. By default, uses the current environment. See the details for other possibilities.
envir	the environment to use. See the details section.
inherits	should the enclosing frames of the environment be inspected?

Details

The pos argument can specify the environment from which to remove the objects in any of several ways: as an integer (the position in the search list); as the character string name of an element in the search list; or as an environment (including using sys.frame to access the currently active function calls). The envir argument is an alternative way to specify an environment, but is primarily there for back compatibility.

It is not allowed to remove variables from the base environment and base name space, nor from any environment which is locked (see lockEnvironment).

Earlier versions of R incorrectly claimed that supplying a character vector in ... removed the objects named in the character vector, but it removed the character vector. Use the list argument to specify objects *via* a character vector.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

ls, objects

Examples

```
tmp <- 1:4
## work with tmp  and cleanup
rm(tmp)

## Not run:
## remove (almost) everything in the working environment.
## You will get no warning, so don't do this unless you are really sure.
rm(list = ls())
## End(Not run)
```

---

rep *Replicate Elements of Vectors and Lists*

Description

`rep` replicates the values in `x`. It is a generic function, and the (internal) default method is described here.

`rep.int` is a faster simplified version for the most common case.

Usage

```
rep(x, ...)

rep.int(x, times)
```

Arguments

`x`	a vector (of any mode including a list) or a pairlist or a factor or (except for `rep.int`) a `POSIXct` or `POSIXlt` or `date` object.
`...`	further arguments to be passed to or from other methods. For the internal default method these can include: **`times`** A vector giving the number of times to repeat each element if of length `length(x)`, or to repeat the whole vector if of length 1. **`length.out`** non-negative integer. The desired length of the output vector. Ignored if `NA` or invalid. **`each`** non-negative integer. Each element of `x` is repeated `each` times. Treated as `1` if `NA` or invalid.
`times`	see `....`

Details

The default behaviour is as if the call was `rep(x, times=1, length.out=NA, each=1)`. Normally just one of the additional arguments is specified, but if `each` is specified with either of the other two, its replication is performed first, and then that implied by `times` or `length.out`.

If `times` consists of a single integer, the result consists of the whole input repeated this many times. If `times` is a vector of the same length as `x` (after replication by `each`), the result consists of `x[1]` repeated `times[1]` times, `x[2]` repeated `times[2]` times and so on.

`length.out` may be given in place of `times`, in which case `x` is repeated as many times as is necessary to create a vector of this length. If both are given, `length.out` takes priority and `times` is ignored.

Non-integer values of `times` will be truncated towards zero. If `times` is a computed quantity it is prudent to add a small fuzz.

If `x` has length zero and `length.out` is supplied and is positive, the values are filled in using the extraction rules, that is by an `NA` of the appropriate class for an atomic vector (`0` for raw vectors) and `NULL` for a list.

Value

An object of the same type as `x` (except that `rep` will coerce pairlists to vector lists).

`rep.int` returns no attributes.

The default method of `rep` gives the result names (which will almost always contain duplicates) if `x` had names, but retains no other attributes except for factors.

Note

Function `rep.int` is a simple case handled by internal code, and provided as a separate function purely for S compatibility.

Function `rep` is a primitive, but (partial) matching of argument names is performed as for normal functions. You can no longer pass a missing argument to. e.g. `length.out`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`seq`, `sequence`, `replicate`.

Examples

```
rep(1:4, 2)
rep(1:4, each = 2)       # not the same.
rep(1:4, c(2,2,2,2))     # same as second.
rep(1:4, c(2,1,2,1))
rep(1:4, each = 2, len = 4)    # first 4 only.
rep(1:4, each = 2, len = 10)   # 8 integers plus two recycled 1's.
rep(1:4, each = 2, times = 3)  # length 24, 3 complete replications

rep(1, 40*(1-.8)) # length 7 on most platforms
rep(1, 40*(1-.8)+1e-7) # better

## replicate a list
fred <- list(happy = 1:10, name = "squash")
rep(fred, 5)

# date-time objects
x <- .leap.seconds[1:3]
rep(x, 2)
rep(as.POSIXlt(x), rep(2, 3))
```

```
## named factor
x <- factor(LETTERS[1:4]); names(x) <- letters[1:4]
x
rep(x, 2)
rep(x, each=2)
rep.int(x, 2)  # no names
```

`replace` *Replace Values in a Vector*

Description

`replace` replaces the values in `x` with indexes given in `list` by those given in `values`. If necessary, the values in `values` are recycled.

Usage

```
replace(x, list, values)
```

Arguments

`x`	vector
`list`	an index vector
`values`	replacement values

Value

A vector with the values replaced.

Note

`x` is unchanged: remember to assign the result.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

`Reserved` *Reserved Words in R*

Description

The reserved words in R's parser are

`if else repeat while function for in next break`

`TRUE FALSE NULL Inf NaN NA NA_integer_ NA_real_ NA_complex_ NA_character_`

`...` and `..1`, `..2` etc, which are used to refer to arguments passed down from an enclosing function.

Details

Reserved words outside quotes are always parsed to be references to the objects linked to in the 'Description', and hence they are not allowed as syntactic names (see `make.names`). They **are** allowed as non-syntactic names, e.g. inside backtick quotes.

`rev` *Reverse Elements*

Description

`rev` provides a reversed version of its argument. It is generic function with a default method for vectors and one for `dendrogram`s.

Note that this is no longer needed (nor efficient) for obtaining vectors sorted into descending order, since that is now rather more directly achievable by `sort(x, decreasing = TRUE)`.

Usage

```
rev(x)
```

Arguments

`x`	a vector or another object for which reversal is defined.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`seq`, `sort`.

Examples

```
x <- c(1:5,5:3)
## sort into descending order; first more efficiently:
stopifnot(sort(x, decreasing = TRUE) == rev(sort(x)))
stopifnot(rev(1:7) == 7:1)#- don't need 'rev' here
```

Rhome *Return the R Home Directory*

Description

Return the R home directory.

Usage

```
R.home(component="home")
```

Arguments

component	As well as "home" which gives the R home directory, other known values are "bin", "doc", "etc" and "share" giving the paths to the corresponding parts of an R installation.

Details

The R home directory is the top-level directory of the R installation being run.

The R home directory is often referred to as *R_HOME*, and is the value of an environment variable of that name in an R session. It can be found outside an R session by R RHOME.

Value

A character string giving the R home directory or path to a particular component. Normally the components are all subdirectories of the R home directory, but this may not be the case in a Unix-like installation.

rle *Run Length Encoding*

Description

Compute the lengths and values of runs of equal values in a vector – or the reverse operation.

Usage

```
rle(x)
inverse.rle(x, ...)
```

Arguments

x	an atomic vector for rle(); an object of class "rle" for inverse.rle().
...	further arguments which are ignored in R.

Details

Missing values are regarded as unequal to the previous value, even if that is also missing.

inverse.rle() is the inverse function of rle(), reconstructing x from the runs.

Value

`rle()` returns an object of class `"rle"` which is a list with components:

`lengths`	an integer vector containing the length of each run.
`values`	a vector of the same length as `lengths` with the corresponding values.

`inverse.rle()` returns an atomic vector.

Examples

```
x <- rev(rep(6:10, 1:5))
rle(x)
## lengths [1:5]  5 4 3 2 1
## values  [1:5] 10 9 8 7 6

z <- c(TRUE,TRUE,FALSE,FALSE,TRUE,FALSE,TRUE,TRUE,TRUE)
rle(z)
rle(as.character(z))

stopifnot(x == inverse.rle(rle(x)),
          z == inverse.rle(rle(z)))
```

Round *Rounding of Numbers*

Description

`ceiling` takes a single numeric argument `x` and returns a numeric vector containing the smallest integers not less than the corresponding elements of `x`.

`floor` takes a single numeric argument `x` and returns a numeric vector containing the largest integers not greater than the corresponding elements of `x`.

`trunc` takes a single numeric argument `x` and returns a numeric vector containing the integers formed by truncating the values in `x` toward `0`.

`round` rounds the values in its first argument to the specified number of decimal places (default 0).

`signif` rounds the values in its first argument to the specified number of significant digits.

`zapsmall` determines a `digits` argument `dr` for calling `round(x, digits = dr)` such that values close to zero (compared with the maximal absolute value) are 'zapped', i.e., treated as `0`.

Usage

```
ceiling(x)
floor(x)
trunc(x, ...)

round(x, digits = 0)
signif(x, digits = 6)
zapsmall(x, digits = getOption("digits"))
```

Arguments

x	a numeric vector. A complex vector is allowed for round, signif and zapsmall.
digits	integer indicating the precision to be used.
...	arguments to be passed to methods.

Details

All but zapsmall are generic functions: methods can be defined for them individually or via the Math group generic.

Note that for rounding off a 5, the IEC 60559 standard is expected to be used, '*go to the even digit*'. Therefore round(0.5) is 0 and round(-1.5) is -2. However, this is dependent on OS services and on representation error (since e.g. 0.15 is not represented exactly, the rounding rule applies to the represented number and not to the printed number, and so round(0.15, 1) could be either 0.1 or 0.2).

For signif the recognized values of digits are 1...22. Complex numbers are rounded to retain the specified number of digits in the larger of the components. Each element of the vector is rounded individually, unlike printing.

All except zapsmall are primitive, but as from R 2.7.0 arguments are matched by name in round and signif (whereas the default method for ceiling, floor and trunc ignore argument names).

S4 methods

ceiling, floor and trunc are S4 generic and members of the Math group generic. As an S4 generic, trunc has only one argument.

round and signif are S4 generic and members of the Math2 group generic.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (Except zapsmall.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (zapsmall.)

See Also

as.integer.

Examples

```
round(.5 + -2:4) # IEEE rounding: -2  0  0  2  2  4  4
( x1 <- seq(-2, 4, by = .5) )
round(x1)#-- IEEE rounding !
x1[trunc(x1) != floor(x1)]
x1[round(x1) != floor(x1 + .5)]
(non.int <- ceiling(x1) != floor(x1))

x2 <- pi * 100^(-1:3)
round(x2, 3)
signif(x2, 3)
```

```
print   (x2 / 1000, digits=4)
zapsmall(x2 / 1000, digits=4)
zapsmall(exp(1i*0:4*pi/2))
```

`round.POSIXt` *Round / Truncate Data-Time Objects*

Description

Round or truncate date-time objects.

Usage

```
## S3 method for class 'POSIXt':
round(x, units = c("secs", "mins", "hours", "days"))
## S3 method for class 'POSIXt':
trunc(x, units = c("secs", "mins", "hours", "days"), ...)

## S3 method for class 'Date':
round(x, ...)
## S3 method for class 'Date':
trunc(x, ...)
```

Arguments

`x`	an object inheriting from `"POSIXt"` or `"Date"`.
`units`	one of the units listed. Can be abbreviated.
`...`	arguments to be passed to or from other methods, notably `digits` for `round`.

Details

The time is rounded or truncated to the second, minute, hour or day. Timezones are only relevant to days, when midnight in the current timezone is used.

The methods for class `"Date"` are of little use except to remove fractional days.

Value

An object of class `"POSIXlt"` or `"Date"`.

See Also

`round` for the generic function and default methods.

`DateTimeClasses`, `Date`

Examples

```
round(.leap.seconds + 1000, "hour")
trunc(Sys.time(), "day")
```

`row` *Row Indexes*

Description

Returns a matrix of integers indicating their row number in a matrix-like object, or a factor indicating the row labels.

Usage

```
row(x, as.factor = FALSE)
```

Arguments

`x`	a matrix-like object, that is one with a two-dimensional `dim`.
`as.factor`	a logical value indicating whether the value should be returned as a factor of row labels (created if necessary) rather than as numbers.

Value

An integer (or factor) matrix with the same dimensions as `x` and whose `ij`-th element is equal to `i` (or the `i`-th row label).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`col` to get columns.

Examples

```
x <- matrix(1:12, 3, 4)
# extract the diagonal of a matrix
dx <- x[row(x) == col(x)]
dx

# create an identity 5-by-5 matrix
x <- matrix(0, nrow = 5, ncol = 5)
x[row(x) == col(x)] <- 1
x
```

`row.names` *Get and Set Row Names for Data Frames*

Description

All data frames have a row names attribute, a character vector of length the number of rows with no duplicates nor missing values.

For convenience, these are generic functions for which users can write other methods, and there are default methods for arrays. The description here is for the `data.frame` method.

Usage

```
row.names(x)
row.names(x) <- value
```

Arguments

`x`	object of class `"data.frame"`, or any other class for which a method has been defined.
`value`	an object to be coerced to character unless an integer vector. It should have (after coercion) the same length as the number of rows of `x` with no duplicated nor missing values. `NULL` is also allowed: see 'Details'.

Details

A data frame has (by definition) a vector of *row names* which has length the number of rows in the data frame, and contains neither missing nor duplicated values. Where a row names sequence has been added by the software to meet this requirement, they are regarded as 'automatic'.

Row names were character are allowed to be integer or character, but for backwards compatibility (with R <= 2.4.0) `row.names` will always return a character vector. (Use `attr(x, "row.names")` if you need an integer value.)

Using `NULL` for the value resets the row names to `seq_len(nrow(x))`, regarded as 'automatic'.

Value

`row.names` returns a character vector.

`row.names<-` returns a data frame with the row names changed.

Note

`row.names` is similar to `rownames` for arrays, and it has a method that calls `rownames` for an array argument.

Row names of the form `1:n` for `n > 2` are stored internally in a compact form, which might be seen from C code or by deparsing but never via `row.names` or `attr(x, "row.names")`. Additionally, some names of this sort are marked as 'automatic' and handled differently by `as.matrix` and `data.matrix` (and potentially other functions). (All zero-row data frames are regarded as having automatic row.names.)

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`data.frame`, `rownames`, `names`.

`.row_names_info` for the internal representations.

`row/colnames` *Row and Column Names*

Description

Retrieve or set the row or column names of a matrix-like object.

Usage

```
rownames(x, do.NULL = TRUE, prefix = "row")
rownames(x) <- value

colnames(x, do.NULL = TRUE, prefix = "col")
colnames(x) <- value
```

Arguments

`x`	a matrix-like R object, with at least two dimensions for `colnames`.
`do.NULL`	logical. Should this create names if they are `NULL`?
`prefix`	for created names.
`value`	a valid value for that component of `dimnames(x)`. For a matrix or array this is either `NULL` or a character vector of non-zero length equal to the appropriate dimension.

Details

The extractor functions try to do something sensible for any matrix-like object `x`. If the object has `dimnames` the first component is used as the row names, and the second component (if any) is used for the column names. For a data frame, `rownames` and `colnames` eventually call `row.names` and `names` respectively, but the latter are preferred.

If `do.NULL` is `FALSE`, a character vector (of length `NROW(x)` or `NCOL(x)`) is returned in any case, prepending `prefix` to simple numbers, if there are no dimnames or the corresponding component of the dimnames is `NULL`.

The replacement methods for arrays/matrices coerce vector and factor values of `value` to character, but do not dispatch methods for `as.character`.

For a data frame, `value` for `rownames` should be a character vector of non-duplicated and non-missing names (this is enforced), and for `colnames` a character vector of (preferably) unique syntactically-valid names. In both cases, `value` will be coerced by `as.character`, and setting `colnames` will convert the row names to character.

See Also

dimnames, case.names, variable.names.

Examples

```
m0 <- matrix(NA, 4, 0)
rownames(m0)

m2 <- cbind(1,1:4)
colnames(m2, do.NULL = FALSE)
colnames(m2) <- c("x","Y")
rownames(m2) <- rownames(m2, do.NULL = FALSE, prefix = "Obs.")
m2
```

---

rowsum *Give column sums of a matrix or data frame, based on a grouping variable*

Description

Compute column sums across rows of a matrix-like object for each level of a grouping variable. rowsum is generic, with a method for data frames and a default method for vectors and matrices.

Usage

```
rowsum(x, group, reorder = TRUE, ...)

## S3 method for class 'data.frame':
rowsum(x, group, reorder = TRUE, na.rm = FALSE, ...)

## Default S3 method:
rowsum(x, group, reorder = TRUE, na.rm = FALSE, ...)
```

Arguments

x	a matrix, data frame or vector of numeric data. Missing values are allowed. A numeric vector will be treated as a column vector.
group	a vector or factor giving the grouping, with one element per row of x. Missing values will be treated as another group and a warning will be given.
reorder	if TRUE, then the result will be in order of sort(unique(group)), if FALSE, it will be in the order that groups were encountered.
na.rm	logical (TRUE or FALSE). Should NA values be discarded?
...	other arguments to be passed to or from methods

Details

The default is to reorder the rows to agree with tapply as in the example below. Reordering should not add noticeably to the time except when there are very many distinct values of group and x has few columns.

The original function was written by Terry Therneau, but this is a new implementation using hashing that is much faster for large matrices.

To sum over all the rows of a matrix (ie, a single `group`) use `colSums`, which should be even faster.

Value

A matrix or data frame containing the sums. There will be one row per unique value of `group`.

See Also

`tapply`, `aggregate`, `rowSums`

Examples

```
require(stats)

x <- matrix(runif(100), ncol=5)
group <- sample(1:8, 20, TRUE)
(xsum <- rowsum(x, group))
## Slower versions
tapply(x, list(group[row(x)], col(x)), sum)
t(sapply(split(as.data.frame(x), group), colSums))
aggregate(x, list(group), sum)[-1]
```

`sample` *Random Samples and Permutations*

Description

`sample` takes a sample of the specified size from the elements of `x` using either with or without replacement.

Usage

```
sample(x, size, replace = FALSE, prob = NULL)
```

Arguments

`x`	Either a (numeric, complex, character or logical) vector of more than one element from which to choose, or a positive integer.
`size`	non-negative integer giving the number of items to choose.
`replace`	Should sampling be with replacement?
`prob`	A vector of probability weights for obtaining the elements of the vector being sampled.

Details

If `x` has length 1, is numeric (in the sense of `is.numeric`) and `x >= 1`, sampling takes place from `1:x`. *Note* that this convenience feature may lead to undesired behaviour when `x` is of varying length `sample(x)`. See the `resample()` example below.

By default `size` is equal to `length(x)` so that `sample(x)` generates a random permutation of the elements of `x` (or `1:x`).

The optional `prob` argument can be used to give a vector of weights for obtaining the elements of the vector being sampled. They need not sum to one, but they should be nonnegative and not all zero. If `replace` is true, Walker's alias method (Ripley, 1987) is used when there are more than 250 reasonably probable values: this gives results incompatible with those from R < 2.2.0, and there will be a warning the first time this happens in a session.

If `replace` is false, these probabilities are applied sequentially, that is the probability of choosing the next item is proportional to the weights amongst the remaining items. The number of nonzero weights must be at least `size` in this case.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Ripley, B. D. (1987) *Stochastic Simulation*. Wiley.

See Also

Package **sampling** for other methods of weighted sampling without replacement.

Examples

```
x <- 1:12
# a random permutation
sample(x)
# bootstrap sampling -- only if length(x) > 1 !
sample(x,replace=TRUE)

# 100 Bernoulli trials
sample(c(0,1), 100, replace = TRUE)

## More careful bootstrapping --  Consider this when using sample()
## programmatically (i.e., in your function or simulation)!

# sample()'s surprise -- example
x <- 1:10
    sample(x[x >  8]) # length 2
    sample(x[x >  9]) # oops -- length 10!
try(sample(x[x > 10]))# error!

## This is safer, but only for sampling without replacement
resample <- function(x, size, ...)
  if(length(x) <= 1) { if(!missing(size) && size == 0) x[FALSE] else x
  } else sample(x, size, ...)

resample(x[x >  8])# length 2
resample(x[x >  9])# length 1
resample(x[x > 10])# length 0
```

`save` *Save R Objects*

Description

save writes an external representation of R objects to the specified file. The objects can be read back from the file at a later date by using the function load (or data in some cases).

save.image() is just a short-cut for 'save my current workspace', i.e., save(list = ls(all=TRUE), file = ".RData"). It is also what happens with q("yes").

Usage

```
save(..., list = character(0),
     file = stop("'file' must be specified"),
     ascii = FALSE, version = NULL, envir = parent.frame(),
     compress = !ascii, eval.promises = TRUE, precheck = TRUE)

save.image(file = ".RData", version = NULL, ascii = FALSE,
           compress = !ascii, safe = TRUE)
```

Arguments

...	the names of the objects to be saved (as symbols or character strings).
list	A character vector containing the names of objects to be saved.
file	a connection or the name of the file where the data will be saved. Must be a file name for workspace format version 1.
ascii	if TRUE, an ASCII representation of the data is written. The default value of ascii is FALSE which leads to a more compact binary file being written.
version	the workspace format version to use. NULL specifies the current default format. The version used from R 0.99.0 to R 1.3.1 was version 1. The default format as from R 1.4.0 is version 2.
envir	environment to search for objects to be saved.
compress	logical specifying whether saving to a named file is to use compression. Ignored when file is a connection and for workspace format version 1.
eval.promises	logical: should objects which are promises be forced before saving?
precheck	logical: should the existence of the objects be checked before starting to save (and in particular before opening the file/connection)? Does not apply to version 1 saves.
safe	logical. If TRUE, a temporary file is used for creating the saved workspace. The temporary file is renamed to file if the save succeeds. This preserves an existing workspace file if the save fails, but at the cost of using extra disk space during the save.

Details

The names of the objects specified either as symbols (or character strings) in ... or as a character vector in list are used to look up the objects from environment envir. By default promises are evaluated, but if eval.promises = FALSE promises are saved (together with their evaluation environments). (Promises embedded in objects are always saved unevaluated.)

All R platforms use the XDR (bigendian) representation of C ints and doubles in binary save-d files, and these are portable across all R platforms. (ASCII saves used to be useful for moving data between platforms but are now mainly of historical interest.)

Default values for the `ascii`, `compress`, `safe` and `version` arguments can be modified with the `save.defaults` option (used both by `save` and `save.image`), see also the example section below. If a `save.image.defaults` option is set it overrides `save.defaults` for function `save.image` (which allows this to have different defaults).

It is possible to compress later (with `gzip`) a file saved with `compress = FALSE`: the effect is the same as saving with `compress = TRUE`.

Warnings

The `...` arguments only give the *names* of the objects to be saved: they are searched for in the environment given by the `envir` argument, and the actual objects given as arguments need not be those found.

Saved R objects are binary files, even those saved with `ascii = TRUE`, so ensure that they are transferred without conversion of end of line markers and of 8-bit characters. The lines are delimited by LF on all platforms.

Although the default version has not changed since R 1.4.0, this does not mean that saved files are necessarily backwards compatible. You will be able to load a saved image into an earlier version of R unless use is made of later additions (for example, raw vectors or external pointers).

Note

The most common reason for failure is lack of write permission in the current directory. For `save.image` and for saving at the end of a session this will shown by messages like

```
Error in gzfile(file, "wb") : unable to open connection
In addition: Warning message:
In gzfile(file, "wb") :
  cannot open compressed file '.RDataTmp',
  probable reason 'Permission denied'
```

The defaults were changed to use compressed saves for `save` in 2.3.0 and for `save.image` in 2.4.0. Any recent version of R can read compressed save files, and a compressed file can be uncompressed (by `gzip -d`) for use with very old versions of R.

See Also

`dput`, `dump`, `load`, `data`.

Examples

```
x <- stats::runif(20)
y <- list(a = 1, b = TRUE, c = "oops")
save(x, y, file = "xy.Rdata")
save.image()
unlink("xy.Rdata")
unlink(".RData")

# set save defaults using option:
options(save.defaults=list(ascii=TRUE, safe=FALSE))
save.image()
unlink(".RData")
```

scale *Scaling and Centering of Matrix-like Objects*

Description

`scale` is generic function whose default method centers and/or scales the columns of a numeric matrix.

Usage

```
scale(x, center = TRUE, scale = TRUE)
```

Arguments

`x`	a numeric matrix(like object).
`center`	either a logical value or a numeric vector of length equal to the number of columns of `x`.
`scale`	either a logical value or a numeric vector of length equal to the number of columns of `x`.

Details

The value of `center` determines how column centering is performed. If `center` is a numeric vector with length equal to the number of columns of `x`, then each column of `x` has the corresponding value from `center` subtracted from it. If `center` is `TRUE` then centering is done by subtracting the column means (omitting `NA`s) of `x` from their corresponding columns, and if `center` is `FALSE`, no centering is done.

The value of `scale` determines how column scaling is performed (after centering). If `scale` is a numeric vector with length equal to the number of columns of `x`, then each column of `x` is divided by the corresponding value from `scale`. If `scale` is `TRUE` then scaling is done by dividing the (centered) columns of `x` by their root-mean-square, and if `scale` is `FALSE`, no scaling is done.

The root-mean-square for a column is obtained by computing the square-root of the sum-of-squares of the non-missing values in the column divided by the number of non-missing values minus one.

Value

For `scale.default`, the centered, scaled matrix. The numeric centering and scalings used (if any) are returned as attributes `"scaled:center"` and `"scaled:scale"`

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`sweep` which allows centering (and scaling) with arbitrary statistics.

For working with the scale of a plot, see `par`.

Examples

```
require(stats)
x <- matrix(1:10, ncol=2)
(centered.x <- scale(x, scale=FALSE))
cov(centered.scaled.x <- scale(x))# all 1
```

---

scan *Read Data Values*

Description

Read data into a vector or list from the console or file.

Usage

```
scan(file = "", what = double(0), nmax = -1, n = -1, sep = "",
     quote = if(identical(sep, "\n")) "" else "'\"", dec = ".",
     skip = 0, nlines = 0, na.strings = "NA",
     flush = FALSE, fill = FALSE, strip.white = FALSE,
     quiet = FALSE, blank.lines.skip = TRUE, multi.line = TRUE,
     comment.char = "", allowEscapes = FALSE,
     encoding = "unknown")
```

Arguments

file	the name of a file to read data values from. If the specified file is "", then input is taken from the keyboard (or whatever stdin() reads if input is redirected or R is embedded). (In this case input can be terminated by a blank line or an EOF signal, Ctrl-D on Unix and Ctrl-Z on Windows.) Otherwise, the file name is interpreted *relative* to the current working directory (given by getwd()), unless it specifies an *absolute* path. Tilde-expansion is performed where supported. When running R from a script, file="stdin" can be used to refer to the process's stdin file stream. Alternatively, file can be a connection, which will be opened if necessary, and if so closed at the end of the function call. Whatever mode the connection is opened in, any of LF, CRLF or CR will be accepted as the EOL marker for a line and so will match sep = "\n". file can also be a complete URL. To read a data file not in the current encoding (for example a Latin-1 file in a UTF-8 locale or conversely) use a file connection setting the encoding argument.
what	the type of what gives the type of data to be read. The supported types are logical, integer, numeric, complex, character, raw and list. If what is a list, it is assumed that the lines of the data file are records each containing length(what) items ('fields') and the list components should have elements which are one of the first six types listed or NULL, see section 'Details' below.
nmax	integer: the maximum number of data values to be read, or if what is a list, the maximum number of records to be read. If omitted or not positive or an invalid value for an integer (and nlines is not set to a positive value), scan will read to the end of file.

`n` integer: the maximum number of data values to be read, defaulting to no limit. Invalid values will be ignored.

`sep` by default, scan expects to read white-space delimited input fields. Alternatively, `sep` can be used to specify a character which delimits fields. A field is always delimited by an end-of-line marker unless it is quoted.

If specified this should be the empty character string (the default) or `NULL` or a character string containing just one single-byte character.

`quote` the set of quoting characters as a single character string or `NULL`. In a multibyte locale the quoting characters must be ASCII (single-byte).

`dec` decimal point character. This should be a character string containing just one single-byte character. (`NULL` and a zero-length character vector are also accepted, and taken as the default.)

`skip` the number of lines of the input file to skip before beginning to read data values.

`nlines` if positive, the maximum number of lines of data to be read.

`na.strings` character vector. Elements of this vector are to be interpreted as missing (`NA`) values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields.

`flush` logical: if `TRUE`, `scan` will flush to the end of the line after reading the last of the fields requested. This allows putting comments after the last field, but precludes putting more that one record on a line.

`fill` logical: if `TRUE`, `scan` will implicitly add empty fields to any lines with fewer fields than implied by `what`.

`strip.white` vector of logical value(s) corresponding to items in the `what` argument. It is used only when `sep` has been specified, and allows the stripping of leading and trailing white space from `character` fields (`numeric` fields are always stripped).

If `strip.white` is of length 1, it applies to all fields; otherwise, if `strip.white[i]` is `TRUE` *and* the `i`-th field is of mode character (because `what[i]` is) then the leading and trailing white space from field `i` is stripped.

`quiet` logical: if `FALSE` (default), scan() will print a line, saying how many items have been read.

`blank.lines.skip`
logical: if `TRUE` blank lines in the input are ignored, except when counting `skip` and `nlines`.

`multi.line` logical. Only used if `what` is a list. If `FALSE`, all of a record must appear on one line (but more than one record can appear on a single line). Note that using `fill = TRUE` implies that a record will terminated at the end of a line.

`comment.char` character: a character vector of length one containing a single character or an empty string. Use `""` to turn off the interpretation of comments altogether (the default).

`allowEscapes` logical. Should C-style escapes such as `\n` be processed (the default) or read verbatim? Note that if not within quotes these could be interpreted as a delimiter (but not as a comment character).

The escapes which are interpreted are the control characters `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v` and octal and hexadecimal representations like `\040` and `\0x2A`. Any other escaped character is treated as itself, including backslash.

encoding — encoding to be assumed for input strings. If the value is "latin1" or "UTF-8" it is used to mark character strings as known to be in Latin-1 or UTF-8: it is not used to re-encode the input. To do the latter, specify the encoding as part of the connection con or *via* options(encoding=): see the example under file.

Details

The value of what can be a list of types, in which case scan returns a list of vectors with the types given by the types of the elements in what. This provides a way of reading columnar data. If any of the types is NULL, the corresponding field is skipped (but a NULL component appears in the result).

The type of what or its components can be one of the six atomic vector types or NULL (see is.atomic).

'White space' is defined for the purposes of this function as one or more contiguous characters from the set space, horizontal tab, carriage return and line feed. It does not include form feed or vertical tab, but in Latin-1 and Windows 8-bit locales 'space' includes non-breaking space.

Empty numeric fields are always regarded as missing values. Empty character fields are scanned as empty character vectors, unless na.strings contains "" when they are regarded as missing values.

The allowed input for a numeric field is optional whitespace followed either NA or an optional sign followed by a decimal or hexadecimal constant (see NumericConstants), or NaN, Inf or infinity (ignoring case). Out-of-range values are recorded as Inf, -Inf or 0.

For an integer field the allowed input is optional whitespace, followed by either NA or an optional sign and one or more digits (0-9): all out-of-range values are converted to NA_integer_.

If sep is the default (""), the character \ in a quoted string escapes the following character, so quotes may be included in the string by escaping them.

If sep is non-default, the fields may be quoted in the style of '.csv' files where separators inside quotes (' or ") are ignored and quotes may be put inside strings by doubling them. However, if sep = "\n" it is assumed by default that one wants to read entire lines verbatim.

Quoting is only interpreted in character fields and in NULL fields (which might be skipping character fields).

Note that since sep is a separator and not a terminator, reading a file by scan("foo", sep="\n", blank.lines.skip=FALSE) will give an empty final line if the file ends in a linefeed and not if it does not. This might not be what you expected; see also readLines.

If comment.char occurs (except inside a quoted character field), it signals that the rest of the line should be regarded as a comment and be discarded. Lines beginning with a comment character (possibly after white space with the default separator) are treated as blank lines.

There is a check for a user interrupt every 1000 lines if what is a list, otherwise every 10000 items.

Value

if what is a list, a list of the same length and same names (as any) as what.

Otherwise, a vector of the type of what.

Character strings in the result will have a declared encoding if encoding is "latin1" or "UTF-8".

Note

The default for `multi.line` differs from S. To read one record per line, use `flush = TRUE` and `multi.line = FALSE`. (Note that quoted character strings can still include embedded newlines.)

If number of items is not specified, the internal mechanism re-allocates memory in powers of two and so could use up to three times as much memory as needed. (It needs both old and new copies.) If you can, specify either `n` or `nmax` whenever inputting a large vector, and `nmax` or `nlines` when inputting a large list.

Using `scan` on an open connection to read partial lines can lose chars: use an explicit separator to avoid this.

Having `nul` bytes in fields (including `\0` if `allowEscapes = TRUE`) may lead to interpretation of the field being terminated at the `nul`. They not normally present in text files – see `readBin`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`read.table` for more user-friendly reading of data matrices; `readLines` to read a file a line at a time. `write`.

`Quotes` for the details of C-style escape sequences.

`readChar` and `readBin` to read fixed or variable length character strings or binary representations of numbers a few at a time from a connection.

Examples

```
cat("TITLE extra line", "2 3 5 7", "11 13 17", file="ex.data", sep="\n")
pp <- scan("ex.data", skip = 1, quiet= TRUE)
scan("ex.data", skip = 1)
scan("ex.data", skip = 1, nlines=1) # only 1 line after the skipped one
scan("ex.data", what = list("","","")) # flush is F -> read "7"
scan("ex.data", what = list("","",""), flush = TRUE)
unlink("ex.data") # tidy up
```

`search` *Give Search Path for R Objects*

Description

Gives a list of `attach`ed *packages* (see `library`), and R objects, usually `data.frames`.

Usage

```
search()
searchpaths()
```

Value

A character vector, starting with ".GlobalEnv", and ending with "package:base" which is R's **base** package required always.

searchpaths gives a similar character vector, with the entries for packages being the path to the package used to load the code.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (search.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (searchPaths.)

See Also

.packages to list just the packages on search path.

loadedNamespaces to list loaded name spaces.

attach and detach to change the search path, objects to find R objects in there.

Examples

```
search()
searchpaths()
```

---

seek *Functions to Reposition Connections*

Description

Functions to re-position connections.

Usage

```
seek(con, ...)
## S3 method for class 'connection':
seek(con, where = NA, origin = "start", rw = "", ...)

isSeekable(con)

truncate(con, ...)
```

Arguments

con	a connection.
where	numeric. A file position (relative to the origin specified by origin), or NA.
rw	character. Empty or "read" or "write", partial matches allowed.
origin	character. One of "start", "current", "end": see 'Details'.
...	further arguments passed to or from other methods.

Details

`seek` with `where = NA` returns the current byte offset of a connection (from the beginning), and with a non-missing `where` argument the connection is re-positioned (if possible) to the specified position. `isSeekable` returns whether the connection in principle supports `seek`: currently only (possibly gz-compressed) file connections do. `gzfile` connections do not support `origin = "end"`; the file position they use is that of the uncompressed file.

`where` is stored as a real but should represent an integer: non-integer values are likely to be truncated. Note that the possible values can exceed the largest representable number in an R `integer` on 64-bit OSes, and on some 32-bit OSes.

File connections can be open for both writing/appending, in which case R keeps separate positions for reading and writing. Which `seek` refers to can be set by its `rw` argument: the default is the last mode (reading or writing) which was used. Most files are only opened for reading or writing and so default to that state. If a file is open for both reading and writing but has not been used, the default is to give the reading position (0).

The initial file position for reading is always at the beginning. The initial position for writing is at the beginning of the file for modes `"r+"` and `"r+b"`, otherwise at the end of the file. Some platforms only allow writing at the end of the file in the append modes. (The reported write position for a file opened in an append mode will typically be unreliable until the file has been written to.)

If `seek` is called with a non-`NA` value of `where`, any pushback on a text-mode connection is discarded.

`truncate` truncates a file opened for writing at its current position. It works only for `file` connections, and is not implemented on all platforms: on others (including Windows) it will not work for large (> 2Gb) files.

Value

`seek` returns the current position (before any move), as a (numeric) byte offset from the origin, if relevant, or `0` if not. Note that the position can exceed the largest representable number in an R `integer` on 64-bit OSes, and on some 32-bit OSes.

`truncate` returns `NULL`: it stops with an error if it fails (or is not implemented).

`isSeekable` returns a logical value, whether the connection supports `seek`.

See Also

`connections`

---

seq *Sequence Generation*

Description

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is an internal generic which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for two common cases.

Usage

```
seq(...)

## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)

seq.int(from, to, by, length.out, along.with, ...)

seq_along(along.with)
seq_len(length.out)
```

Arguments

`...`	arguments passed to or from methods.
`from, to`	the starting and (maximal) end value of the sequence.
`by`	number: increment of the sequence.
`length.out`	desired length of the sequence. A non-negative number, which for `seq` and `seq.int` will be rounded up if fractional.
`along.with`	take the length from the length of this argument.

Details

The interpretation of the unnamed arguments of `seq` and `seq.int` is *not* standard, and it is recommended always to name the arguments when programming.

Both `seq` are `seq.int` are generic, and only the default method is described here. Typical usages are

```
seq(from, to)
seq(from, to, by= )
seq(from, to, length.out= )
seq(along.with= )
seq(from)
seq(length.out= )
```

The first form generates the sequence `from, from+/-1, ..., to` (identical to `from:to`).

The second form generates `from, from+by`, ..., up to the sequence value less than or equal to `to`. Specifying `to - from` and `by` of opposite signs is an error. Note that the final value can go just beyond `to` to allow for rounding error.

The third generates a sequence of `length.out` equally spaced values from `from` to `to`. (`length.out` is usually abbreviated to `length` or `len`, and `seq_len` is much faster.)

The fourth form generates the sequence `1, 2, ..., length(along.with)`. (`along.with` is usually abbreviated to `along`, and `seq_along` is much faster.)

The fifth form generates the sequence `1, 2, ..., length(from)` (as if argument `along.with` had been specified), *unless* the argument is numeric of length 1 when it is interpreted as `1:from` (even for `seq(0)` for compatibility with S).

The final form generates `1, 2, ..., length.out` unless `length.out = 0`, when it generates `integer(0)`.

Very small sequences (with `from - to` of the order of 10^{-14} times the larger of the ends) will return `from`.

For `seq`(only), up to two of `from`, `to` and `by` can be supplied as complex values provided `length.out` or `along.with` is specified.

Value

Currently, the default method returns a result of type `"integer"` if `from` is (numerically equal to an) integer and, e.g., only `to` is specified, or also if only `length` or only `along.with` is specified. **Note:** this may change in the future and programmers should not rely on it.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

The methods `seq.Date` and `seq.POSIXt`.

`:`, `rep`, `sequence`, `row`, `col`.

Examples

```
seq(0, 1, length.out=11)
seq(stats::rnorm(20))
seq(1, 9, by = 2) # match
seq(1, 9, by = pi)# stay below
seq(1, 6, by = 3)
seq(1.575, 5.125, by=0.05)
seq(17) # same as 1:17
```

`seq.Date` *Generate Regular Sequences of Dates*

Description

The method for `seq` for objects of class class `"Date"` representing calendar dates.

Usage

```
## S3 method for class 'Date':
seq(from, to, by, length.out = NULL, along.with = NULL, ...)
```

Arguments

`from`	starting date. Required
`to`	end date. Optional.
`by`	increment of the sequence. Optional. See 'Details'.
`length.out`	integer, optional. desired length of the sequence.
`along.with`	take the length from the length of this argument.
`...`	arguments passed to or from other methods.

Details

`by` can be specified in several ways.

- A number, taken to be in days.
- A object of class `difftime`
- A character string, containing one of `"day"`, `"week"`, `"month"` or `"year"`. This can optionally be preceded by a (positive or negative) integer and a space, or followed by `"s"`. See `seq.POSIXt` for the details of `"month"`.

Value

A vector of class `"Date"`.

See Also

`Date`

Examples

```
## first days of years
seq(as.Date("1910/1/1"), as.Date("1999/1/1"), "years")
## by month
seq(as.Date("2000/1/1"), by="month", length.out=12)
## quarters
seq(as.Date("2000/1/1"), as.Date("2003/1/1"), by="3 months")

## find all 7th of the month between two dates, the last being a 7th.
st <- as.Date("1998-12-17")
en <- as.Date("2000-1-7")
ll <- seq(en, st, by="-1 month")
rev(ll[ll > st & ll < en])
```

`seq.POSIXt` *Generate Regular Sequences of Dates*

Description

The method for `seq` for date-time classes.

Usage

```
## S3 method for class 'POSIXt':
seq(from, to, by, length.out = NULL, along.with = NULL, ...)
```

Arguments

`from`	starting date. Required.
`to`	end date. Optional.
`by`	increment of the sequence. Optional. See 'Details'.
`length.out`	integer, optional. desired length of the sequence.
`along.with`	take the length from the length of this argument.
`...`	arguments passed to or from other methods.

Details

by can be specified in several ways.

- A number, taken to be in seconds.
- A object of class difftime
- A character string, containing one of "sec", "min", "hour", "day", "DSTday", "week", "month" or "year". This can optionally be preceded by a (positive or negative) integer and a space, or followed by "s".

The difference between "day" and "DSTday" is that the former ignores changes to/from daylight savings time and the latter takes the same clock time each day. ("week" ignores DST (it is a period of 144 hours), but "7 DSTdays") can be used as an alternative. "month" and "year" allow for DST.)

The timezone of the result is taken from from: remember than GMT does not have daylight savings time.

Using "month" first advances the month without changing the day: if this results in an invalid day of the month, it is counted forward into the next month: see the examples.

Value

A vector of class "POSIXct".

See Also

DateTimeClasses

Examples

```
## first days of years
seq(ISOdate(1910,1,1), ISOdate(1999,1,1), "years")
## by month
seq(ISOdate(2000,1,1), by = "month", length.out = 12)
seq(ISOdate(2000,1,31), by = "month", length.out = 4)
## quarters
seq(ISOdate(1990,1,1), ISOdate(2000,1,1), by = "3 months")
## days vs DSTdays: use c() to lose the timezone.
seq(c(ISOdate(2000,3,20)), by = "day", length.out = 10)
seq(c(ISOdate(2000,3,20)), by = "DSTday", length.out = 10)
seq(c(ISOdate(2000,3,20)), by = "7 DSTdays", length.out = 4)
```

---

sequence *Create A Vector of Sequences*

Description

For each element of nvec the sequence seq_len(nvec[i]) is created. These are concatenated and the result returned.

Usage

```
sequence(nvec)
```

Arguments

`nvec`	a non-negative integer vector each element of which specifies the end point of a sequence.

Details

Earlier versions of `sequence` used to work for 0 or negative inputs as `seq(x) == 1:x`.

Note that `sequence <- function(nvec) unlist(lapply(nvec, seq_len))` and it mainly exists in reverence to the very early history of R.

See Also

`gl`, `seq`, `rep`.

Examples

```
sequence(c(3,2))# the concatenated sequences 1:3 and 1:2.
#> [1] 1 2 3 1 2
```

`sets` *Set Operations*

Description

Performs **set** union, intersection, (asymmetric!) difference, equality and membership on two vectors.

Usage

```
union(x, y)
intersect(x, y)
setdiff(x, y)
setequal(x, y)

is.element(el, set)
```

Arguments

`x, y, el, set`	vectors (of the same mode) containing a sequence of items (conceptually) with no duplicated values.

Details

Each of `union`, `intersect`, `setdiff` and `setequal` will discard any duplicated values in the arguments, and they apply `as.vector` to their arguments (and so in particular coerce factors to character vectors).

`is.element(x, y)` is identical to `x %in% y`.

Value

A vector of the same mode as x or y for setdiff and intersect, respectively, and of a common mode for union.

A logical scalar for setequal and a logical of the same length as x for is.element.

See Also

%in%

'plotmath' for the use of union and intersect in plot annotation.

Examples

```
(x <- c(sort(sample(1:20, 9)),NA))
(y <- c(sort(sample(3:23, 7)),NA))
union(x, y)
intersect(x, y)
setdiff(x, y)
setdiff(y, x)
setequal(x, y)

## True for all possible x & y :
setequal( union(x,y),
          c(setdiff(x,y), intersect(x,y), setdiff(y,x)))

is.element(x, y)# length 10
is.element(y, x)# length  8
```

---

setTimeLimit *Set CPU and/or Elapsed Time Limits*

Description

Functions to set CPU and/or elapsed time limits for top-level computations or the current session.

Usage

```
setTimeLimit(cpu = Inf, elapsed = Inf, transient = FALSE)

setSessionTimeLimit(cpu = Inf, elapsed = Inf)
```

Arguments

cpu	double. Limit on total cpu time.
elapsed	double. Limit on elapsed time.
transient	logical. If TRUE, the limits apply only to the rest of the current computation.

Details

setTimeLimit sets limits which apply to each top-level computation, that is a command line (including any continuation lines) entered at the console or from a file. If it is called from within a computation the limits apply to the rest of the computation and (unless transient = TRUE to subsequent top-level computations.

setSessionTimeLimit sets limits for the rest of the session. Once a session limit is reached it is reset to Inf.

Setting any limit has a small overhead – well under 1% on the systems measured.

Time limits are checked whenever a user interrupt could occur. This will happen frequently in R code and during Sys.sleep, but only at points in compiled C and Foreign code identified by the code author.

'Total cpu time' includes that used by child processes where the latter is reported.

It is possible (but very unusual) to build R without support for proc.time, in which case these functions have no effect.

---

showConnections *Display Connections*

Description

Display aspects of connections.

Usage

```
showConnections(all = FALSE)
getConnection(what)
closeAllConnections()

stdin()
stdout()
stderr()
```

Arguments

all	logical: if true all connections, including closed ones and the standard ones are displayed. If false only open user-created connections are included.
what	integer: a row number of the table given by showConnections.

Details

stdin(), stdout() and stderr() are standard connections corresponding to input, output and error on the console respectively (and not necessarily to file streams). They are text-mode connections of class "terminal" which cannot be opened or closed, and are read-only, write-only and write-only respectively. The stdout() and stderr() connections can be re-directed by sink (and in some circumstances the output from stdout() can be split: see the help page).

The encoding for stdin() when redirected can be set by the command-line flag --encoding.

showConnections returns a matrix of information. If a connection object has been lost or forgotten, getConnection will take a row number from the table and return a connection object

for that connection, which can be used to close the connection, for example. However, if there is no R level object referring to the connection it will be closed automatically at the next garbage collection.

`closeAllConnections` closes (and destroys) all user connections, restoring all `sink` diversions as it does so.

Value

`stdin()`, `stdout()` and `stderr()` return connection objects.

`showConnections` returns a character matrix of information with a row for each connection, by default only for open non-standard connections.

`getConnection` returns a connection object, or `NULL`.

Note

`stdin()` refers to the 'console' and not to the C-level 'stdin' of the process. The distinction matters in GUI consoles (which may not have an active 'stdin', and if they do it may not be connected to console input), and also in embedded applications. If you want access to the C-level file stream 'stdin', use `file("stdin")`.

When R is reading a script from a file, the *file* is the 'console': this is traditional usage to allow in-line data (see 'An Introduction to R' for an example).

See Also

`connections`

Examples

```
showConnections(all = TRUE)

textConnection(letters)
# oops, I forgot to record that one
showConnections()
#  class     description      mode text   isopen   can read can write
#3 "letters" "textConnection" "r"  "text" "opened" "yes"    "no"
## Not run: close(getConnection(3))

showConnections()
```

---

shQuote *Quote Strings for Use in OS Shells*

Description

Quote a string to be passed to an operating system shell.

Usage

```
shQuote(string, type = c("sh", "csh", "cmd"))
```

Arguments

string	a character vector, usually of length one.
type	character: the type of shell. Partial matching is supported. "cmd" refers to the Windows NT shell, and is the default under Windows.

Details

The default type of quoting supported under Unix-alikes is that for the Bourne shell sh. If the string does not contain single quotes, we can just surround it with single quotes. Otherwise, the string is surrounded in double quotes, which suppresses all special meanings of metacharacters except dollar, backquote and backslash, so these (and of course double quote) are preceded by backslash. This type of quoting is also appropriate for bash, ksh and zsh.

The other type of quoting is for the C-shell (csh and tcsh). Once again, if the string does not contain single quotes, we can just surround it with single quotes. If it does contain single quotes, we can use double quotes provided it does not contain dollar or backquote (and we need to escape backslash, exclamation mark and double quote). As a last resort, we need to split the string into pieces not containing single quotes and surround each with single quotes, and the single quotes with double quotes.

References

Loukides, M. et al (2002) *Unix Power Tools* Third Edition. O'Reilly. Section 27.12.

http://www.mhuffman.com/notes/dos/bash_cmd.htm

See Also

Quotes for quoting R code.

sQuote for quoting English text.

Examples

```
test <- "abc$def`gh`i\\j"
cat(shQuote(test), "\n")
## Not run: system(paste("echo", shQuote(test)))
test <- "don't do it!"
cat(shQuote(test), "\n")

tryit <- paste("use the", sQuote("-c"), "switch\nlike this")
cat(shQuote(tryit), "\n")
## Not run: system(paste("echo", shQuote(tryit)))
cat(shQuote(tryit, type="csh"), "\n")

## Windows-only example.
perlcmd <- 'print "Hello World\n";'
## Not run: shell(paste("perl -e", shQuote(perlcmd, type="cmd")))
```

sign *Sign Function*

Description

`sign` returns a vector with the signs of the corresponding elements of `x` (the sign of a real number is 1, 0, or -1 if the number is positive, zero, or negative, respectively).

Note that `sign` does not operate on complex vectors.

Usage

```
sign(x)
```

Arguments

`x`	a numeric vector

Details

This is a generic function: methods can be defined for it directly or via the `Math` group generic.

See Also

`abs`

Examples

```
sign(pi) # == 1
sign(-2:3)# -1 -1 0 1 1 1
```

Signals *Interrupting Execution of R*

Description

On receiving `SIGUSR1` R will save the workspace and quit. `SIGUSR2` has the same result except that the `.Last` function and `on.exit` expressions will not be called.

Usage

```
kill -USR1 pid
kill -USR2 pid
```

Arguments

`pid`	The process ID of the R process

Warning

It is possible that one or more R objects will be undergoing modification at the time the signal is sent. These objects could be saved in a corrupted form.

`sink` *Send R Output to a File*

Description

`sink` diverts R output to a connection.

`sink.number()` reports how many diversions are in use.

`sink.number(type = "message")` reports the number of the connection currently being used for error messages.

Usage

```
sink(file = NULL, append = FALSE, type = c("output", "message"),
     split = FALSE)

sink.number(type = c("output", "message"))
```

Arguments

`file`	a writable connection or a character string naming the file to write to, or `NULL` to stop sink-ing.
`append`	logical. If `TRUE`, output will be appended to `file`; otherwise, it will overwrite the contents of `file`.
`type`	character. Either the output stream or the messages stream.
`split`	logical: if `TRUE`, output will be sent to the new sink and to the current output stream, like the Unix program `tee`.

Details

`sink` diverts R output to a connection. If `file` is a character string, a file connection with that name will be established for the duration of the diversion.

Normal R output (to connection `stdout`)) is diverted by the default `type = "output"`. Only prompts and (most) messages continue to appear on the console. Messages sent to `stderr()` (including those from `message`, `warning` and `stop`) can be diverted by `sink(type = "message")` (see below).

`sink()` or `sink(file=NULL)` ends the last diversion (of the specified type). There is a stack of diversions for normal output, so output reverts to the previous diversion (if there was one). The stack is of up to 21 connections (20 diversions).

If `file` is a connection it will be opened if necessary (in `"wt"` mode) and closed once it is removed from the stack of diversions.

`split = TRUE` only splits R output (via `Rvprintf`) and the default output from `writeLines`: it does not split all output that might be sent to `stdout()`.

Sink-ing the messages stream should be done only with great care. For that stream `file` must be an already open connection, and there is no stack of connections.

Value

`sink` returns `NULL`.

For `sink.number()` the number (0, 1, 2, ...) of diversions of output in place.

For `sink.number("message")` the connection number used for messages, 2 if no diversion has been used.

Warning

Do not use a connection that is open for `sink` for any other purpose. The software will stop you closing one such inadvertently.

Do not sink the messages stream unless you understand the source code implementing it and hence the pitfalls.

Note

`sink(split = TRUE)` is only available on systems which support the C99 function `va_copy` (or under the name `__va_copy`), but we know of no current systems which do not.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

See Also

`capture.output`

Examples

```
sink("sink-examp.txt")
i <- 1:10
outer(i, i, "*")
sink()
unlink("sink-examp.txt")
## Not run:
## capture all the output to a file.
zz <- file("all.Rout", open="wt")
sink(zz)
sink(zz, type="message")
try(log("a"))
## back to the console
sink(type="message")
sink()
try(log("a"))
## End(Not run)
```

slice.index *Slice Indexes in an Array*

Description

Returns a matrix of integers indicating the number of their slice in a given array.

Usage

```
slice.index(x, MARGIN)
```

Arguments

x	an array. If x has no dimension attribute, it is considered a one-dimensional array.
MARGIN	an integer giving the dimension number to slice by.

Value

An integer array y with dimensions corresponding to those of x such that all elements of slice number i with respect to dimension MARGIN have value i.

See Also

row and col for determining row and column indexes; in fact, these are special cases of slice.index corresponding to MARGIN equal to 1 and 2, respectively when x is a matrix.

Examples

```
x <- array(1 : 24, c(2, 3, 4))
slice.index(x, 2)
```

slotOp *Extract Slots*

Description

Extract tbe contents of a slot in a object with a formal (S4) class structure.

Usage

```
object@name
```

Arguments

object	An object from a formally defined (S4) class.
name	The character-string name of the slot.

Details

This operator supports the formal classes of package **methods**, and is enabled only when **methods** is loaded (as per default). See `slot` for further details.

As from R 2.7.0 it is checked that `object` is an S4 object (see `isS4`), and as from R 2.8.0 it is an error to attempt to use `@` on any other object. (There is an exception for name `.Data` for internal use only.)

If `name` is not a slot name, an error is thrown.

Value

The current contents of the slot.

See Also

`Extract`, `slot`

`socketSelect` *Wait on Socket Connections*

Description

Waits for the first of several socket connections to become available.

Usage

```
socketSelect(socklist, write = FALSE, timeout = NULL)
```

Arguments

`socklist`	list of open socket connections
`write`	logical. If `TRUE` wait for corresponding socket to become available for writing; otherwise wait for it to become available for reading.
`timeout`	numeric or `NULL`. Time in seconds to wait for a socket to become available; `NULL` means wait indefinitely.

Details

The values in `write` are recycled if necessary to make up a logical vector the same length as `socklist`. Socket connections can appear more than once in `socklist`; this can be useful if you want to determine whether a socket is available for reading or writing.

Value

Logical the same length as `socklist` indicating whether the corresponding socket connection is available for output or input, depending on the corresponding value of `write`.

Examples

```
## Not run:
## test whether socket connection s is available for writing or reading
socketSelect(list(s,s),c(TRUE,FALSE),timeout=0)
## End(Not run)
```

`solve` *Solve a System of Equations*

Description

This generic function solves the equation `a %*% x = b` for `x`, where `b` can be either a vector or a matrix.

Usage

```
solve(a, b, ...)

## Default S3 method:
solve(a, b, tol, LINPACK = FALSE, ...)
```

Arguments

`a`	a square numeric or complex matrix containing the coefficients of the linear system.
`b`	a numeric or complex vector or matrix giving the right-hand side(s) of the linear system. If missing, `b` is taken to be an identity matrix and `solve` will return the inverse of `a`.
`tol`	the tolerance for detecting linear dependencies in the columns of `a`. If `LINPACK` is `TRUE` the default is `1e-7`, otherwise it is `.Machine$double.eps`. Future versions of R may use a tighter tolerance. Not presently used with complex matrices `a`.
`LINPACK`	logical. Should LINPACK be used (for compatibility with R < 1.7.0)? Otherwise LAPACK is used.
`...`	further arguments passed to or from other methods

Details

`a` or `b` can be complex, but this uses double complex arithmetic which might not be available on all platforms and LAPACK will always be used.

The row and column names of the result are taken from the column names of `a` and of `b` respectively. If `b` is missing the column names of the result are the row names of `a`. No check is made that the column names of `a` and the row names of `b` are equal.

For back-compatibility `a` can be a (real) QR decomposition, although `qr.solve` should be called in that case. `qr.solve` can handle non-square systems.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`solve.qr` for the `qr` method, `chol2inv` for inverting from the Choleski factor `backsolve`, `qr.solve`.

Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
h8 <- hilbert(8); h8
sh8 <- solve(h8)
round(sh8 %*% h8, 3)

A <- hilbert(4)
A[] <- as.complex(A)
## might not be supported on all platforms
try(solve(A))
```

---

sort *Sorting or Ordering Vectors*

Description

Sort (or *order*) a vector or factor (partially) into ascending (or descending) order. For ordering along more than one variable, e.g., for sorting data frames, see `order`.

Usage

```
sort(x, decreasing = FALSE, ...)

## Default S3 method:
sort(x, decreasing = FALSE, na.last = NA, ...)

sort.int(x, partial = NULL, na.last = NA, decreasing = FALSE,
         method = c("shell", "quick"), index.return = FALSE)
```

Arguments

`x`	for `sort` an R object with a class or a numeric, complex, character or logical vector. For `sort.int`, a numeric, complex, character or logical vector, or a factor.
`decreasing`	logical. Should the sort be increasing or decreasing? Not available for partial sorting.
`...`	arguments to be passed to or from methods or (for the default methods and objects without a class) to `sort.int`.
`na.last`	for controlling the treatment of `NA`s. If `TRUE`, missing values in the data are put last; if `FALSE`, they are put first; if `NA`, they are removed.
`partial`	`NULL` or an integer vector of indices for partial sorting.
`method`	character string specifying the algorithm used.
`index.return`	logical indicating if the ordering index vector should be returned as well; this is only available for a few cases, the default `na.last = NA` and full sorting of non-factors.

Details

`sort` is a generic function for which methods can be written, and `sort.int` is the internal method which is compatible with S if only the first three arguments are used.

The default `sort` method makes use of `order` for objects with classes, which in turn makes use of the generic function `xtfrm`.

If `partial` is not `NULL`, it is taken to contain indices of elements of `x` which are to be placed in their correct positions by partial sorting. After the sort, the values specified in `partial` are in their correct position in the sorted array. Any values smaller than these values are guaranteed to have a smaller index in the sorted array and any values which are greater are guaranteed to have a bigger index in the sorted array. (This is included for efficiency, and many of the options are not available for partial sorting. It is only substantially more efficient if `partial` has a handful of elements, and a full sort is done if there are more than 10.) Names are discarded for partial sorting.

Complex values are sorted first by the real part, then the imaginary part.

The sort order for character vectors will depend on the collating sequence of the locale in use: see `Comparison`. The sort order for factors is the order of their levels (which is particularly appropriate for ordered factors).

Method `"shell"` uses Shellsort (an $O(n^{4/3})$ variant from Sedgewick (1996)). If `x` has names a stable sort is used, so ties are not reordered. (This only matters if names are present.)

Method `"quick"` uses Singleton's Quicksort implementation and is only available when `x` is numeric (double or integer) and `partial` is `NULL`. (For other types of `x` Shellsort is used, silently.) It is normally somewhat faster than Shellsort (perhaps twice as fast on vectors of length a million) but has poor performance in the rare worst case. (Peto's modification using a pseudo-random midpoint is used to make the worst case rarer.) This is not a stable sort, and ties may be reordered.

Value

For `sort`, the result depends on the S3 method which is dispatched. If `x` does not have a class the rest of this section applies. For classed objects which do not have a specific method the default method will be used and is equivalent to `x[order(x, ...)]`: this depends on the class having a suitable method for `[` (and also that `order` will work, which is not the case for a class based on a list).

For `sort.int` the sorted vector unless `index.return` is true, when the result is a list with components named `x` and `ix` containing the sorted numbers and the ordering index vector. In the latter case, if `method == "quick"` ties may be reversed in the ordering, unlike `sort.list`, as quicksort is not stable.

All attributes are removed from the return value (see Becker *et al*, 1988, p.146) except names, which are sorted. (If `partial` is specified even the names are removed.) Note that this means that the returned value has no class, except for factors and ordered factors (which are treated specially and whose result is transformed back to the original class).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Sedgewick, R. (1986) A new upper bound for Shell sort. *J. Algorithms* **7**, 159–173.

Singleton, R. C. (1969) An efficient algorithm for sorting with minimal storage: Algorithm 347. *Communications of the ACM* **12**, 185–187.

See Also

order for sorting on or reordering multiple variables.

is.unsorted. rank.

Examples

```
require(stats)

x <- swiss$Education[1:25]
x; sort(x); sort(x, partial = c(10, 15))
median.default # shows you another example for 'partial'

## illustrate 'stable' sorting (of ties):
sort(c(10:3,2:12), method = "sh", index.return=TRUE) # is stable
## $x : 2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 12
## $ix: 9  8 10  7 11  6 12  5 13  4 14  3 15  2 16  1 17 18 19
sort(c(10:3,2:12), method = "qu", index.return=TRUE) # is not
## $x : 2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 12
## $ix: 9 10  8  7 11  6 12  5 13  4 14  3 15 16  2 17  1 18 19
##        ^^^^^

x <- c(1:3, 3:5, 10)
is.unsorted(x)                #-> FALSE: is sorted
is.unsorted(x, strictly=TRUE) #-> TRUE : is not (and cannot be) sorted strictly

## Not run: ## Small speed comparison simulation:
N <- 2000
Sim <- 20
rep <- 1000 # << adjust to your CPU
c1 <- c2 <- numeric(Sim)
for(is in 1:Sim){
  x <- rnorm(N)
  c1[is] <- system.time(for(i in 1:rep) sort(x, method = "shell"))[1]
  c2[is] <- system.time(for(i in 1:rep) sort(x, method = "quick"))[1]
  stopifnot(sort(x, method = "s") == sort(x, method = "q"))
}
rbind(ShellSort = c1, QuickSort = c2)
cat("Speedup factor of quick sort():\n")
summary({qq <- c1 / c2; qq[is.finite(qq)]})

## A larger test
x <- rnorm(1e7)
system.time(x1 <- sort(x, method = "shell"))
system.time(x2 <- sort(x, method = "quick"))
stopifnot(identical(x1, x2))
## End(Not run)
```

---

source *Read R Code from a File or a Connection*

Description

source causes R to accept its input from the named file or URL (the name must be quoted) or connection. Input is read and parsed by from that file until the end of the file is reached, then the parsed expressions are evaluated sequentially in the chosen environment.

Usage

```
source(file, local = FALSE, echo = verbose, print.eval = echo,
       verbose = getOption("verbose"),
       prompt.echo = getOption("prompt"),
       max.deparse.length = 150, chdir = FALSE,
       encoding = getOption("encoding"),
       continue.echo = getOption("continue"),
       skip.echo = 0, keep.source = getOption("keep.source"))
```

Arguments

file	a connection or a character string giving the pathname of the file or URL to read from. "" indicates the connection stdin().
local	if local is FALSE, the statements scanned are evaluated in the user's workspace (the global environment), otherwise in the environment calling source.
echo	logical; if TRUE, each expression is printed after parsing, before evaluation.
print.eval	logical; if TRUE, the result of eval(i) is printed for each expression i; defaults to the value of echo.
verbose	if TRUE, more diagnostics (than just echo = TRUE) are printed during parsing and evaluation of input, including extra info for **each** expression.
prompt.echo	character; gives the prompt to be used if echo = TRUE.
max.deparse.length	integer; is used only if echo is TRUE and gives the maximal number of characters output for the deparse of a single expression.
chdir	logical; if TRUE and file is a pathname, the R working directory is temporarily changed to the directory containing file for evaluating.
encoding	character vector. The encoding(s) to be assumed when file is a character string: see file. A possible value is "unknown" when the encoding is guesses: see the 'Details'.
continue.echo	character; gives the prompt to use on continuation lines if echo = TRUE.
skip.echo	integer; how many comment lines at the start of the file to skip if echo = TRUE.
keep.source	logical: should the source formatting be retained when echo expressions, if possible?

Details

Note that running code via source differs in a few respects from entering it at the R command line. Since expressions are not executed at the top level, auto-printing is not done. So you will need to include explicit print calls for things you want to be printed (and remember that this includes plotting by **lattice**, FAQ Q7.22). Since the complete file is parsed before any of it is run, syntax errors result in none of the code being run. If an error occurs in running a syntactically correct script, anything assigned into the workspace by code that has been run will be kept (just as from the command line), but diagnostic information such as traceback() will contain additional calls to eval.with.vis, an undocumented internal function.

All versions of R accept input from a connection with end of line marked by LF (as used on Unix), CRLF (as used on DOS/Windows) or CR (as used on classic MacOS). The final line can be incomplete, that is missing the final end-of-line marker.

If `options`("keep.source") is true (the default in interactive use), the source of functions is kept so they can be listed exactly as input. This imposes a limit of 128K bytes on the function size and a nesting limit of 265. Use `keep.source = FALSE` when these limits might take effect: if exceeded they generate an error.

This paragraph applies if `file` is a filename (rather than a connection). If `encoding = "unknown"`, an attempt is made to guess the encoding. The result of `localeToCharset()` is used as a guide. If `encoding` has two or more elements, they are tried in turn until the file/URL can be read without error in the trial encoding.

If `file` is a connection (including one specified by `""`, it is not possible to re-encode the input inside `source`, and so the `encoding` argument is just used to mark character strings in the parsed input: see `parse`.

Unlike input from a console, lines in the file or on a connection can contain an unlimited number of characters.

When `skip.echo > 0`, that many comment lines at the start of the file will not be echoed. This does not affect the execution of the code at all. If there are executable lines within the first `skip.echo` lines, echoing will start with the first of them.

If `echo` is true and a deparsed expression exceeds `max.deparse.length`, that many characters are output followed by `.... [TRUNCATED] .`

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`demo` which uses `source`; `eval`, `parse` and `scan`; `options("keep.source")`.

`sys.source` which is a streamlined version to source a file into an environment.

Examples

```
## If you want to source() a bunch of files, something like
## the following may be useful:
 sourceDir <- function(path, trace = TRUE, ...) {
    for (nm in list.files(path, pattern = "\\.[RrSsQq]$")) {
       if(trace) cat(nm,":")
       source(file.path(path, nm), ...)
       if(trace) cat("\n")
    }
 }
```

`Special` *Special Functions of Mathematics*

Description

Special mathematical functions related to the beta and gamma functions.

Usage

```
beta(a, b)
lbeta(a, b)

gamma(x)
lgamma(x)
psigamma(x, deriv = 0)
digamma(x)
trigamma(x)

choose(n, k)
lchoose(n, k)
factorial(x)
lfactorial(x)
```

Arguments

`a, b`	non-negative numeric vectors.
`x, n`	numeric vectors.
`k, deriv`	integer vectors.

Details

The functions `beta` and `lbeta` return the beta function and the natural logarithm of the beta function,

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

The formal definition is

$$B(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1}dt$$

(Abramowitz and Stegun section 6.2.1, page 258). Note that it is only defined in R for non-negative `a` and `b`, and is infinite if either is zero.

The functions `gamma` and `lgamma` return the gamma function $\Gamma(x)$ and the natural logarithm of *the absolute value of* the gamma function. The gamma function is defined by (Abramowitz and Stegun section 6.1.1, page 255)

$$\Gamma(x) = \int_0^\infty t^{a-1}e^{-t}dt$$

for all real `x` except zero and negative integers (when `NaN` is returned). `factorial(x)` ($x!$ for non-negative integer `x`) is defined to be `gamma(x+1)` and `lfactorial` to be `lgamma(x+1)`.

The functions `digamma` and `trigamma` return the first and second derivatives of the logarithm of the gamma function. `psigamma(x, deriv)` (`deriv >= 0`) computes the `deriv`-th derivative of $\psi(x)$.

$$\texttt{digamma(x)} = \psi(x) = \frac{d}{dx}\ln\Gamma(x) = \frac{\Gamma'(x)}{\Gamma(x)}$$

This is often called the 'polygamma' function, e.g. in Abramowitz and Stegun (section 6.4.1, page 260); and its higher derivatives (`deriv = 2:4`) have occasionally been called 'tetragamma', 'pentagamma', and 'hexagamma'.

The functions `choose` and `lchoose` return binomial coefficients and their logarithms. Note that `choose(n,k)` is defined for all real numbers n and integer k. For $k \geq 1$ as $n(n-1)\cdots(n-k+$

$1)/k!$, as 1 for $k = 0$ and as 0 for negative k. Non-integer values of `k` are rounded to an integer, with a warning.
`choose(*,k)` uses direct arithmetic (instead of `[l]gamma` calls) for small `k`, for speed and accuracy reasons. Note the function `combn` (package **utils**) for enumeration of all possible combinations.

The `gamma`, `lgamma`, `digamma` and `trigamma` functions are generic: methods can be defined for them individually or via the `Math` group generic.

Source

`gamma`, `lgamma`, `beta` and `lbeta` are based on C translations of Fortran subroutines by W. Fullerton of Los Alamos Scientific Laboratory (now available as part of SLATEC).

`digamma`, `trigamma` and `psigamma` are based on

Amos, D. E. (1983). A portable Fortran subroutine for derivatives of the psi function, Algorithm 610, *ACM Transactions on Mathematical Software* **9(4)**, 494–502.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (For `gamma` and `lgamma`.)

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* New York: Dover. Chapter 6: Gamma and Related Functions.

See Also

`Arithmetic` for simple, `sqrt` for miscellaneous mathematical functions and `Bessel` for the real Bessel functions. Note that `gammaCody(x)` is considerably faster than `gamma(x)` but slightly less accurate and (potentially) less reliable.

For the incomplete gamma function see `pgamma`.

Examples

```
require(graphics)

choose(5, 2)
for (n in 0:10) print(choose(n, k = 0:n))

factorial(100)
lfactorial(10000)

## gamma has 1st order poles at 0, -1, -2, ...
## this will generate loss of precision warnings, so turn off
op <- options("warn")
options(warn = -1)
x <- sort(c(seq(-3,4, length.out=201), outer(0:-3, (-1:1)*1e-6, "+")))
plot(x, gamma(x), ylim=c(-20,20), col="red", type="l", lwd=2,
     main=expression(Gamma(x)))
abline(h=0, v=-3:0, lty=3, col="midnightblue")
options(op)

x <- seq(.1, 4, length.out = 201); dx <- diff(x)[1]
par(mfrow = c(2, 3))
for (ch in c("", "l","di","tri","tetra","penta")) {
  is.deriv <- nchar(ch) >= 2
```

```
  nm <- paste(ch, "gamma", sep = "")
  if (is.deriv) {
    dy <- diff(y) / dx # finite difference
    der <- which(ch == c("di","tri","tetra","penta")) - 1
    nm2 <- paste("psigamma(*, deriv = ", der,")",sep='')
    nm  <- if(der >= 2) nm2 else paste(nm, nm2, sep = " ==\n")
    y <- psigamma(x, deriv=der)
  } else {
    y <- get(nm)(x)
  }
  plot(x, y, type = "l", main = nm, col = "red")
  abline(h = 0, col = "lightgray")
  if (is.deriv) lines(x[-1], dy, col = "blue", lty = 2)
}
par(mfrow = c(1, 1))

## "Extended" Pascal triangle:
fN <- function(n) formatC(n, width=2)
for (n in -4:10) cat(fN(n),":", fN(choose(n, k= -2:max(3,n+2))), "\n")

## R code version of choose()  [simplistic; warning for k < 0]:
mychoose <- function(r,k)
    ifelse(k <= 0, (k==0),
           sapply(k, function(k) prod(r:(r-k+1))) / factorial(k))
k <- -1:6
cbind(k=k, choose(1/2, k), mychoose(1/2, k))

## Binomial theorem for n=1/2 ;
## sqrt(1+x) = (1+x)^(1/2) = sum_{k=0}^Inf  choose(1/2, k) * x^k :
k <- 0:10 # 10 is sufficient for ~ 9 digit precision:
sqrt(1.25)
sum(choose(1/2, k)* .25^k)
```

`split` *Divide into Groups and Reassemble*

Description

`split` divides the data in the vector `x` into the groups defined by `f`. The replacement forms replace values corresponding to such a division. `unsplit` reverses the effect of `split`.

Usage

```
split(x, f, drop = FALSE, ...)
split(x, f, drop = FALSE, ...) <- value
unsplit(value, f, drop = FALSE)
```

Arguments

`x`	vector or data frame containing values to be divided into groups.
`f`	a 'factor' in the sense that `as.factor(f)` defines the grouping, or a list of such factors in which case their interaction is used for the grouping.

drop	logical indicating if levels that do not occur should be dropped (if f is a factor or a list).
value	a list of vectors or data frames compatible with a splitting of x. Recycling applies if the lengths do not match.
...	further potential arguments passed to methods.

Details

split and split<- are generic functions with default and data.frame methods. The data frame method can also be used to split a matrix into a list of matrices, and the replacement form likewise, provided they are invoked explicitly.

unsplit works with lists of vectors or data frames (assumed to have compatible structure, as if created by split). It puts elements or rows back in the positions given by f. In the data frame case, row names are obtained by unsplitting the row name vectors from the elements of value.

f is recycled as necessary and if the length of x is not a multiple of the length of f a warning is printed.

Any missing values in f are dropped together with the corresponding values of x.

Value

The value returned from split is a list of vectors containing the values for the groups. The components of the list are named by the levels of f (after converting to a factor, or if already a factor and drop=TRUE, dropping unused levels).

The replacement forms return their right hand side. unsplit returns a vector or data frame for which split(x, f) equals value

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

cut

Examples

```
require(stats); require(graphics)
n <- 10; nn <- 100
g <- factor(round(n * stats::runif(n * nn)))
x <- rnorm(n * nn) + sqrt(as.numeric(g))
xg <- split(x, g)
boxplot(xg, col = "lavender", notch = TRUE, varwidth = TRUE)
sapply(xg, length)
sapply(xg, mean)

### Calculate z-scores by group

z <- unsplit(lapply(split(x, g), scale), g)
tapply(z, g, mean)

# or
```

```
z <- x
split(z, g) <- lapply(split(x, g), scale)
tapply(z, g, sd)

### data frame variation

## Notice that assignment form is not used since a variable is being added

g <- airquality$Month
l <- split(airquality, g)
l <- lapply(l, transform, Oz.Z = scale(Ozone))
aq2 <- unsplit(l, g)
head(aq2)
with(aq2, tapply(Oz.Z,  Month, sd, na.rm=TRUE))


### Split a matrix into a list by columns
ma <- cbind(x = 1:10, y = (-4:5)^2)
split(ma, col(ma))

split(1:10, 1:2)
```

`sprintf` *Use C-style String Formatting Commands*

Description

A wrapper for the C function `sprintf`, that returns a character vector containing a formatted combination of text and variable values.

Usage

```
sprintf(fmt, ...)
gettextf(fmt, ..., domain = NULL)
```

Arguments

`fmt`	a character vector of format strings, each of up to 8192 bytes.
`...`	values to be passed into `fmt`. Only logical, integer, real and character vectors are supported, but some coercion will be done: see the 'Details' section.
`domain`	see `gettext`.

Details

`sprintf` is a wrapper for the system `sprintf` C-library function. Attempts are made to check that the mode of the values passed match the format supplied, and R's special values (`NA`, `Inf`, `-Inf` and `NaN`) are handled correctly.

`gettextf` is a convenience function which provides C-style string formatting with possible translation of the format string.

The arguments (including `fmt`) are recycled if possible a whole number of times to the length of the longest, and then the formatting is done in parallel.

The following is abstracted from Kernighan and Ritchie (see References). The string `fmt` contains normal characters, which are passed through to the output string, and also conversion specifications which operate on the arguments provided through `...`. The allowed conversion specifications start with a `%` and end with one of the letters in the set `difeEgGsxX%`. These letters denote the following types:

`d, i, x, X` Integer value, `x` and `X` being hexadecimal (using the same case for `a-f` as the code). Numeric variables with exactly integer values will be coerced to integer. Formats `d` and `i` can also be used for logical variables, which will be converted to `0`, `1` or `NA`.

`f` Double precision value, in decimal notation of the form "[-]mmm.ddd". The number of decimal places is specified by the precision: the default is 6; a precision of 0 suppresses the decimal point. Non-finite values are converted to `NA`, `NaN` or (perhaps a sign followed by) `Inf`.

`e, E` Double precision value, in decimal notation of the form `[-]m.ddde[+-]xx` or `[-]m.dddE[+-]xx`.

`g, G` Double precision value, in `%e` or `%E` format if the exponent is less than -4 or greater than or equal to the precision, and `%f` format otherwise.

`a, A` Double precision value, in binary notation of the form `[-]0xh.hhhp[+-]d`. This is a binary fraction expressed in hex multiplied by a (decimal) power of 2. The number of hex digits after the decimal point is specified by the precision: the default is enough digits to represent exactly the internal binary representation. Non-finite values are converted to `NA`, `NaN` or (perhaps a sign followed by) `Inf`. Format `%a` uses lower-case for `x`, `p` and the hex values: format `%A` uses upper-case.

This might not be supported on all platforms as if is a feature of C99. The format is not uniquely defined: although it would be possible to make the leading `h` always zero or one, this is not always done. Most systems will suppress trailing zeros, but a few do not. On a well-written platform, for normal numbers there will be a leading one before the decimal point plus (by default) 13 hexadecimal digits, hence 53 bits. (The treatment of denormalized numbers is very platform-dependent.)

`s` Character string. Character `NA`s are converted to `"NA"`.

`%` Literal `%` (none of the extra formatting characters given below are permitted in this case).

Conversion by `as.character` is used for non-character arguments with `s` and by `as.double` for non-double arguments with `f, e, E, g, G`. NB: the length is determined before conversion, so do not rely on the internal coercion if this would change the length. The coercion is done only once, so if `length(fmt) > 1` then all elements must expect the same types of arguments.

In addition, between the initial `%` and the terminating conversion character there may be, in any order:

`m.n` Two numbers separated by a period, denoting the field width (`m`) and the precision (`n`).

`-` Left adjustment of converted argument in its field.

`+` Always print number with sign: by default only negative numbers are printed with a sign.

a space Prefix a space if the first character is not a sign.

`0` For numbers, pad to the field width with leading zeros.

Further, immediately after `%` may come `1$` to `99$` to refer to numbered argument: this allows arguments to be referenced out of order and is mainly intended for translators of error messages. If this is done it is best if all formats are numbered: if not the unnumbered ones process the arguments in order. See the examples. This notation allows arguments to be used more than once, in which case they must be used as the same type (integer, double or character).

A field width or precision (but not both) may be indicated by an asterisk `*`: in this case an argument specifies the desired number. A negative field width is taken as a '-' flag followed by a positive field width. A negative precision is treated as if the precision were omitted. The argument should be integer, but a double argument will be coerced to integer.

There is a limit of 8192 bytes on elements of `fmt` and also on strings included by a `%s` conversion specification.

Field widths and precisions of `%s` conversions are interpreted as bytes, not characters, as described in the C standard.

Value

A character vector of length that of the longest input. If any element of `fmt` or any character argument is declared as UTF-8, the element of the result will be in UTF-8 and have the encoding declared as UTF-8. Otherwise it will be in the current locale's encoding.

Warning

The format string is passed down the OS's `sprintf` function, and incorrect formats can cause the latter to crash the R process . R does perform basic sanity checks on the format, but not all possible user errors will be caught and some might be terminal.

Author(s)

Original code by Jonathan Rougier.

References

Kernighan, B. W. and Ritchie, D. M. (1988) *The C Programming Language.* Second edition, Prentice Hall. describes the format options in table B-1 in the Appendix.

See Also

`formatC` for a way of formatting vectors of numbers in a similar fashion.

`paste` for another way of creating a vector combining text and values.

`gettext` for the mechanisms for the automated translation of text.

Examples

```
## be careful with the format: most things in R are floats
## only integer-valued reals get coerced to integer.

sprintf("%s is %f feet tall\n", "Sven", 7.1)      # OK
try(sprintf("%s is %i feet tall\n", "Sven", 7.1)) # not OK
try(sprintf("%s is %i feet tall\n", "Sven", 7))   # OK

## use a literal % :

sprintf("%.0f%% said yes (out of a sample of size %.0f)", 66.666, 3)

## various formats of pi :

sprintf("%f", pi)
sprintf("%.3f", pi)
sprintf("%1.0f", pi)
```

```
sprintf("%5.1f", pi)
sprintf("%05.1f", pi)
sprintf("%+f", pi)
sprintf("% f", pi)
sprintf("%-10f", pi) # left justified
sprintf("%e", pi)
sprintf("%E", pi)
sprintf("%g", pi)
sprintf("%g",   1e6 * pi) # -> exponential
sprintf("%.9g", 1e6 * pi) # -> "fixed"
sprintf("%G", 1e-6 * pi)

## no truncation:
sprintf("%1.f",101)

## re-use one argument three times, show difference between %x and %X
xx <- sprintf("%1$d %1$x %1$X", 0:15)
xx <- matrix(xx, dimnames=list(rep("", 16), "%d%x%X"))
noquote(format(xx, justify="right"))

## More sophisticated:

sprintf("min 10-char string '%10s'",
        c("a", "ABC", "and an even longer one"))

n <- 1:18
sprintf(paste("e with %2d digits = %.",n,"g",sep=""), n, exp(1))

## Using arguments out of order
sprintf("second %2$1.0f, first %1$5.2f, third %3$1.0f", pi, 2, 3)

## Using asterisk for width or precision
sprintf("precision %.*f, width '%*.3f'", 3, pi, 8, pi)

## Asterisk and argument re-use, 'e' example reiterated:
sprintf("e with %1$2d digits = %2$.*1$g", n, exp(1))

## re-cycle arguments
sprintf("%s %d", "test", 1:3)

## binary output showing rounding/representation errors
x <- seq(0, 1.0, 0.1); y <- c(0,.1,.2,.3,.4,.5,.6,.7,.8,.9,1)
cbind(x, sprintf("%a", x), sprintf("%a", y))
```

`sQuote` *Quote Text*

Description

Single or double quote text by combining with appropriate single or double left and right quotation marks.

Usage

```
sQuote(x)
dQuote(x)
```

Arguments

`x`	an R object, to be coerced to a character vector.

Details

The purpose of the functions is to provide a simple means of markup for quoting text to be used in the R output, e.g., in warnings or error messages.

The choice of the appropriate quotation marks depends on both the locale and the available character sets. Older Unix/X11 fonts displayed the grave accent (ASCII code 0x60) and the apostrophe (0x27) in a way that they could also be used as matching open and close single quotation marks. Using modern fonts, or non-Unix systems, these characters no longer produce matching glyphs. Unicode provides left and right single quotation mark characters (U+2018 and U+2019); if Unicode markup cannot be assumed to be available, it seems good practice to use the apostrophe as a non-directional single quotation mark.

Similarly, Unicode has left and right double quotation mark characters (U+201C and U+201D); if only ASCII's typewriter characteristics can be employed, than the ASCII quotation mark (0x22) should be used as both the left and right double quotation mark.

Some other locales also have the directional quotation marks, notably on Windows. TeX uses grave and apostrophe for the directional single quotation marks, and doubled grave and doubled apostrophe for the directional double quotation marks.

What rendering is used depend on the `options` setting for `useFancyQuotes`. If this is `FALSE` then the undirectional ASCII quotation style is used. If this is `TRUE` (the default), Unicode directional quotes are used are used where available (currently, UTF-8 locales on Unix-alikes and all Windows locales except `C`): if set to `"UTF-8"` UTF-8 markup is used (whatever the current locale). If set to `"TeX"`, TeX-style markup is used. Finally, if this is set to a character vector of length four, the first two entries are used for beginning and ending single quotes and the second two for beginning and ending double quotes: this can be used to implement non-English quoting conventions such as the use of guillemets.

Where fancy quotes are used, you should be aware that they may not be rendered correctly as not all fonts include the requisite glyphs: for example some have directional single quotes but not directional double quotes.

Value

A character vector in the current locale's encoding.

References

Markus Kuhn, "ASCII and Unicode quotation marks". http://www.cl.cam.ac.uk/~mgk25/ucs/quotes.html

See Also

`Quotes` for quoting R code.

`shQuote` for quoting OS commands.

Examples

```
op <- options("useFancyQuotes")
paste("argument", sQuote("x"), "must be non-zero")
options(useFancyQuotes = FALSE)
```

```
cat("\ndistinguish plain", sQuote("single"), "and",
    dQuote("double"), "quotes\n")
options(useFancyQuotes = TRUE)
cat("\ndistinguish fancy", sQuote("single"), "and",
    dQuote("double"), "quotes\n")
options(useFancyQuotes = "TeX")
cat("\ndistinguish TeX", sQuote("single"), "and",
    dQuote("double"), "quotes\n")
if(l10n_info()$`Latin-1`) {
    options(useFancyQuotes = c("\xab", "\xbb", "\xbf", "?"))
    cat("\n", sQuote("guillemet"), "and",
        dQuote("Spanish question"), "styles\n")
} else if(l10n_info()$`UTF-8`) {
    options(useFancyQuotes = c("\xc2\xab", "\xc2\xbb", "\xc2\xbf", "?"))
    cat("\n", sQuote("guillemet"), "and",
        dQuote("Spanish question"), "styles\n")
}
options(op)
```

`srcfile` *References to source files*

Description

These functions are for working with source files.

Usage

```
srcfile(filename, encoding = getOption("encoding"))
srcfilecopy(filename, lines)
getSrcLines(srcfile, first, last)
srcref(srcfile, lloc)
## S3 method for class 'srcfile':
print(x, ...)
## S3 method for class 'srcfile':
open(con, line, ...)
## S3 method for class 'srcfile':
close(con, ...)
## S3 method for class 'srcref':
print(x, useSource = TRUE, ...)
## S3 method for class 'srcref':
as.character(x, useSource = TRUE, ...)
.isOpen(srcfile)
```

Arguments

`filename`	The name of a file
`encoding`	The character encoding to assume for the file
`lines`	A character vector of source lines. Other R objects will be coerced to character.
`srcfile`	A `srcfile` object.
`first, last, line`	Line numbers.

lloc	A vector of four values giving a source location; see 'Details'.
x, con	An object of the appropriate type.
useSource	Whether to read the srcfile to obtain the text of a srcref.
...	Additional arguments to the methods; these will be ignored.

Details

These functions and classes handle source code references.

The srcfile function produces an object of class srcfile, which contains the name and directory of a source code file, along with its timestamp, for use in source level debugging (not yet implemented) and source echoing. The encoding of the file is saved; see file for a discussion of encodings, and iconvlist for a list of allowable encodings on your platform.

The srcfilecopy function produces an object of the descendant class srcfilecopy, which saves the source lines in a character vector.

The getSrcLines function reads the specified lines from srcfile.

The srcref function produces an object of class srcref, which describes a range of characters in a srcfile. The lloc value gives the following values: c(first_line, first_column, last_line, last_column).

Methods are defined for print, open, and close for classes srcfile and srcfilecopy. The open method opens its internal file connection at a particular line; if it was already open, it will be repositioned to that line.

Methods are defined for print and as.character for class srcref. The as.character method will read the associated source file to obtain the text corresponding to the reference. If an error occurs (e.g. the file no longer exists), text like <srcref: "file" chars 1:1 to 2:10> will be returned instead, indicating the line:column ranges of the first and last character.

Lists of srcref objects may be attached to expressions as the "srcref" attribute. (The list of srcref objects should be the same length as the expression.) By default, expressions are printed by print.default using the associated srcref. To see deparsed code instead, call print with argument useSource = FALSE. If a srcref object is printed with useSource = FALSE, the <srcref: ...> record will be printed.

.isOpen is intended for internal use: it checks whether the connection associated with a srcfile object is open.

Value

srcfile returns a srcfile object.

srcfilecopy returns a srcfilecopy object.

getSrcLines returns a character vector of source code lines.

srcref returns a srcref object.

Author(s)

Duncan Murdoch

Examples

```
src <- srcfile(system.file("DESCRIPTION", package = "base"))
getSrcLines(src, 1, 4)
ref <- srcref(src, c(1, 1, 2, 1000))
ref
print(ref, useSource = FALSE)
```

Startup *Initialization at Start of an R Session*

Description

In R, the startup mechanism is as follows.

Unless '`--no-environ`' was given on the command line, R searches for site and user files to process for setting environment variables. The name of the site file is the one pointed to by the environment variable `R_ENVIRON`; if this is unset or empty, '*R_HOME*/etc/Renviron.site' is used (if it exists, which it does not in a 'factory-fresh' installation). The name of the user file can be specified by the `R_ENVIRON_USER` environment variable; if this is unset, the user files searched for are '.Renviron' in the current or in the user's home directory (in that order). See 'Details' for how the files are read.

Then R searches for the site-wide startup profile unless the command line option '`--no-site-file`' was given. The name of this file is taken from the value of the `R_PROFILE` environment variable. If this variable is unset, the default is '*R_HOME*/etc/Rprofile.site', which is used if it exists (which it does not in a 'factory-fresh' installation). This code is sourced into the **base** package. Users need to be careful not to unintentionally overwrite objects in **base**, and it is normally advisable to use `local` if code needs to be executed: see the examples.

Then, unless '`--no-init-file`' was given, R searches for a user profile file. The name of this file can be specified by the `R_PROFILE_USER` environment variable. If this is unset, a file called '.Rprofile' in the current directory or in the user's home directory (in that order) is searched for. The user profile is sourced into the user workspace.

Note that when the site and user profile files are sourced only the **base** package is loaded, so objects in other packages need to be referred to by e.g. `utils::dump.frames` or after explicitly loading the package concerned.

It then loads a saved image of the user workspace from '.RData' if there is one (unless '`--no-restore-data`' or '`--no-restore`' was specified on the command line).

Next, if a function `.First` is found on the search path, it is executed as `.First()`. Finally, function `.First.sys()` in the **base** package is run. This calls `require` to attach the default packages specified by `options("defaultPackages")`. If the **methods** package is included, this will have been attached earlier (by function `.OptRequireMethods()`) so that name space initializations such as those from the user workspace will proceed correctly.

A function `.First` (and `.Last`) can be defined in appropriate '.Rprofile' or 'Rprofile.site' files or have been saved in '.RData'. If you want a different set of packages than the default ones when you start, insert a call to `options` in the '.Rprofile' or 'Rprofile.site' file. For example, `options(defaultPackages = character())` will attach no extra packages on startup (only the `base` package) (or set `R_DEFAULT_PACKAGES=NULL` as an environment variable before running R). Using `options(defaultPackages = "")` or `R_DEFAULT_PACKAGES=""` enforces the R *system* default.

On front-ends which support it, the commands history is read from the file specified by the environment variable `R_HISTFILE` (default '.Rhistory') unless '`--no-restore-history`' was specified (or '`--no-restore`').

The command-line flag '`--vanilla`' implies '`--no-site-file`', '`--no-init-file`', '`--no-restore`' and '`--no-environ`'.

Usage

```
.First <- function() { ..... }

.Rprofile <startup file>
```

Details

Note that there are two sorts of files used in startup: *environment files* which contain lists of environment variables to be set, and *profile files* which contain R code.

Lines in a site or user environment file should be either comment lines starting with `#`, or lines of the form *name*=*value*. The latter sets the environmental variable *name* to *value*, overriding an existing value. If *value* contains an expression of the form `${foo-bar}`, the value is that of the environmental variable `foo` if that exists and is set to a non-empty value, otherwise `bar`. (If it is of the form `${foo}`, the default is `""`.) This construction can be nested, so `bar` can be of the same form (as in `${foo-${bar-blah}}`). Note that the braces are essential: `$HOME` will not be interpreted.

Leading and trailing white space in *value* are stripped. *value* is then processed in a similar way to a Unix shell: in particular the outermost level of (single or double) quotes is stripped, and backslashes are removed except inside quotes.

Note

The file '*R_HOME*/etc/Renviron' is always read very early in the start-up processing. It contains environment variables set by R in the configure process. Values in that file can be overridden in site or user environment files: do not change '*R_HOME*/etc/Renviron' itself. Note that this is distinct from '*R_HOME*/etc/Renviron.site'.

See Also

`.Last` for final actions at the close of an R session. `commandArgs` for accessing the command line arguments.

There are examples of using startup files to set defaults for graphics devices in the help for `X11` and `quartz`.

An Introduction to R for more command-line options: those affecting memory management are covered in the help file for Memory.

For profiling code, see `Rprof`.

Examples

```
## Not run:
## Example ~/.Renviron on Unix
R_LIBS=~/R/library
PAGER=/usr/local/bin/less

## Example .Renviron on Windows
```

```
R_LIBS=C:/R/library
MY_TCLTK="c:/Program Files/Tcl/bin"

## Example of setting R_DEFAULT_PACKAGES (from R CMD check)
R_DEFAULT_PACKAGES='utils,grDevices,graphics,stats'
# this loads the packages in the order given, so they appear on
# the search path in reverse order.

## Example of .Rprofile
options(width=65, digits=5)
options(show.signif.stars=FALSE)
setHook(packageEvent("grDevices", "onLoad"),
        function(...) grDevices::ps.options(horizontal=FALSE))
set.seed(1234)
.First <- function() cat("\n   Welcome to R!\n\n")
.Last <- function()  cat("\n   Goodbye!\n\n")

## Example of Rprofile.site
local({
  # add MASS to the default packages, set a CRAN mirror
  old <- getOption("defaultPackages"); r <- getOption("repos")
  r["CRAN"] <- "http://my.local.cran"
  options(defaultPackages = c(old, "MASS"), repos = r)
})

## if .Renviron contains
FOOBAR="coo\bar"doh\ex"abc\"def'"

## then we get
> cat(Sys.getenv("FOOBAR"), "\n")
coo\bardoh\exabc"def'
## End(Not run)
```

`stop` *Stop Function Execution*

Description

`stop` stops execution of the current expression and executes an error action.

`geterrmessage` gives the last error message.

Usage

```
stop(..., call. = TRUE, domain = NULL)
geterrmessage()
```

Arguments

`...`	zero or more objects which can be coerced to character (and which are pasted together with no separator) or a single condition object.
`call.`	logical, indicating if the call should become part of the error message.
`domain`	see `gettext`. If `NA`, messages will not be translated.

Details

The error action is controlled by error handlers established within the executing code and by the current default error handler set by `options(error=)`. The error is first signaled as if using `signalCondition()`. If there are no handlers or if all handlers return, then the error message is printed (if `options("show.error.messages")` is true) and the default error handler is used. The default behaviour (the `NULL` error-handler) in interactive use is to return to the top level prompt or the top level browser, and in non-interactive use to (effectively) call `q("no", status=1, runLast=FALSE)`. The default handler stores the error message in a buffer; it can be retrieved by `geterrmessage()`. It also stores a trace of the call stack that can be retrieved by `traceback()`.

Errors will be truncated to `getOption("warning.length")` characters, default 1000.

If a condition object is supplied it should be the only argument, and further arguments will be ignored, with a warning.

Value

`geterrmessage` gives the last error message, as a character string ending in `"\n"`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`warning`, `try` to catch errors and retry, and `options` for setting error handlers. `stopifnot` for validity testing. `tryCatch` and `withCallingHandlers` can be used to establish custom handlers while executing an expression.

`gettext` for the mechanisms for the automated translation of messages.

Examples

```
options(error = expression(NULL))
# don't stop on stop(.)  << Use with CARE! >>

iter <- 12
if(iter > 10) stop("too many iterations")

tst1 <- function(...) stop("dummy error")
tst1(1:10, long, calling, expression)

tst2 <- function(...) stop("dummy error", call. = FALSE)
tst2(1:10, longcalling, expression, but.not.seen.in.Error)

options(error = NULL)# revert to default
```

stopifnot *Ensure the 'Truth' of R Expressions*

Description

If any of the expressions in `...` are not `all TRUE`, `stop` is called, producing an error message indicating the *first* of the elements of `...` which were not true.

Usage

```
stopifnot(...)
```

Arguments

`...`	any number of (`logical`) R expressions, which should evaluate to `TRUE`.

Details

This function is intended for use in regression tests or also argument checking of functions, in particular to make them easier to read.

`stopifnot(A, B)` is conceptually equivalent to `{ if(any(is.na(A)) || !all(A)) stop(...) ; if(any(is.na(B)) || !all(B)) stop(...) }`.

Value

(`NULL` if all statements in `...` are `TRUE`.)

See Also

`stop`, `warning`.

Examples

```
stopifnot(1 == 1, all.equal(pi, 3.14159265), 1 < 2) # all TRUE

m <- matrix(c(1,3,3,1), 2,2)
stopifnot(m == t(m), diag(m) == rep(1,2)) # all(.) |=>  TRUE

op <- options(error = expression(NULL))
# "disable stop(.)"  << Use with CARE! >>

stopifnot(all.equal(pi, 3.141593),  2 < 2, all(1:10 < 12), "a" < "b")
stopifnot(all.equal(pi, 3.1415927), 2 < 2, all(1:10 < 12), "a" < "b")

options(op)# revert to previous error handler
```

strptime *Date-time Conversion Functions to and from Character*

Description

Functions to convert between character representations and objects of classes "POSIXlt" and "POSIXct" representing calendar dates and times.

Usage

```
## S3 method for class 'POSIXct':
format(x, format = "", tz = "", usetz = FALSE, ...)
## S3 method for class 'POSIXlt':
format(x, format = "", usetz = FALSE, ...)

## S3 method for class 'POSIXt':
as.character(x, ...)

strftime(x, format="", usetz = FALSE, ...)
strptime(x, format, tz = "")

ISOdatetime(year, month, day, hour, min, sec, tz = "")
ISOdate(year, month, day, hour = 12, min = 0, sec = 0, tz = "GMT")
```

Arguments

x	An object to be converted.
tz	A timezone specification to be used for the conversion. System-specific (see as.POSIXlt), but "" is the current time zone, and "GMT" is UTC.
format	A character string. The default is "%Y-%m-%d %H:%M:%S" if any component has a time component which is not midnight, and "%Y-%m-%d" otherwise. If options("digits.secs") is set, up to the specified number of digits will be printed for seconds.
...	Further arguments to be passed from or to other methods.
usetz	logical. Should the timezone be appended to the output? This is used in printing times, and as a workaround for problems with using "%Z" on most Linux systems.
year, month, day	numerical values to specify a day.
hour, min, sec	numerical values for a time within a day. Fractional seconds are allowed.

Details

The format and as.character methods and strftime convert objects from the classes "POSIXlt" and "POSIXct" (not strftime) to character vectors.

strptime converts character strings to class "POSIXlt": its input x is first coerced to character if necessary. Each string is processed as far as necessary for the format specified: any trailing characters are ignored.

`strftime` is an alias for `format.POSIXlt`, and `format.POSIXct` first converts to class `"POSIXlt"` by calling `as.POSIXlt`. Note that only that conversion depends on the time zone.

The usual vector re-cycling rules are applied to `x` and `format` so the answer will be of length that of the longer of the vectors.

Locale-specific conversions to and from character strings are used where appropriate and available. This affects the names of the days and months, the AM/PM indicator (if used) and the separators in formats such as `%x` and `%X` (via the setting of the `LC_TIME` locale category).

The details of the formats are system-specific, but the following are defined by the ISO C / POSIX standard for `strftime` and are likely to be widely available. A *conversion specification* is introduced by `%`, usually followed by a single letter or `O` or `E` and then a single letter. Any character in the format string not part of a conversion specification is interpreted literally (and `%%` gives `%`). Widely implemented conversion specifications include

`%a` Abbreviated weekday name in the current locale. (Also matches full name on input.)

`%A` Full weekday name in the current locale. (Also matches abbreviated name on input.)

`%b` Abbreviated month name in the current locale. (Also matches full name on input.)

`%B` Full month name in the current locale. (Also matches abbreviated name on input.)

`%c` Date and time, locale-specific.

`%d` Day of the month as decimal number (01–31).

`%H` Hours as decimal number (00–23).

`%I` Hours as decimal number (01–12).

`%j` Day of year as decimal number (001–366).

`%m` Month as decimal number (01–12).

`%M` Minute as decimal number (00–59).

`%p` AM/PM indicator in the locale. Used in conjuction with `%I` and **not** with `%H`. An empty string in some locales.

`%S` Second as decimal number (00–61), allowing for up to two leap-seconds (but POSIX-compliant OSes will ignore leap seconds).

`%U` Week of the year as decimal number (00–53) using Sunday as the first day 1 of the week (and typically with the first Sunday of the year as day 1 of week 1). The US convention.

`%w` Weekday as decimal number (0–6, Sunday is 0).

`%W` Week of the year as decimal number (00–53) using Monday as the first day of week (and typically with the first Monday of the year as day 1 of week 1). The UK convention.

`%x` Date, locale-specific.

`%X` Time, locale-specific.

`%y` Year without century (00–99). If you use this on input, which century you get is system-specific. So don't! Often values up to 68 (or 69) are prefixed by 20 and 69 (or 70) to 99 by 19.

`%Y` Year with century.

`%z` (output only.) Offset from Greenwich, so `-0800` is 8 hours west of Greenwich.

`%Z` (output only.) Time zone as a character string (empty if not available).

Where leading zeros are shown they will be used on output but are optional on input.

Also defined in the current standards but less widely implemented (e.g. not for output on Windows) are

`%C` Century (00–99): the integer part of the year divided by 100.

%D Locale-specific date format such as `%m/%d/%y`: ISO C99 says it should be that exact format.

%e Day of the month as decimal number (1–31), with a leading space for a single-digit number.

%F Equivalent to %Y-%m-%d (the ISO 8601 date format).

%g The last two digits of the week-based year (see `%V`). (Typically accepted but ignored on input.)

%G The week-based year (see `%V`) as a decimal number. (Typically accepted but ignored on input.)

%h Equivalent to `%b`.

%k The 24-hour clock time with single digits preceded by a blank.

%l The 12-hour clock time with single digits preceded by a blank.

%n Newline on output, arbitrary whitespace on input.

%r The 12-hour clock time (using the locale's AM or PM).

%R Equivalent to `%H:%M`.

%t Tab on output, arbitrary whitespace on input.

%T Equivalent to `%H:%M:%S`.

%u Weekday as a decimal number (1–7, Monday is 1).

%V Week of the year as decimal number (00–53) as defined in ISO 8601. If the week (starting on Monday) containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1. (Typically accepted but ignored on input.)

For output (and possibly input) there are also `%O[dHImMUVwWy]` which may emit numbers in an alternative locale-dependent format (e.g. roman numerals), and `%E[cCyYxX]` which can use an alternative 'era' (e.g. a different religious calendar). Which of these are supported is OS-dependent.

Specific to R is `%OSn`, which for output gives the seconds to `0 <= n <= 6` decimal places (and if `%OS` is not followed by a digit, it uses the setting of `getOption("digits.secs")`, or if that is unset, `n = 3`). Further, for `strptime` `%OS` will input seconds including fractional seconds. Note that `%S` ignores (and not rounds) fractional parts on output.

The behaviour of other conversion specifications (and even if other character sequences commencing with `%` *are* conversion specifications) is system-specific.

`ISOdatetime` and `ISOdate` are convenience wrappers for `strptime`, that differ only in their defaults and that `ISOdate` sets a timezone. (For dates without times it would be better to use the `"Date"` class.)

Value

The `format` methods and `strftime` return character vectors representing the time. `NA` times are returned as `NA_character_`.

`strptime` turns character representations into an object of class `"POSIXlt"`. The timezone is used to set the `isdst` component and to set the `"tzone"` attribute if `tz != ""`.

`ISOdatetime` and `ISOdate` return an object of class `"POSIXct"`.

Note

The default formats follow the rules of the ISO 8601 international standard which expresses a day as `"2001-02-28"` and a time as `"14:01:02"` using leading zeroes as here. The ISO form uses no space to separate dates and times.

If the date string does not specify the date completely, the returned answer may be system-specific. The most common behaviour is to assume that unspecified seconds, minutes or hours are zero,

and a missing year, month or day is the current one. If it specifies a date incorrectly, reliable implementations will give an error and the date is reported as NA. Unfortunately some common implementations (such as 'glibc') are unreliable and guess at the intended meaning.

If the timezone specified is invalid on your system, what happens is system-specific but it will probably be ignored.

OS facilities will probably not print years before 1CE (aka 1AD) correctly.

Remember that in most timezones some times do not occur and some occur twice because of transitions to/from summer time. What happens in those cases is OS-specific.

References

International Organization for Standardization (2004, 1988, 1997, ...) *ISO 8601. Data elements and interchange formats – Information interchange – Representation of dates and times.* For links to versions available on-line see (at the time of writing) http://www.qsl.net/g1smd/isopdf.htm; for information on the current official version, see http://www.iso.org/iso/en/prods-services/popstds/datesandtime.html.

See Also

DateTimeClasses for details of the date-time classes; locales to query or set a locale.

Your system's help pages on strftime and strptime to see how to specify their formats. (On some systems strptime is replaced by corrected code from 'glibc', when all the conversion specifications described here are supported, but with no alternative number representation nor era available in any locale.)

Examples

```
## locale-specific version of date()
format(Sys.time(), "%a %b %d %X %Y %Z")

## time to sub-second accuracy (if supported by the OS)
format(Sys.time(), "%H:%M:%OS3")

## read in date info in format 'ddmmmyyyy'
## This will give NA(s) in some locales; setting the C locale
## as in the commented lines will overcome this on most systems.
## lct <- Sys.getlocale("LC_TIME"); Sys.setlocale("LC_TIME", "C")
x <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
z <- strptime(x, "%d%b%Y")
## Sys.setlocale("LC_TIME", lct)
z

## read in date/time info in format 'm/d/y h:m:s'
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
times <- c("23:03:20", "22:29:56", "01:03:30", "18:21:03", "16:56:26")
x <- paste(dates, times)
strptime(x, "%m/%d/%y %H:%M:%S")

## time with fractional seconds
z <- strptime("20/2/06 11:16:16.683", "%d/%m/%y %H:%M:%OS")
z # prints without fractional seconds
op <- options(digits.secs=3)
z
options(op)
```

```
## timezones are not portable, but 'EST5EDT' comes pretty close.
(x <- strptime(c("2006-01-08 10:07:52", "2006-08-07 19:33:02"),
               "%Y-%m-%d %H:%M:%S", tz="EST5EDT"))
attr(x, "tzone")
```

`strsplit` *Split the Elements of a Character Vector*

Description

Split the elements of a character vector `x` into substrings according to the presence of substring `split` within them.

Usage

```
strsplit(x, split, extended = TRUE, fixed = FALSE, perl = FALSE)
```

Arguments

`x`	character vector, each element of which is to be split. Other inputs, including a factor, will give an error.
`split`	character vector (or object which can be coerced to such) containing regular expression(s) (unless `fixed = TRUE`) to use for splitting. If empty matches occur, in particular if `split` has length 0, `x` is split into single characters. If `split` has length greater than 1, it is re-cycled along `x`.
`extended`	logical. If `TRUE` (the default), extended regular expression matching is used, and if `FALSE` basic regular expressions are used.
`fixed`	logical. If `TRUE` match `split` exactly, otherwise use regular expressions. Has priority over `perl` and `extended`.
`perl`	logical. Should perl-compatible regexps be used? Has priority over `extended`.

Details

Argument `split` will be coerced to character, so you will see uses with `split = NULL` to mean `split = character(0)`, including in the examples below.

Note that splitting into single characters can be done *via* `split=character(0)` or `split=""`; the two are equivalent. The definition of 'character' here depends on the locale (and perhaps OS): in a single-byte locale it is a byte, and in a multi-byte locale it is the unit represented by a 'wide character' (almost always a Unicode point).

A missing value of `split` does not split the corresponding element(s) of `x` at all.

The algorithm applied to each input string is

```
repeat {
    if the string is empty
        break.
    if there is a match
        add the string to the left of the match to the output.
        remove the match and all to the left of it.
    else
```

```
            add the string to the output.
            break.
    }
```

Note that this means that if there is a match at the beginning of a (non-empty) string, the first element of the output is `""`, but if there is a match at the end of the string, the output is the same as with the match removed.

Value

A list of length `length(x)` the `i`-th element of which contains the vector of splits of `x[i]`.

If `fixed = TRUE` or `perl = TRUE` and if any element of `x` or `split` is declared to be in UTF-8 (see `Encoding`, non-ASCII character strings in the result will be in UTF-8 and have the encoding declared as UTF-8. Otherwise they will be in the current locale's encoding, and be declared to have the encoding of the current locale if either Latin-1 or UTF-8 and the corresponding input had a declared encoding.

Warning

The standard regular expression code has been reported to be very slow when applied to extremely long character strings (tens of thousands of characters or more): the code used when `perl = TRUE` seems much faster and more reliable for such usages.

The `perl = TRUE` option is only implemented for single-byte and UTF-8 encodings, and will warn if used in a non-UTF-8 multibyte locale.

See Also

`paste` for the reverse, `grep` and `sub` for string search and manipulation; further `nchar`, `substr`.

'regular expression' for the details of the pattern specification.

Examples

```
noquote(strsplit("A text I want to display with spaces", NULL)[[1]])

x <- c(as = "asfef", qu = "qwerty", "yuiop[", "b", "stuff.blah.yech")
# split x on the letter e
strsplit(x,"e")

unlist(strsplit("a.b.c", "."))
## [1] "" "" "" "" ""
## Note that 'split' is a regexp!
## If you really want to split on '.', use
unlist(strsplit("a.b.c", "\\."))
## [1] "a" "b" "c"
## or
unlist(strsplit("a.b.c", ".", fixed = TRUE))

## a useful function: rev() for strings
strReverse <- function(x)
        sapply(lapply(strsplit(x, NULL), rev), paste, collapse="")
strReverse(c("abc", "Statistics"))

## get the first names of the members of R-core
```

```
a <- readLines(file.path(R.home("doc"),"AUTHORS"))[-(1:8)]
a <- a[(0:2)-length(a)]
(a <- sub(" .*","", a))
# and reverse them
strReverse(a)

## Note that final empty strings are not produced:
strsplit(paste(c("", "a", ""), collapse="#"), split="#")[[1]]
# [1] ""  "a"
## and also an empty string is only produced before a definite match:
strsplit("", " ")[[1]]    # character(0)
strsplit(" ", " ")[[1]]   # [1] ""
```

`strtrim` *Trim Character Strings to Specified Widths*

Description

Trim character strings to specified display widths.

Usage

```
strtrim(x, width)
```

Arguments

`x`	a character vector, or an object which can be coerced to a character vector by `as.character`.
`width`	Positive integer values: recycled to the length of `x`.

Details

'Width' is interpreted as the display width in a monospaced font. What happens with non-printable characters (such as backspace, tab) is implementation-dependent and may depend on the locale (e.g. they may be included in the count or they may be omitted).

Using this function rather than `substr` is important when there might be double-width characters in character vectors

Value

A character vector of the same length and with the same attributes as `x` (after possible coercion).

Elements of the result will be have the encoding declared as that of the current locale (see `Encoding` if the corresponding input had a declared encoding and the current locale is either Latin-1 or UTF-8.

Examples

```
strtrim(c("abcdef", "abcdef", "abcdef"), c(1,5,10))
```

`structure` *Attribute Specification*

Description

`structure` returns the given object with further attributes set.

Usage

```
structure(.Data, ...)
```

Arguments

`.Data`	an object which will have various attributes attached to it.
`...`	attributes, specified in `tag=value` form, which will be attached to data.

Details

Adding a class `"factor"` will ensure that numeric codes are given integer storage mode.

For historical reasons (these names are used when deparsing), attributes `".Dim"`, `".Dimnames"`, `".Names"`, `".Tsp"` and `".Label"` are renamed to `"dim"`, `"dimnames"`, `"names"`, `"tsp"` and `"levels"`.

It is possible to give the same tag more than once, in which case the last value assigned wins. As with other ways of assigning attributes, using `tag=NULL` removes attribute `tag` from `.Data` if it is present.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`attributes`, `attr`.

Examples

```
structure(1:6, dim = 2:3)
```

strwrap *Wrap Character Strings to Format Paragraphs*

Description

Each character string in the input is first split into paragraphs (or lines containing whitespace only). The paragraphs are then formatted by breaking lines at word boundaries. The target columns for wrapping lines and the indentation of the first and all subsequent lines of a paragraph can be controlled independently.

Usage

```
strwrap(x, width = 0.9 * getOption("width"), indent = 0,
        exdent = 0, prefix = "", simplify = TRUE)
```

Arguments

x	a character vector, or an object which can be converted to a character vector by as.character.
width	a positive integer giving the target column for wrapping lines in the output.
indent	a non-negative integer giving the indentation of the first line in a paragraph.
exdent	a non-negative integer specifying the indentation of subsequent lines in paragraphs.
prefix	a character string to be used as prefix for each line.
simplify	a logical. If TRUE, the result is a single character vector of line text; otherwise, it is a list of the same length as x the elements of which are character vectors of line text obtained from the corresponding element of x. (Hence, the result in the former case is obtained by unlisting that of the latter.)

Details

Whitespace (space, tab or newline characters) in the input is destroyed. Double spaces after periods (thought as representing sentence ends) are preserved. Currently, possible sentence ends at line breaks are not considered specially.

Indentation is relative to the number of characters in the prefix string.

Value

A character vector in the current locale's encoding.

Examples

```
## Read in file 'THANKS'.
x <- paste(readLines(file.path(R.home("doc"), "THANKS")), collapse = "\n")
## Split into paragraphs and remove the first three ones
x <- unlist(strsplit(x, "\n[ \t\n]*\n"))[-(1:3)]
## Join the rest
x <- paste(x, collapse = "\n\n")
## Now for some fun:
writeLines(strwrap(x, width = 60))
```

```
writeLines(strwrap(x, width = 60, indent = 5))
writeLines(strwrap(x, width = 60, exdent = 5))
writeLines(strwrap(x, prefix = "THANKS> "))

## Note that messages are wrapped AT the target column indicated by
## 'width' (and not beyond it).
## From an R-devel posting by J. Hosking <jh910@juno.com>.
x <- paste(sapply(sample(10, 100, replace=TRUE),
           function(x) substring("aaaaaaaaaa", 1, x)), collapse = " ")
sapply(10:40,
       function(m)
       c(target = m, actual = max(nchar(strwrap(x, m)))))
```

`subset` *Subsetting Vectors, Matrices and Data Frames*

Description

Return subsets of vectors, matrices or data frames which meet conditions.

Usage

```
subset(x, ...)

## Default S3 method:
subset(x, subset, ...)

## S3 method for class 'matrix':
subset(x, subset, select, drop = FALSE, ...)

## S3 method for class 'data.frame':
subset(x, subset, select, drop = FALSE, ...)
```

Arguments

`x`	object to be subsetted.
`subset`	logical expression indicating elements or rows to keep: missing values are taken as false.
`select`	expression, indicating columns to select from a data frame.
`drop`	passed on to `[` indexing operator.
`...`	further arguments to be passed to or from other methods.

Details

This is a generic function, with methods supplied for matrices, data frames and vectors (including lists). Packages and users can add further methods.

For ordinary vectors, the result is simply `x[subset & !is.na(subset)]`.

For data frames, the `subset` argument works on the rows. Note that `subset` will be evaluated in the data frame, so columns can be referred to (by name) as variables in the expression (see the examples).

The `select` argument exists only for the methods for data frames and matrices. It works by first replacing column names in the selection expression with the corresponding column numbers in the data frame and then using the resulting integer vector to index the columns. This allows the use of the standard indexing conventions so that for example ranges of columns can be specified easily, or single columns can be dropped (see the examples).

The `drop` argument is passed on to the indexing method for matrices and data frames: note that the default for matrices is different from that for indexing.

Value

An object similar to `x` contain just the selected elements (for a vector), rows and columns (for a matrix or data frame), and so on.

Author(s)

Peter Dalgaard and Brian Ripley

See Also

`[`, `transform`

Examples

```
subset(airquality, Temp > 80, select = c(Ozone, Temp))
subset(airquality, Day == 1, select = -Temp)
subset(airquality, select = Ozone:Wind)

with(airquality, subset(Ozone, Temp > 80))

## sometimes requiring a logical 'subset' argument is a nuisance
nm <- rownames(state.x77)
start_with_M <- nm %in% grep("^M", nm, value=TRUE)
subset(state.x77, start_with_M, Illiteracy:Murder)
```

`substitute` *Substituting and Quoting Expressions*

Description

`substitute` returns the parse tree for the (unevaluated) expression `expr`, substituting any variables bound in `env`.

`quote` simply returns its argument. The argument is not evaluated and can be any R expression.

Usage

```
substitute(expr, env)
quote(expr)
```

Arguments

`expr`	Any syntactically valid R expression
`env`	An environment or a list object. Defaults to the current evaluation environment.

Details

The typical use of `substitute` is to create informative labels for data sets and plots. The `myplot` example below shows a simple use of this facility. It uses the functions `deparse` and `substitute` to create labels for a plot which are character string versions of the actual arguments to the function `myplot`.

Substitution takes place by examining each component of the parse tree as follows: If it is not a bound symbol in `env`, it is unchanged. If it is a promise object, i.e., a formal argument to a function or explicitly created using `delayedAssign()`, the expression slot of the promise replaces the symbol. If it is an ordinary variable, its value is substituted, unless `env` is `.GlobalEnv` in which case the symbol is left unchanged.

`substitute` is a primitive function so positional matching is used and names of supplied arguments are ignored.

Value

The `mode` of the result is generally `"call"` but may in principle be any type. In particular, single-variable expressions have mode `"name"` and constants have the appropriate base mode.

Note

Substitute works on a purely lexical basis. There is no guarantee that the resulting expression makes any sense.

Substituting and quoting often causes confusion when the argument is `expression(...)`. The result is a call to the `expression` constructor function and needs to be evaluated with `eval` to give the actual expression object.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`missing` for argument 'missingness', `bquote` for partial substitution, `sQuote` and `dQuote` for adding quotation marks to strings.

Examples

```
require(graphics)
(s.e <- substitute(expression(a + b), list(a = 1)))  #> expression(1 + b)
(s.s <- substitute( a + b,            list(a = 1)))  #> 1 + b
c(mode(s.e), typeof(s.e)) #  "call", "language"
c(mode(s.s), typeof(s.s)) #   (the same)
# but:
(e.s.e <- eval(s.e))          #>  expression(1 + b)
c(mode(e.s.e), typeof(e.s.e)) #  "expression", "expression"

substitute(x <- x + 1, list(x=1)) # nonsense

myplot <- function(x, y)
    plot(x, y, xlab=deparse(substitute(x)),
         ylab=deparse(substitute(y)))
```

```
## Simple examples about lazy evaluation, etc:

f1 <- function(x, y = x)             { x <- x + 1; y }
s1 <- function(x, y = substitute(x)) { x <- x + 1; y }
s2 <- function(x, y) { if(missing(y)) y <- substitute(x); x <- x + 1; y }
a <- 10
f1(a)# 11
s1(a)# 11
s2(a)# a
typeof(s2(a))# "symbol"
```

`substr` *Substrings of a Character Vector*

Description

Extract or replace substrings in a character vector.

Usage

```
substr(x, start, stop)
substring(text, first, last = 1000000)
substr(x, start, stop) <- value
substring(text, first, last = 1000000) <- value
```

Arguments

`x, text`	a character vector.
`start, first`	integer. The first element to be replaced.
`stop, last`	integer. The last element to be replaced.
`value`	a character vector, recycled if necessary.

Details

`substring` is compatible with S, with `first` and `last` instead of `start` and `stop`. For vector arguments, it expands the arguments cyclically to the length of the longest *provided* none are of zero length.

When extracting, if `start` is larger than the string length then `""` is returned.

For the extraction functions, `x` or `text` will be converted to a character vector by `as.character` if it is not already one.

For the replacement functions, if `start` is larger than the string length then no replacement is done. If the portion to be replaced is longer than the replacement string, then only the portion the length of the string is replaced.

If any argument is an `NA` element, the corresponding element of the answer is `NA`.

Elements of the result will be have the encoding declared as that of the current locale (see `Encoding` if the corresponding input had a declared encoding and the current locale is either Latin-1 or UTF-8.

Value

For `substr`, a character vector of the same length and with the same attributes as `x` (after possible coercion).

For `substring`, a character vector of length the longest of the arguments. This will have names taken from `x` (if it has any after coercion, repeated as needed), and other attributes copied from `x` if it is the longest of the arguments).

Elements of `x` with a declared encoding (see `Encoding`) will be returned with the same encoding.

Note

The S4 version of `substring<-` ignores `last`; this version does not.

These functions are often used with `nchar` to truncate a display. That does not really work (you want to limit the width, not the number of characters, so it would be better to use `strtrim`), but at least make sure you use `nchar(type="c")` (the default since R 2.6.0).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`substring`.)

See Also

`strsplit`, `paste`, `nchar`.

Examples

```
substr("abcdef",2,4)
substring("abcdef",1:6,1:6)
## strsplit is more efficient ...

substr(rep("abcdef",4),1:4,4:5)
x <- c("asfef", "qwerty", "yuiop[", "b", "stuff.blah.yech")
substr(x, 2, 5)
substring(x, 2, 4:6)

substring(x, 2) <- c("..", "+++")
x
```

`sum` *Sum of Vector Elements*

Description

`sum` returns the sum of all the values present in its arguments.

Usage

```
sum(..., na.rm = FALSE)
```

Arguments

`...`	numeric or complex or logical vectors.
`na.rm`	logical. Should missing values be removed?

Details

This is a generic function: methods can be defined for it directly or via the `Summary` group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

If `na.rm` is `FALSE` an `NA` value in any of the arguments will cause a value of `NA` to be returned, otherwise `NA` values are ignored.

Logical true values are regarded as one, false values as zero. For historical reasons, `NULL` is accepted and treated as if it were `integer(0)`.

Value

The sum. If all of `...` are of type integer or logical, then the sum is integer, and in that case the result will be `NA` (with a warning) if integer overflow occurs. Otherwise it is a length-one numeric or complex vector.

NB: the sum of an empty set is zero, by definition.

S4 methods

This is part of the S4 `Summary` group generic. Methods for it must use the signature `x, ..., na.rm`.

'plotmath' for the use of `sum` in plot annotation.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`colSums` for row and column sums.

`summary` *Object Summaries*

Description

`summary` is a generic function used to produce result summaries of the results of various model fitting functions. The function invokes particular `methods` which depend on the `class` of the first argument.

Usage

```
summary(object, ...)

## Default S3 method:
summary(object, ..., digits = max(3, getOption("digits")-3))
## S3 method for class 'data.frame':
summary(object, maxsum = 7,
        digits = max(3, getOption("digits")-3), ...)

## S3 method for class 'factor':
summary(object, maxsum = 100, ...)

## S3 method for class 'matrix':
summary(object, ...)
```

Arguments

`object`	an object for which a summary is desired.
`maxsum`	integer, indicating how many levels should be shown for `factor`s.
`digits`	integer, used for number formatting with `signif()` (for `summary.default`) or `format()` (for `summary.data.frame`).
`...`	additional arguments affecting the summary produced.

Details

For `factor`s, the frequency of the first `maxsum - 1` most frequent levels is shown, where the less frequent levels are summarized in `"(Others)"` (resulting in `maxsum` frequencies).

The functions `summary.lm` and `summary.glm` are examples of particular methods which summarize the results produced by `lm` and `glm`.

Value

The form of the value returned by `summary` depends on the class of its argument. See the documentation of the particular methods for details of what is produced by that method.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`anova`, `summary.glm`, `summary.lm`.

Examples

```
summary(attenu, digits = 4) #-> summary.data.frame(...), default precision
summary(attenu $ station, maxsum = 20) #-> summary.factor(...)

lst <- unclass(attenu$station) > 20 # logical with NAs
## summary.default() for logicals -- different from *.factor:
summary(lst)
summary(as.factor(lst))
```

`svd` *Singular Value Decomposition of a Matrix*

Description

Compute the singular-value decomposition of a rectangular matrix.

Usage

```
svd(x, nu = min(n, p), nv = min(n, p), LINPACK = FALSE)

La.svd(x, nu = min(n, p), nv = min(n, p))
```

Arguments

`x`	a real or complex matrix whose SVD decomposition is to be computed.
`nu`	the number of left singular vectors to be computed. This must between `0` and `n = nrow(x)`.
`nv`	the number of right singular vectors to be computed. This must be between `0` and `p = ncol(x)`.
`LINPACK`	logical. Should LINPACK be used (for compatibility with R < 1.7.0)? In this case `nu` must be `0`, `nrow(x)` or `ncol(x)`.

Details

The singular value decomposition plays an important role in many statistical techniques. `svd` and `La.svd` provide two slightly different interfaces. The main functions used are the LAPACK routines DGESDD and ZGESVD; `svd(LINPACK = TRUE)` provides an interface to the LINPACK routine DSVDC, purely for backwards compatibility.

Computing the singular vectors is the slow part for large matrices. The computation will be more efficient if `nu <= min(n, p)` and `nv <= min(n, p)`, and even more efficient if one or both are zero.

Unsuccessful results from the underlying LAPACK code will result in an error giving a positive error code: these can only be interpreted by detailed study of the FORTRAN code.

Value

The SVD decomposition of the matrix as computed by LAPACK/LINPACK,

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}',$$

where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal, $\boldsymbol{V}'$ means *V transposed*, and $\boldsymbol{D}$ is a diagonal matrix with the singular values D_{ii}. Equivalently, $\boldsymbol{D} = \boldsymbol{U}'\boldsymbol{X}\boldsymbol{V}$, which is verified in the examples, below.

The returned value is a list with components

`d`	a vector containing the singular values of `x`, of length `min(n, p)`.
`u`	a matrix whose columns contain the left singular vectors of `x`, present if `nu > 0`. Dimension `c(n, nu)`.
`v`	a matrix whose columns contain the right singular vectors of `x`, present if `nv > 0`. Dimension `c(p, nv)`.

For `La.svd` the return value replaces `v` by `vt`, the (conjugated if complex) transpose of `v`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at http://www.netlib.org/lapack/lug/lapack_lug.html.

See Also

eigen, qr.

Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
X <- hilbert(9)[,1:6]
(s <- svd(X))
D <- diag(s$d)
s$u %*% D %*% t(s$v) #  X = U D V'
t(s$u) %*% X %*% s$v #  D = U' X V
```

---

sweep *Sweep out Array Summaries*

Description

Return an array obtained from an input array by sweeping out a summary statistic.

Usage

```
sweep(x, MARGIN, STATS, FUN="-", check.margin=TRUE, ...)
```

Arguments

x	an array.
MARGIN	a vector of indices giving the extents of x which correspond to STATS.
STATS	the summary statistic which is to be swept out.
FUN	the function to be used to carry out the sweep. In the case of binary operators such as "/" etc., the function name must backquoted or quoted. (FUN is found by a call to match.fun.)
check.margin	logical. If TRUE (the default), warn if the length or dimensions of STATS do not match the specified dimensions of x. Set to FALSE for a small speed gain when you *know* that dimensions match.
...	optional arguments to FUN.

Details

The consistency check among STATS, MARGIN and x is stricter if STATS is an array than if it is a vector. In the vector case, some kinds of recycling are allowed without a warning. Use sweep(x,MARGIN,as.array(STATS)) if STATS is a vector and you want to be warned if any recycling occurs.

Value

An array with the same shape as `x`, but with the summary statistics swept out.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`apply` on which `sweep` used to be based; `scale` for centering and scaling.

Examples

```
require(stats) # for median
med.att <- apply(attitude, 2, median)
sweep(data.matrix(attitude), 2, med.att)# subtract the column medians

## More sweeping:
A <- array(1:24, dim = 4:2)

## no warnings in normal use
sweep(A, 1, 5)
(A.min <- apply(A, 1, min)) # == 1:4
sweep(A, 1, A.min)
sweep(A, 1:2, apply(A, 1:2, median))

## warnings when mismatch
sweep(A, 1, 1:3)## STATS does not recycle
sweep(A, 1, 6:1)## STATS is longer

## exact recycling:
sweep(A, 1, 1:2)## no warning
sweep(A, 1, as.array(1:2))## warning
```

`switch` *Select One of a List of Alternatives*

Description

`switch` evaluates `EXPR` and accordingly chooses one of the further arguments (in `...`).

Usage

```
switch(EXPR, ...)
```

Arguments

`EXPR`	an expression evaluating to a number or a character string.
`...`	the list of alternatives, given explicitly.

Details

If the value of EXPR is an integer between 1 and nargs()-1 then the corresponding element of ... is evaluated and the result returned.

If EXPR returns a character string then that string is used to match the names of the elements in .... If there is an exact match then that element is evaluated and returned if there is one, otherwise the next element is chosen, e.g., switch("cc", a=1, cc=, d=2) evaluates to 2.

In the case of no match, if there's a further argument in switch that one is returned, otherwise NULL.

Warning

Beware of partial matching: an alternative E = foo will match the first argument EXPR unless that is named. See the examples for good practice in naming the first argument.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
require(stats)
centre <- function(x, type) {
  switch(type,
         mean = mean(x),
         median = median(x),
         trimmed = mean(x, trim = .1))
}
x <- rcauchy(10)
centre(x, "mean")
centre(x, "median")
centre(x, "trimmed")

ccc <- c("b","QQ","a","A","bb")
for(ch in ccc)
    cat(ch,":",switch(EXPR = ch, a=1,     b=2:3), "\n")
for(ch in ccc)
    cat(ch,":",switch(EXPR = ch, a=, A=1, b=2:3, "Otherwise: last"),"\n")

## Numeric EXPR don't allow an 'otherwise':
for(i in c(-1:3,9))  print(switch(i, 1,2,3,4))
```

Syntax *Operator Syntax and Precedence*

Description

Outlines R syntax and gives the precedence of operators

Details

The following unary and binary operators are defined. They are listed in precedence groups, from highest to lowest.

`[ [[`	indexing
`::  :::`	access variables in a name space
`$ @`	component / slot extraction
`^`	exponentiation (right to left)
`- +`	unary minus and plus
`:`	sequence operator
`%any%`	special operators
`* /`	multiply, divide
`+ -`	(binary) add, subtract
`< > <= >= == !=`	ordering and comparison
`!`	negation
`& &&`	and
`| ||`	or
`~`	as in formulae
`-> ->>`	rightwards assignment
`=`	assignment (right to left)
`<- <<-`	assignment (right to left)
`?`	help (unary and binary)

Within an expression operators of equal precedence are evaluated from left to right except where indicated.

The links in the **See Also** section cover most other aspects of the basic syntax.

Note

There are substantial precedence differences between R and S. In particular, in S `?` has the same precedence as (binary) `+ -` and `& && | ||` have equal precedence.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`Arithmetic`, `Comparison`, `Control`, `Extract`, `Logic`, `NumericConstants`, `Paren`, `Quotes`, `Reserved`.

The *R Language Definition* manual.

`Sys.getenv` *Get Environment Variables*

Description

`Sys.getenv` obtains the values of the environment variables.

Usage

```
Sys.getenv(x = NULL, unset = "")
```

Arguments

x	a character vector, or NULL.
unset	a character string.

Details

Both arguments will be coerced to character if necessary.

Setting unset = NA will enable unset variables and those set to the value "" to be distinguished, *if the OS does*.

Value

A vector of the same length as x, with the variable names as its names attribute. Each element holds the value of the environment variable named by the corresponding component of x (or the value of unset if no environment variable with that name was found).

On most platforms Sys.getenv() will return a named vector giving the values of all the environment variables, sorted in the current locale. It may be confused by names containing = which some platforms allow but POSIX does not.

See Also

Sys.setenv, Sys.getlocale for the locale in use, getwd for the working directory.

The help for 'environment variables' lists many of the environment variables used by R.

Examples

```
## whether HOST is set will be shell-dependent e.g. Solaris' csh does not.
Sys.getenv(c("R_HOME", "R_PAPERSIZE", "R_PRINTCMD", "HOST"))

names(s <- Sys.getenv()) # all settings (the values could be very long)

## Language and Locale settings -- but rather use Sys.getlocale()
s[grep("^L(C|ANG)", names(s))]
```

Sys.glob *Wildcard Expansion on File Paths*

Description

Function to do wildcard expansion (also known as 'globbing') on file paths.

Not all platforms support this.

Usage

```
Sys.glob(paths, dirmark = FALSE)
```

Arguments

paths	character vector of patterns for relative or absolute filepaths. Missing values will be ignored.
dirmark	logical: should matches to directories from patterns that do not already end in / have a slash appended? May not be supported on all platforms.

Details

This expands wildcards in file paths. For precise details, see your system's documentation on the `glob` system call. There is a POSIX 1003.2 standard (see http://www.opengroup.org/onlinepubs/009695399/functions/glob.html) but some OSes will go beyond this (in particular some BSD-based OSes also do tilde expansion, see `path.expand`).

All systems should interpret `*` (match zero or more characters), `?` (match a single character) and `[` (begin a character class or range). If a filename starts with `.` this must be matched explicitly. By default paths ending in `/` will be accepted and matched only to directories.

The rest of these details are indicative (and based on the POSIX standard).

`[` begins a character class. If the first character in `[...]` is not `!`, this is a character class which matches a single character against any of the characters specified. The class cannot be empty, so `]` can be included provided it is first. If the first character is `!`, the character class matches a single character which is *none* of the specified characters.

Character classes can include ranges such as `[A-Z]`: include `-` as a character by having it first or last in a class. (The interpretation of ranges should be locale-specific, so the example is not a good idea in an Estonian locale.)

One can remove the special meaning of `?`, `*` and `[` by preceding them by a backslash (except within a character class).

Value

A character vector of matched file paths. The order is system-specific (but in the order of the elements of `paths`): it is normally collated in either the current locale or in byte (ASCII) order.

Directory errors are normally ignored, so the matches are to accessible file paths (but not necessarily accessible files).

On platforms which do not have the `glob` system call (nor, as for R under Windows, an emulation), `paths` is returned unchanged.

See Also

`path.expand`.

Quotes for handling backslashes in character strings.

Examples

```
## Not run:
Sys.glob(file.path(R.home(), "library", "*", "R", "*.rdx"))
## End(Not run)
```

`Sys.info` *Extract System and User Information*

Description

Reports system and user information.

Usage

```
Sys.info()
```

Details

This function is not implemented on all R platforms, and returns `NULL` when not available. Where possible it is based on POSIX system calls.

`Sys.info()` returns details of the platform R is running on, whereas `R.version` gives details of the platform R was built on: they may well be different.

Value

A character vector with fields

`sysname`	The operating system.
`release`	The OS release.
`version`	The OS version.
`nodename`	A name by which the machine is known on the network (if any).
`machine`	A concise description of the hardware.
`login`	The user's login name, or `"unknown"` if it cannot be ascertained.
`user`	The name of the real user ID, or `"unknown"` if it cannot be ascertained.

The first five fields come from the `uname(2)` system call. The login name comes from `getlogin(2)`, and the user name from `getpwuid(getuid())`

Note

The meaning of OS 'release' and 'version' is system-dependent and there is no guarantee that the node or login or user names will be what you might reasonably expect. (In particular on some Linux distributions the login name is unknown from sessions with re-directed inputs.)

See Also

`.Platform`, and `R.version`. `sessionInfo()` gives a synopsis of both your system and the R session.

Examples

```
Sys.info()
## An alternative (and probably better) way to get the login name on Unix
Sys.getenv("LOGNAME")
```

---

sys.parent *Functions to Access the Function Call Stack*

Description

These functions provide access to `environment`s ('frames' in S terminology) associated with functions further up the calling stack.

Usage

```
sys.call(which = 0)
sys.frame(which = 0)
sys.nframe()
sys.function(which = 0)
sys.parent(n = 1)

sys.calls()
sys.frames()
sys.parents()
sys.on.exit()
sys.status()
parent.frame(n = 1)
```

Arguments

`which`	the frame number if non-negative, the number of frames to go back if negative.
`n`	the number of generations to go back. (See the 'Details' section.)

Details

`.GlobalEnv` is given number 0 in the list of frames. Each subsequent function evaluation increases the frame stack by 1 and the call, function definition and the environment for evaluation of that function are returned by `sys.call`, `sys.function` and `sys.frame` with the appropriate index.

`sys.call`, `sys.frame` and `sys.function` accept integer values for the argument `which`. Non-negative values of `which` are frame numbers whereas negative values are counted back from the frame number of the current evaluation.

The parent frame of a function evaluation is the environment in which the function was called. It is not necessarily numbered one less than the frame number of the current evaluation, nor is it the environment within which the function was defined. `sys.parent` returns the number of the parent frame if `n` is 1 (the default), the grandparent if `n` is 2, and so on. See also the Note.

`sys.nframe` returns an integer, the number of the current frame as described in the first paragraph.

`sys.calls` and `sys.frames` give a pairlist of all the active calls and frames, respectively, and `sys.parents` returns an integer vector of indices of the parent frames of each of those frames.

Notice that even though the `sys.`*xxx* functions (except `sys.status`) are interpreted, their contexts are not counted nor are they reported. There is no access to them.

`sys.status()` returns a list with components `sys.calls`, `sys.parents` and `sys.frames`, the results of calls to those three functions (which this will include the call to `sys.status`: see the first example).

`sys.on.exit()` returns the expression stored for use by `on.exit` in the function currently being evaluated. (Note that this differs from S, which returns a list of expressions for the current frame and its parents.)

`parent.frame(n)` is a convenient shorthand for `sys.frame(sys.parent(n))` (implemented slightly more efficiently).

Value

`sys.call` returns a call, `sys.function` a function definition, and `sys.frame` and `parent.frame` return an environment.

For the other functions, see the 'Details' section.

Note

Strictly, `sys.parent` and `parent.frame` refer to the *context* of the parent interpreted function. So internal functions (which may or may not set contexts and so may or may not appear on the call stack) are not counted, and S3 methods can also do surprising things.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (not `parent.frame`.)

See Also

`eval` for a usage of `sys.frame` and `parent.frame`.

Examples

```
require(utils)

## Note: the first two examples will give different results
## if run by example().
ff <- function(x) gg(x)
gg <- function(y) sys.status()
str(ff(1))

gg <- function(y) {
    ggg <- function() {
        cat("current frame is", sys.nframe(), "\n")
        cat("parents are", sys.parents(), "\n")
        print(sys.function(0)) # ggg
        print(sys.function(2)) # gg
    }
    if(y > 0) gg(y-1) else ggg()
}
gg(3)

t1 <- function() {
  aa <- "here"
  t2 <- function() {
    ## in frame 2 here
    cat("current frame is", sys.nframe(), "\n")
    str(sys.calls()) ## list with two components t1() and t2()
    cat("parents are frame numbers", sys.parents(), "\n") ## 0 1
    print(ls(envir=sys.frame(-1))) ## [1] "aa" "t2"
    invisible()
  }
  t2()
}
t1()

test.sys.on.exit <- function() {
  on.exit(print(1))
  ex <- sys.on.exit()
  str(ex)
```

```
  cat("exiting...\n")
}
test.sys.on.exit()
## gives 'language print(1)', prints 1 on exit

## An example where the parent is not the next frame up the stack
## since method dispatch uses a frame.
as.double.foo <- function(x)
{
    str(sys.calls())
    print(sys.frames())
    print(sys.parents())
    print(sys.frame(-1)); print(parent.frame())
    x
}
t2 <- function(x) as.double(x)
a <- structure(pi, class = "foo")
t2(a)
```

`Sys.setenv` *Set or Unset Environment Variables*

Description

`Sys.setenv` sets environment variables (for other processes called from within R or future calls to `Sys.getenv` from this R process).

`Sys.unsetenv` removes environment variables.

Usage

```
Sys.setenv(...)

Sys.unsetenv(x)
```

Arguments

`...`	named arguments with values coercible to a character string.
`x`	a character vector, or an object coercible to character.

Details

The names `setenv` and `putenv` come from different Unix traditions: R also has `Sys.putenv`, but this is now deprecated. The internal code uses `setenv` if available, otherwise `putenv`.

Non-standard R names must be quoted in `Sys.setenv`: see the examples. Most platforms (and POSIX) do not allow names containing `"="`.

There may be system-specific limits on the maximum length of the values of individual environment variables or of all environment variables.

Value

A logical vector, with elements being true if (un)setting the corresponding variable succeeded. (For `Sys.unsetenv` this includes attempting to remove a non-existent variable.)

Note

Not all systems need support `Sys.setenv` (although all known current platforms do) nor `Sys.unsetenv`. If `Sys.unsetenv` is not supported, it will at least try to set the value of the environment variable to `""`, with a warning.

See Also

`Sys.getenv`, Startup for ways to set environment variables for the R session.

`setwd` for the working directory.

The help for 'environment variables' lists many of the environment variables used by R.

Examples

```
print(Sys.setenv(R_TEST="testit", "A+C"=123))  # `A+C` could also be used
Sys.getenv("R_TEST")
Sys.unsetenv("R_TEST")  # may warn and not succeed
Sys.getenv("R_TEST", unset=NA)
```

`Sys.sleep` *Suspend Execution for a Time Interval*

Description

Suspend execution of R expressions for a given number of seconds

Usage

```
Sys.sleep(time)
```

Arguments

`time`	The time interval to suspend execution for, in seconds.

Details

Using this function allows R to be given very low priority and hence not to interfere with more important foreground tasks. A typical use is to allow a process launched from R to set itself up and read its input files before R execution is resumed.

The intention is that this function suspends execution of R expressions but wakes the process up often enough to respond to GUI events, typically every 0.5 seconds.

There is no guarantee that the process will sleep for the whole of the specified interval, and it may well take slightly longer in real time to resume execution. The resolution of the time interval is system-dependent, but will normally be down to 0.02 secs or better. (On modern Unix-alikes it will be better than 1ms.)

Value

Invisible `NULL`.

Note

This function may not be implemented on all systems.

Examples

```
testit <- function(x)
{
    p1 <- proc.time()
    Sys.sleep(x)
    proc.time() - p1 # The cpu usage should be negligible
}
testit(3.7)
```

---

sys.source *Parse and Evaluate Expressions from a File*

Description

Parses expressions in the given file, and then successively evaluates them in the specified environment.

Usage

```
sys.source(file, envir = baseenv(), chdir = FALSE,
           keep.source = getOption("keep.source.pkgs"))
```

Arguments

file	a character string naming the file to be read from
envir	an R object specifying the environment in which the expressions are to be evaluated. May also be a list or an integer. The default value NULL corresponds to evaluation in the base environment. This is probably not what you want; you should typically supply an explicit envir argument.
chdir	logical; if TRUE, the R working directory is changed to the directory containing file for evaluating.
keep.source	logical. If TRUE, functions keep their source including comments, see options(keep.source = *) for more details.

Details

For large files, keep.source = FALSE may save quite a bit of memory. In order for the code being evaluated to use the correct environment (for example, in global assignments), source code in packages should call topenv(), which will return the name space, if any, the environment set up by sys.source, or the global environment if a saved image is being used.

See Also

source, and library which uses sys.source.

Examples

```
## a simple way to put some objects in an environment
## high on the search path
tmp <- tempfile()
writeLines("aaa <- pi", tmp)
env <- attach(NULL, name = "myenv")
sys.source(tmp, env)
unlink(tmp)
search()
aaa
detach("myenv")
```

Sys.time *Get Current Date and Time*

Description

`Sys.time` and `Sys.Date` returns the system's idea of the current date with and without time.

Usage

```
Sys.time()
Sys.Date()
```

Details

`Sys.time` returns an absolute date-time value which can be converted in various time zones and may return different days.

`Sys.Date` returns the day in the current timezone.

Value

`Sys.time` returns an object of class `"POSIXct"` (see DateTimeClasses). On some systems it will have sub-second accuracy, but on others it will increment in seconds. On systems conforming to POSIX 1003.1-2001 the time will be reported in microsecond increments.

`Sys.Date` returns an object of class `"Date"` (see Date).

See Also

`date` for the system time in a fixed-format character string; the elapsed time component of `proc.time` for possibly finer resolution in changes in time.

`Sys.timezone`.

Examples

```
Sys.time()
## print with possibly greater accuracy:
op <- options(digits.secs=6)
Sys.time()
options(op)

## locale-specific version of date()
```

```
format(Sys.time(), "%a %b %d %X %Y")

Sys.Date()
```

`Sys.which` *Find Full Paths of Executables*

Description

This is an interface to the system command `which`.

Usage

```
Sys.which(names)
```

Arguments

`names`	Character vector of names of possible executables.

Details

The system command `which` reports on the full names of an executable (including an executable script) found on the current path.

Value

A character vector of the same length as `names`, named by `names`. The elements are either the full path to the executable/script or `""` if no executable of that name was found.

Examples

```
## the first two are likely to exist everywhere
## texi2dvi exists on most Unix-alikes and under MiKTeX
Sys.which(c("ftp", "ping", "texi2dvi", "this-does-not-exist"))
```

`system` *Invoke a System Command*

Description

`system` invokes the OS command specified by `command`.

Usage

```
system(command, intern = FALSE, ignore.stderr = FALSE,
       wait = TRUE, input = NULL, show.output.on.console = TRUE,
       minimized = FALSE, invisible = TRUE)
```

Arguments

command	the system command to be invoked, as a string.
intern	a logical (not NA) which indicates whether to make the output of the command an R object. Not available unless popen is supported on the platform.
ignore.stderr	a logical indicating whether error messages written to 'stderr' should be ignored.
wait	a logical indicating whether the R interpreter should wait for the command to finish, or run it asynchronously. This will be ignored (and the interpreter will always wait) if intern = TRUE.
input	if a character vector is supplied, this is copied one string per line to a temporary file, and the standard input of command is redirected to the file.
show.output.on.console, minimized, invisible	arguments that are accepted on other platforms but ignored on this one, with a warning.

Details

command is parsed as a command plus arguments separated by spaces. So if the path to the command (or a filepath argument) contains spaces, it must be quoted e.g. by shQuote.

How the command is run differs by platform: Unix-alikes use a shell ('/bin/sh' by default), and Windows executes the command directly (extensions '.exe', '.com') or as a batch file (extensions '.cmd' and '.bat').

If intern is TRUE then popen is used to invoke the command and the output collected, line by line, into an R character vector. If intern is FALSE then the C function system is used to invoke the command.

The ordering of arguments after the first two has changed from time to time: it is recommended to name all arguments after the first.

Value

If intern = TRUE, a character vector giving the output of the command, one line per character string. (Output lines of more than 8095 characters will be split.) If the command could not be run or gives an error this will be reported on the shell's 'stderr' (unless popen is not supported, when there is an R error).

If intern = FALSE, the return value is an error code (0 for success), given the invisible attribute (so needs to be printed explicitly). If the command could not be run for any reason, the value is 256*127 = 52512. Otherwise if wait = TRUE the value is 256 times the error code returned by the command, and if wait = FALSE it is 0 (the conventional success value).

Stdout and stderr

Error messages written to 'stderr' will be sent by the shell to the terminal unless ignore.stderr = TRUE. They can be captured (in the most likely shells) by

```
system("some command 2>&1", intern=TRUE)
```

What happens to output sent to 'stdout' and 'stderr' if intern = FALSE is interface-specific, and it is unsafe to assume that such messages will appear on the console (they do on the MacOS X console but not on the **gnomeGUI** console, for example).

Note

`wait` is implemented by appending `&` to the command: this is shell-dependent, but required by POSIX and so widely supported.

See Also

`.Platform` for platform-specific variables.

Examples

```
# list all files in the current directory using the -F flag
## Not run: system("ls -F")

# t1 is a character vector, each one
# representing a separate line of output from who
# (if the platform has popen and who)
t1 <- try(system("who", intern = TRUE))

try(system("ls fizzlipuzzli", intern = TRUE, ignore.stderr = TRUE))
# empty since file doesn't exist
```

`system.file` *Find Names of R System Files*

Description

Finds the full file names of files in packages etc.

Usage

```
system.file(..., package = "base", lib.loc = NULL)
```

Arguments

`...`	character strings, specifying subdirectory and file(s) within some package. The default, none, returns the root of the package. Wildcards are not supported.
`package`	a character string with the name of a single package. An error occurs if more than one package name is given.
`lib.loc`	a character vector with path names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.

Value

A character vector of positive length, containing the file names that matched `...`, or the empty string, `""`, if none matched. If matching the root of a package, there is no trailing separator.

As a special case, `system.file()` gives the root of the **base** package only.

See Also

`R.home` for the root directory of the R installation, `list.files`

Examples

```
system.file()                    # The root of the 'base' package
system.file(package = "stats") # The root of package 'stats'
system.file("INDEX")
system.file("help", "AnIndex", package = "splines")
```

---

system.time *CPU Time Used*

Description

Return CPU (and other) times that `expr` used.

Usage

```
system.time(expr, gcFirst = TRUE)
unix.time(expr, gcFirst = TRUE)
```

Arguments

`expr`	Valid R expression to be timed.
`gcFirst`	Logical - should a garbage collection be performed immediately before the timing? Default is `TRUE`.

Details

`system.time` calls the function `proc.time`, evaluates `expr`, and then calls `proc.time` once more, returning the difference between the two `proc.time` calls.

`unix.time` is an alias of `system.time`, for compatibility with S.

Timings of evaluations of the same expression can vary considerably depending on whether the evaluation triggers a garbage collection. When `gcFirst` is `TRUE` a garbage collection (`gc`) will be performed immediately before the evaluation of `expr`. This will usually produce more consistent timings.

Value

A object of class `"proc_time"`: see `proc.time` for details.

Note

It is possible to compile R without support for `system.time`, when the function will throw an error.

See Also

`proc.time`, `time` which is for time series.

Examples

```
require(stats)
system.time(for(i in 1:100) mad(runif(1000)))
## Not run:
exT <- function(n = 1000) {
  # Purpose: Test if system.time works ok;   n: loop size
  system.time(for(i in 1:n) x <- mean(rt(1000, df=4)))
}
#-- Try to interrupt one of the following (using Ctrl-C / Escape):
exT()                  #- about 3 secs on a 1GHz PIII
system.time(exT())     #~ +/- same
## End(Not run)
```

---

t *Matrix Transpose*

Description

Given a matrix or `data.frame` `x`, `t` returns the transpose of `x`.

Usage

```
t(x)
```

Arguments

`x`	a matrix or data frame, typically.

Details

This is a generic function for which methods can be written. The description here applies to the default and `"data.frame"` methods.

A data frame is first coerced to a matrix: see `as.matrix`. When `x` is a vector, it is treated as a column, i.e., the result is a 1-row matrix.

Value

A matrix, with `dim` and `dimnames` constructed appropriately from those of `x`, and other attributes except names copied across.

Note

The *conjugate* transpose of a complex matrix A, denoted A^H or A^*, is computed as `Conj(t(A))`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`aperm` for permuting the dimensions of arrays.

Examples

```
a <- matrix(1:30, 5,6)
ta <- t(a) ##-- i.e.,  a[i, j] == ta[j, i] for all i,j :
for(j in seq(ncol(a)))
  if(! all(a[, j] == ta[j, ])) stop("wrong transpose")
```

---

table *Cross Tabulation and Table Creation*

Description

`table` uses the cross-classifying factors to build a contingency table of the counts at each combination of factor levels.

Usage

```
table(..., exclude = if (useNA == "no") c(NA, NaN), useNA = c("no",
    "ifany", "always"), dnn = list.names(...), deparse.level = 1)

as.table(x, ...)
is.table(x)

## S3 method for class 'table':
as.data.frame(x, row.names = NULL, ...,
              responseName = "Freq")
```

Arguments

`...`	one of more objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted. (For `as.table` and `as.data.frame`, arguments passed to specific methods.)
`exclude`	levels to remove from all factors in `...`. If set to `NULL`, it implies `useNA="always"`.
`useNA`	whether to include extra `NA` levels in the table.
`dnn`	the names to be given to the dimensions in the result (the *dimnames names*).
`deparse.level`	controls how the default `dnn` is constructed. See details.
`x`	an arbitrary R object, or an object inheriting from class `"table"` for the `as.data.frame` method.
`row.names`	a character vector giving the row names for the data frame.
`responseName`	The name to be used for the column of table entries, usually counts.

Details

If the argument `dnn` is not supplied, the internal function `list.names` is called to compute the 'dimname names'. If the arguments in `...` are named, those names are used. For the remaining arguments, `deparse.level = 0` gives an empty name, `deparse.level = 1` uses the supplied argument if it is a symbol, and `deparse.level = 2` will deparse the argument.

Only when `exclude` is specified and non-NULL (i.e., not by default), will `table` potentially drop levels of factor arguments.

Both `exclude` and `useNA` operate on an "all or none" basis. If you want to control the dimensions of a multiway table separately, modify each argument using `factor` or `addNA`

The `summary` method for class `"table"` (used for objects created by `table` or `xtabs`) which gives basic information and performs a chi-squared test for independence of factors (note that the function `chisq.test` currently only handles 2-d tables).

Value

`table()` returns a *contingency table*, an object of `class` `"table"`, an array of integer values. Note that unlike S the result is always an array, a 1D array if one factor is given.

`as.table` and `is.table` coerce to and test for contingency table, respectively.

The `as.data.frame` method for objects inheriting from class `"table"` can be used to convert the array-based representation of a contingency table to a data frame containing the classifying factors and the corresponding entries (the latter as component named by `responseName`). This is the inverse of `xtabs`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`tabulate` is the underlying function and allows finer control.

Use `ftable` for printing (and more) of multidimensional tables. `margin.table`, `prop.table`, `addmargins`.

Examples

```
require(stats) # for rpois and xtabs
## Simple frequency distribution
table(rpois(100,5))
## Check the design:
with(warpbreaks, table(wool, tension))
table(state.division, state.region)

# simple two-way contingency table
with(airquality, table(cut(Temp, quantile(Temp)), Month))

a <- letters[1:3]
table(a, sample(a))                    # dnn is c("a", "")
table(a, sample(a), deparse.level = 0) # dnn is c("", "")
table(a, sample(a), deparse.level = 2) # dnn is c("a", "sample(a)")

## xtabs() <-> as.data.frame.table() :
UCBAdmissions ## already a contingency table
DF <- as.data.frame(UCBAdmissions)
class(tab <- xtabs(Freq ~ ., DF)) # xtabs & table
## tab *is* "the same" as the original table:
all(tab == UCBAdmissions)
all.equal(dimnames(tab), dimnames(UCBAdmissions))
```

```
a <- rep(c(NA, 1/0:3), 10)
table(a)
table(a, exclude=NULL)
b <- factor(rep(c("A","B","C"), 10))
table(b)
table(b, exclude="B")
d <- factor(rep(c("A","B","C"), 10), levels=c("A","B","C","D","E"))
table(d, exclude="B")
print(table(b,d), zero.print = ".")

## NA counting:
is.na(d) <- 3:4
d. <- addNA(d)
d.[1:7]
table(d.) # ", exclude = NULL" is not needed
## i.e., if you want to count the NA's of 'd', use
table(d, useNA="ifany")

## Two-way tables with NA counts. The 3rd variant is absurd, but shows
## something that cannot be done using exclude or useNA.
with(airquality,
   table(OzHi=Ozone > 80, Month, useNA="ifany"))
with(airquality,
   table(OzHi=Ozone > 80, Month, useNA="always"))
with(airquality,
   table(OzHi=Ozone > 80, addNA(Month)))
```

`tabulate` *Tabulation for Vectors*

Description

`tabulate` takes the integer-valued vector `bin` and counts the number of times each integer occurs in it.

Usage

```
tabulate(bin, nbins = max(1, bin))
```

Arguments

`bin`	a numeric vector (of positive integers), or a factor.
`nbins`	the number of bins to be used.

Details

`tabulate` is used as the basis of the `table` function.

If `bin` is a factor, its internal integer representation is tabulated.

If the elements of `bin` are numeric but not integers, they are truncated to the nearest integer.

Value

An integer vector (without names). There is a bin for each of the values `1, ..., nbins`; values outside that range are (silently) ignored.

See Also

table, factor.

Examples

```
tabulate(c(2,3,5))
tabulate(c(2,3,3,5), nbins = 10)
tabulate(c(-2,0,2,3,3,5))  # -2 and 0 are ignored
tabulate(c(-2,0,2,3,3,5), nbins = 3)
tabulate(factor(letters[1:10]))
```

---

tapply *Apply a Function Over a "Ragged" Array*

Description

Apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.

Usage

```
tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

Arguments

X	an atomic object, typically a vector.
INDEX	list of factors, each of same length as X. The elements are coerced to factors by as.factor.
FUN	the function to be applied. In the case of functions like +, %*%, etc., the function name must be quoted. If FUN is NULL, tapply returns a vector which can be used to subscript the multi-way array tapply normally produces.
...	optional arguments to FUN: the Note section.
simplify	If FALSE, tapply always returns an array of mode "list". If TRUE (the default), then if FUN always returns a scalar, tapply returns an array with the mode of the scalar.

Value

When FUN is present, tapply calls FUN for each cell that has any data in it. If FUN returns a single atomic value for each such cell (e.g., functions mean or var) and when simplify is TRUE, tapply returns a multi-way array containing the values, and NA for the empty cells. The array has the same number of dimensions as INDEX has components; the number of levels in a dimension is the number of levels (nlevels()) in the corresponding component of INDEX. Note that if the return value has a class (e.g. an object of class "Date") the class is discarded.

Note that contrary to S, simplify = TRUE always returns an array, possibly 1-dimensional.

If FUN does not return a single atomic value, tapply returns an array of mode list whose components are the values of the individual calls to FUN, i.e., the result is a list with a dim attribute.

When there is an array answer, its dimnames are named by the names of INDEX and are based on the levels of the grouping factors (possibly after coercion).

For a list result, the elements corresponding to empty cells are NULL.

Note

Optional arguments to FUN supplied by the ... argument are not divided into cells. It is therefore inappropriate for FUN to expect additional arguments with the same length as X.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

the convenience functions by and aggregate (using tapply); apply, lapply with its versions sapply and mapply.

Examples

```
require(stats)
groups <- as.factor(rbinom(32, n = 5, prob = 0.4))
tapply(groups, groups, length) #- is almost the same as
table(groups)

## contingency table from data.frame : array with named dimnames
tapply(warpbreaks$breaks, warpbreaks[,-1], sum)
tapply(warpbreaks$breaks, warpbreaks[, 3, drop = FALSE], sum)

n <- 17; fac <- factor(rep(1:3, length = n), levels = 1:5)
table(fac)
tapply(1:n, fac, sum)
tapply(1:n, fac, sum, simplify = FALSE)
tapply(1:n, fac, range)
tapply(1:n, fac, quantile)

## example of ... argument: find quarterly means
tapply(presidents, cycle(presidents), mean, na.rm = TRUE)

ind <- list(c(1, 2, 2), c("A", "A", "B"))
table(ind)
tapply(1:3, ind) #-> the split vector
tapply(1:3, ind, sum)
```

---

taskCallback *Add or remove a top-level task callback*

Description

addTaskCallback registers an R function that is to be called each time a top-level task is completed.

removeTaskCallback un-registers a function that was registered earlier via addTaskCallback.

These provide low-level access to the internal/native mechanism for managing task-completion actions. One can use taskCallbackManager at the S-language level to manage S functions that are called at the completion of each task. This is easier and more direct.

Usage

```
addTaskCallback(f, data = NULL, name = character(0))
removeTaskCallback(id)
```

Arguments

f	the function that is to be invoked each time a top-level task is successfully completed. This is called with 5 or 4 arguments depending on whether data is specified or not, respectively. The return value should be a logical value indicating whether to keep the callback in the list of active callbacks or discard it.
data	if specified, this is the 5-th argument in the call to the callback function f.
id	a string or an integer identifying the element in the internal callback list to be removed. Integer indices are 1-based, i.e the first element is 1. The names of currently registered handlers is available using getTaskCallbackNames and is also returned in a call to addTaskCallback.
name	character: names to be used.

Details

Top-level tasks are individual expressions rather than entire lines of input. Thus an input line of the form expression1 ; expression2 will give rise to 2 top-level tasks.

A top-level task callback is called with the expression for the top-level task, the result of the top-level task, a logical value indicating whether it was successfully completed or not (always TRUE at present), and a logical value indicating whether the result was printed or not. If the data argument was specified in the call to addTaskCallback, that value is given as the fifth argument.

The callback function should return a logical value. If the value is FALSE, the callback is removed from the task list and will not be called again by this mechanism. If the function returns TRUE, it is kept in the list and will be called on the completion of the next top-level task.

Value

addTaskCallback returns an integer value giving the position in the list of task callbacks that this new callback occupies. This is only the current position of the callback. It can be used to remove the entry as long as no other values are removed from earlier positions in the list first.

removeTaskCallback returns a logical value indicating whether the specified element was removed. This can fail (i.e., return FALSE) if an incorrect name or index is given that does not correspond to the name or position of an element in the list.

Note

This is an experimental feature and the interface may be changed in the future.

There is also C-level access to top-level task callbacks to allow C routines rather than R functions be used.

See Also

getTaskCallbackNames taskCallbackManager http://developer.r-project.org/TaskHandlers.pdf

Examples

```
times <- function(total = 3, str="Task a") {
  ctr <- 0

  function(expr, value, ok, visible) {
   ctr <<- ctr + 1
   cat(str, ctr, "\n")
   if(ctr == total) {
     cat("handler removing itself\n")
   }
   return(ctr < total)
  }
}

# add the callback that will work for
# 4 top-level tasks and then remove itself.
n <- addTaskCallback(times(4))

# now remove it, assuming it is still first in the list.
removeTaskCallback(n)

## Not run:
# There is no point in running this
# as
 addTaskCallback(times(4))

 sum(1:10)
 sum(1:10)
 sum(1:10)
 sum(1:10)
 sum(1:10)
## End(Not run)
```

`taskCallbackManager`

Create an R-level task callback manager

Description

This provides an entirely S-language mechanism for managing callbacks or actions that are invoked at the conclusion of each top-level task. Essentially, we register a single R function from this manager with the underlying, native task-callback mechanism and this function handles invoking the other R callbacks under the control of the manager. The manager consists of a collection of functions that access shared variables to manage the list of user-level callbacks.

Usage

```
taskCallbackManager(handlers = list(), registered = FALSE,
                    verbose = FALSE)
```

Arguments

handlers	this can be a list of callbacks in which each element is a list with an element named "f" which is a callback function, and an optional element named "data" which is the 5-th argument to be supplied to the callback when it is invoked. Typically this argument is not specified, and one uses add to register callbacks after the manager is created.
registered	a logical value indicating whether the evaluate function has already been registered with the internal task callback mechanism. This is usually FALSE and the first time a callback is added via the add function, the evaluate function is automatically registered. One can control when the function is registered by specifying TRUE for this argument and calling addTaskCallback manually.
verbose	a logical value, which if TRUE, causes information to be printed to the console about certain activities this dispatch manager performs. This is useful for debugging callbacks and the handler itself.

Value

A list containing 6 functions:

add	register a callback with this manager, giving the function, an optional 5-th argument, an optional name by which the callback is stored in the list, and a register argument which controls whether the evaluate function is registered with the internal C-level dispatch mechanism if necessary.
remove	remove an element from the manager's collection of callbacks, either by name or position/index.
evaluate	the 'real' callback function that is registered with the C-level dispatch mechanism and which invokes each of the R-level callbacks within this manager's control.
suspend	a function to set the suspend state of the manager. If it is suspended, none of the callbacks will be invoked when a task is completed. One sets the state by specifying a logical value for the status argument.
register	a function to register the evaluate function with the internal C-level dispatch mechanism. This is done automatically by the add function, but can be called manually.
callbacks	returns the list of callbacks being maintained by this manager.

Note

This is an experimental feature and the interface may be changed in the future.

See Also

addTaskCallback, removeTaskCallback, getTaskCallbackNames\ http://developer.r-project.org/TaskHandlers.pdf

Examples

```
# create the manager
h <- taskCallbackManager()

# add a callback
```

```
h$add(function(expr, value, ok, visible) {
                        cat("In handler\n")
                        return(TRUE)
                      }, name = "simpleHandler")

# look at the internal callbacks.
getTaskCallbackNames()

# look at the R-level callbacks
names(h$callbacks())

getTaskCallbackNames()
removeTaskCallback("R-taskCallbackManager")
```

`taskCallbackNames` *Query the names of the current internal top-level task callbacks*

Description

This provides a way to get the names (or identifiers) for the currently registered task callbacks that are invoked at the conclusion of each top-level task. These identifiers can be used to remove a callback.

Usage

```
getTaskCallbackNames()
```

Value

A character vector giving the name for each of the registered callbacks which are invoked when a top-level task is completed successfully. Each name is the one used when registering the callbacks and returned as the in the call to `addTaskCallback`.

Note

One can use `taskCallbackManager` to manage user-level task callbacks, i.e., S-language functions, entirely within the S language and access the names more directly.

See Also

`addTaskCallback`, `removeTaskCallback`, `taskCallbackManager`\ `http://developer.r-project.org/TaskHandlers.pdf`

Examples

```
n <- addTaskCallback(function(expr, value, ok, visible) {
                        cat("In handler\n")
                        return(TRUE)
                      }, name = "simpleHandler")

getTaskCallbackNames()

  # now remove it by name
removeTaskCallback("simpleHandler")
```

```
h <- taskCallbackManager()
h$add(function(expr, value, ok, visible) {
                       cat("In handler\n")
                       return(TRUE)
                     }, name = "simpleHandler")
getTaskCallbackNames()
removeTaskCallback("R-taskCallbackManager")
```

`tempfile` *Create Names for Temporary Files*

Description

`tempfile` returns a vector of character strings which can be used as names for temporary files.

Usage

```
tempfile(pattern = "file", tmpdir = tempdir())
tempdir()
```

Arguments

`pattern`	a non-empty character vector giving the initial part of the name.
`tmpdir`	a non-empty character vector giving the directory name

Details

If `pattern` has length greater than one then the result is of the same length giving a temporary file name for each component of `pattern`.

The names are very likely to be unique among calls to `tempfile` in an R session and across simultaneous R sessions. The filenames are guaranteed not to be currently in use.

The file name is made of the pattern and a random suffix in hex. By default, the filenames will be in the directory given by `tempdir()`. This will be a subdirectory of the temporary directory found by the following rule. The environment variables `TMPDIR`, `TMP` and `TEMP` are checked in turn and the first found which points to a writable directory is used: if none succeeds '/tmp' is used.

Value

For `tempfile` a character vector giving the names of possible (temporary) files. Note that no files are generated by `tempfile`.

For `tempdir`, the path of the per-session temporary directory.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`unlink` for deleting files.

Examples

```
tempfile(c("ab", "a b c"))   # give file name with spaces in!

tempdir() # working on all platforms with quite platform dependent result
```

`textConnection` *Text Connections*

Description

Input and output text connections.

Usage

```
textConnection(object, open = "r", local = FALSE)

textConnectionValue(con)
```

Arguments

`object`	character. A description of the connection. For an input this is an R character vector object, and for an output connection the name for the R character vector to receive the output, or `NULL` (for none).
`open`	character. Either `"r"` (or equivalently `""`) for an input connection or `"w"` or `"a"` for an output connection.
`local`	logical. Used only for output connections. If `TRUE`, output is assigned to a variable in the calling environment. Otherwise the global environment is used.
`con`	An output text connection.

Details

An input text connection is opened and the character vector is copied at time the connection object is created, and `close` destroys the copy.

An output text connection is opened and creates an R character vector of the given name in the user's workspace or in the calling environment, depending on the value of the `local` argument. This object will at all times hold the completed lines of output to the connection, and `isIncomplete` will indicate if there is an incomplete final line. Closing the connection will output the final line, complete or not. (A line is complete once it has been terminated by end-of-line, represented by `"\n"` in R.) The output character vector has locked bindings (see `lockBinding`) until `close` is called on the connection. The character vector can also be retrieved *via* `textConnectionValue`, which is the only way to do so if `object = NULL`. If the current locale is detected as Latin-1 or UTF-8, non-ASCII elements of the character vector will be marked accordingly (see `Encoding`).

Opening a text connection with `mode = "a"` will attempt to append to an existing character vector with the given name in the user's workspace or the calling environment. If none is found (even if an object exists of the right name but the wrong type) a new character vector will be created, with a warning.

You cannot `seek` on a text connection, and `seek` will always return zero as the position.

Value

For `textConnection`, a connection object of class `"textConnection"` which inherits from class `"connection"`.

For `textConnectionValue`, a character vector.

Note

As output text connections keep the character vector up to date line-by-line, they are relatively expensive to use, and it is often better to use an anonymous `file()` connection to collect output.

On (rare) platforms where `vsnprintf` does not return the needed length of output there is a 100,000 character limit on the length of line for output connections: longer lines will be truncated with a warning.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language.* Springer.
[S has input text connections only.]

See Also

`connections`, `showConnections`, `pushBack`, `capture.output`.

Examples

```
zz <- textConnection(LETTERS)
readLines(zz, 2)
scan(zz, "", 4)
pushBack(c("aa", "bb"), zz)
scan(zz, "", 4)
close(zz)

zz <- textConnection("foo", "w")
writeLines(c("testit1", "testit2"), zz)
cat("testit3 ", file=zz)
isIncomplete(zz)
cat("testit4\n", file=zz)
isIncomplete(zz)
close(zz)
foo

## Not run:
# capture R output: use part of example from help(lm)
zz <- textConnection("foo", "w")
ctl <- c(4.17, 5.58, 5.18, 6.11, 4.5, 4.61, 5.17, 4.53, 5.33, 5.14)
trt <- c(4.81, 4.17, 4.41, 3.59, 5.87, 3.83, 6.03, 4.89, 4.32, 4.69)
group <- gl(2, 10, 20, labels = c("Ctl", "Trt"))
weight <- c(ctl, trt)
sink(zz)
anova(lm.D9 <- lm(weight ~ group))
cat("\nSummary of Residuals:\n\n")
summary(resid(lm.D9))
sink()
close(zz)
cat(foo, sep = "\n")
## End(Not run)
```

tilde *Tilde Operator*

Description

Tilde is used to separate the left- and right-hand sides in model formula.

Usage

```
y ~ model
```

Arguments

`y, model`	symbolic expressions.

Details

The left-hand side is optional, and one-sided formulae are used in some contexts.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`formula`

timezones *Time Zones*

Description

Information about time zones in R. `Sys.timezone` returns the current time zone.

Usage

```
Sys.timezone()
```

Details

Time zones are a system-specific topic, but as from R 2.7.0 almost all R platforms use the same underlying code, used by Linux, Mac OS X, Solaris, AIX, FreeBSD, Sun Java >= 1.4 and Tcl >= 8.5, and supplied with R on Windows.

It is not in general possible to retrieve the system's own name(s) for the current timezone, but `Sys.timezone` will retrieve the name it uses for the current time (and the name may differ depending on whether daylight saving time is in effect).

On most platforms it is possible to set the time zone via the environment variable `TZ`: see the section on 'Time zone names' for suitable values.

Note that the principal difficulty with time zones is their individual history: over the last 100 years places have changed their affiliation between major time zones, have opted out of (or in to) DST in various years or adopted rule changes late or not at all. This often involves tiny administrative units in the US/Canada: Iowa had 23 different implementations of DST in the 1960's!

Time zones did not come into use until the second half of the nineteenth century, and DST was first introduced in the early twentieth century, most widely during the First World War (in 1916). The most common implementation of `POSIXct` is a signed 32-bit integers and so only goes back to the end of 1901: on such systems R assumes that dates prior to that are in the same time zone as they were in 1902.

Value

`Sys.timezone` returns an OS-specific character string, possibly an empty string. Typically this is an abbreviation such as `"EST"`.

Time zone names

Where OSes describe their valid time zones can be obscure. The help for the C function `tzset` can be helpful, but it can also be inaccurate. There is a cumbersome POSIX specification (listed under environment variable `TZ` at http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap08.html), which is often at least partially supported, but there usually are other more user-friendly ways to specify timezones.

Many systems make use of a timezone database compiled by Arthur Olson, in which the preferred way to refer to a time zone by a location (typically of a city) e.g. `Europe/London`, `America/Los_Angeles`, `Pacific/Easter`. Some traditional designations are also allowed such as `EST5EDT` or `GB`. (Beware that some of these designations may not be what you think: in particular `EST` is a time zone used in Canada *without* daylight savings time, and not `EST5EDT` nor (Australian) Eastern Standard Time.) The designation can also be an optional colon prepended to the path to a file giving complied zone information (and the examples above are all files in a system-specific location). See http://www.twinsun.com/tz/tz-link.htm for more details and references.

For most Unix-alikes use the Olson databases. The system-specific default location in the file system varies, e.g. '/usr/share/zoneinfo', '/usr/etc/zoneinfo', '/usr/lib/zoneinfo' or '/usr/share/lib/zoneinfo'.

Note

There is currently (2007/8) considerable disruption over changes to the timings of the DST transitions, aimed at energy conservation. These often have short notice and time zone databases may not be up to date (even if the OS has been updated recently).

Note that except on Windows, the operation of time zones is an OS service, and even on Windows a third-party database is used and can be updated (see the section on 'Time zone names'). Incorrect results will never be an R issue, so please ensure that you have the courtesy not to blame R for them.

See Also

`Sys.time`, `as.POSIXlt`.

http://en.wikipedia.org/wiki/Time_zone and http://www.twinsun.com/tz/tz-link.htm for extensive sets of links.

Examples

```
Sys.timezone()
```

`toString` *Convert an R Object to a Character String*

Description

This is a helper function for `format` to produce a single character string describing an R object.

Usage

```
toString(x, ...)

## Default S3 method:
toString(x, width = NULL, ...)
```

Arguments

`x`	The object to be converted.
`width`	Suggestion for the maximum field width. Values of `NULL` or `0` indicate no maximum. The minimum value accepted is 6 and smaller values are taken as 6.
`...`	Optional arguments passed to or from methods.

Details

This is a generic function for which methods can be written: only the default method is described here. Most methods should honor the `width` argument to specify the maximum display width (as measured by `nchar(type = "width")`) of the result.

The default method first converts `x` to character and then concatenates the elements separated by `", "`. If `width` is supplied and is not `NULL`, the default method returns the first `width - 4` characters of the result with `....` appended, if the full result would use more than `width` characters.

Value

A character vector of length 1 is returned.

Author(s)

Robert Gentleman

See Also

`format`

Examples

```
x <- c("a", "b", "aaaaaaaaaaa")
toString(x)
toString(x, width=8)
```

`trace` *Interactive Tracing and Debugging of Calls to a Function or Method*

Description

A call to `trace` allows you to insert debugging code (e.g., a call to `browser` or `recover`) at chosen places in any function. A call to `untrace` cancels the tracing. Specified methods can be traced the same way, without tracing all calls to the function. Trace code can be any R expression. Tracing can be temporarily turned on or off globally by calling `tracingState`.

Usage

```
trace(what, tracer, exit, at, print, signature,
      where = topenv(parent.frame()), edit = FALSE)
untrace(what, signature = NULL, where = topenv(parent.frame()))

tracingState(on = NULL)
.doTrace(expr, msg)
```

Arguments

`what`	The name (quoted or not) of a function to be traced or untraced. For `untrace` or for `trace` with more than one argument, more than one name can be given in the quoted form, and the same action will be applied to each one.
`tracer`	Either a function or an unevaluated expression. The function will be called or the expression will be evaluated either at the beginning of the call, or before those steps in the call specified by the argument `at`. See the details section.
`exit`	Either a function or an unevaluated expression. The function will be called or the expression will be evaluated on exiting the function. See the details section.
`at`	optional numeric vector. If supplied, `tracer` will be called just before the corresponding step in the body of the function. See the details section.
`print`	If `TRUE` (as per default), a descriptive line is printed before any trace expression is evaluated.
`signature`	If this argument is supplied, it should be a signature for a method for function `what`. In this case, the method, and *not* the function itself, is traced.
`edit`	For complicated tracing, such as tracing within a loop inside the function, you will need to insert the desired calls by editing the body of the function. If so, supply the `edit` argument either as `TRUE`, or as the name of the editor you want to use. Then `trace()` will call `edit` and use the version of the function after you edit it. See the details section for additional information.
`where`	where to look for the function to be traced; by default, the top-level environment of the call to `trace`. An important use of this argument is to trace a function when it is called from a package with a name space. The current name space mechanism imports the functions to be called (with the exception of functions in the base package). The functions being called are *not* the same objects seen from the top-level (in general, the imported packages may not even be attached). Therefore, you must ensure that the correct versions are being traced. The way to do this is to set argument `where` to a function in the name space. The tracing computations

will then start looking in the environment of that function (which will be the name space of the corresponding package). (Yes, it's subtle, but the semantics here are central to how name spaces work in R.)

`on` logical; a call to the support function `tracingState` returns `TRUE` if tracing is globally turned on, `FALSE` otherwise. An argument of one or the other of those values sets the state. If the tracing state is `FALSE`, none of the trace actions will actually occur (used, for example, by debugging functions to shut off tracing during debugging).

`expr, msg` arguments to the support function `.doTrace`, calls to which are inserted into the modified function or method: `expr` is the tracing action (such as a call to `browser()`, and `msg` is a string identifying the place where the trace action occurs.

Details

The `trace` function operates by constructing a revised version of the function (or of the method, if `signature` is supplied), and assigning the new object back where the original was found. If only the `what` argument is given, a line of trace printing is produced for each call to the function (back compatible with the earlier version of `trace`).

The object constructed by `trace` is from a class that extends `"function"` and which contains the original, untraced version. A call to `untrace` re-assigns this version.

If the argument `tracer` or `exit` is the name of a function, the tracing expression will be a call to that function, with no arguments. This is the easiest and most common case, with the functions `browser` and `recover` the likeliest candidates; the former browses in the frame of the function being traced, and the latter allows browsing in any of the currently active calls.

The `tracer` or `exit` argument can also be an unevaluated expression (such as returned by a call to `quote` or `substitute`). This expression itself is inserted in the traced function, so it will typically involve arguments or local objects in the traced function. An expression of this form is useful if you only want to interact when certain conditions apply (and in this case you probably want to supply `print=FALSE` in the call to `trace` also).

When the `at` argument is supplied, it should be a vector of integers referring to the substeps of the body of the function (this only works if the body of the function is enclosed in `{ ...}`. In this case `tracer` is *not* called on entry, but instead just before evaluating each of the steps listed in `at`. (Hint: you don't want to try to count the steps in the printed version of a function; instead, look at `as.list(body(f))` to get the numbers associated with the steps in function `f`.)

An intrinsic limitation in the `exit` argument is that it won't work if the function itself uses `on.exit`, since the existing calls will override the one supplied by `trace`.

Tracing does not nest. Any call to `trace` replaces previously traced versions of that function or method (except for edited versions as discussed below), and `untrace` always restores an untraced version. (Allowing nested tracing has too many potentials for confusion and for accidentally leaving traced versions behind.)

When the `edit` argument is used repeatedly with no call to `untrace` on the same function or method in between, the previously edited version is retained. If you want to throw away all the previous tracing and then edit, call `untrace` before the next call to `trace`. Editing may be combined with automatic tracing; just supply the other arguments such as `tracer`, and the `edit` argument as well. The `edit=TRUE` argument uses the default editor (see `edit`).

Tracing primitive functions (builtins and specials) from the base package works, but only by a special mechanism and not very informatively. Tracing a primitive causes the primitive to be replaced by a function with argument `...` (only). You can get a bit of information out, but not much. A warning message is issued when `trace` is used on a primitive.

The practice of saving the traced version of the function back where the function came from means that tracing carries over from one session to another, *if* the traced function is saved in the session image. (In the next session, `untrace` will remove the tracing.) On the other hand, functions that were in a package, not in the global environment, are not saved in the image, so tracing expires with the session for such functions.

Tracing a method is basically just like tracing a function, with the exception that the traced version is stored by a call to `setMethod` rather than by direct assignment, and so is the untraced version after a call to `untrace`.

The version of `trace` described here is largely compatible with the version in S-Plus, although the two work by entirely different mechanisms. The S-Plus `trace` uses the session frame, with the result that tracing never carries over from one session to another (R does not have a session frame). Another relevant distinction has nothing directly to do with `trace`: The browser in S-Plus allows changes to be made to the frame being browsed, and the changes will persist after exiting the browser. The R browser allows changes, but they disappear when the browser exits. This may be relevant in that the S-Plus version allows you to experiment with code changes interactively, but the R version does not. (A future revision may include a 'destructive' browser for R.)

Value

In the simple version (just the first argument), invisible `NULL`. Otherwise, the traced function(s) name(s). The relevant consequence is the assignment that takes place.

Note

The version of function tracing that includes any of the arguments except for the function name requires the **methods** package (because it uses special classes of objects to store and restore versions of the traced functions).

If methods dispatch is not currently on, `trace` will load the methods name space, but will not put the methods package on the search list.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`browser` and `recover`, the likeliest tracing functions; also, `quote` and `substitute` for constructing general expressions.

Examples

```
require(graphics)

##  Very simple use
trace(sum)
hist(stats::rnorm(100)) # shows about 3-4 calls to sum()
untrace(sum)

if(.isMethodsDispatchOn()) { # non-simple use needs 'methods' package

f <- function(x, y) {
    y <- pmax(y, .001)
    x ^ y
```

```
}

## arrange to call the browser on entering and exiting
## function f
trace("f", browser, exit = browser)

## instead, conditionally assign some data, and then browse
## on exit, but only then.  Don't bother me otherwise

trace("f", quote(if(any(y < 0)) yOrig <- y),
      exit = quote(if(exists("yOrig")) browser()),
      print = FALSE)

## trace a utility function, with recover so we
## can browse in the calling functions as well.

trace("as.matrix", recover)

## turn off the tracing

untrace(c("f", "as.matrix"))

## Not run:
## trace calls to the function lm() that come from
## the nlme package.
## (The function nlme is in that package, and the package
## has a name space, so the where= argument must be used
## to get the right version of lm)

trace(lm, exit = recover, where = nlme)
## End(Not run)
}
```

`traceback` *Print Call Stacks*

Description

By default `traceback()` prints the call stack of the last uncaught error, i.e., the sequence of calls that lead to the error. This is useful when an error occurs with an unidentifiable error message. It can also be used to print arbitrary lists of deparsed calls.

Usage

```
traceback(x = NULL, max.lines = getOption("deparse.max.lines"))
```

Arguments

`x`	`NULL` (default, meaning `.Traceback`), or a list or pairlist of deparsed calls.
`max.lines`	The maximum number of lines to be printed *per call*. The default is unlimited.

Details

The stack of the last uncaught error is stored as a list of deparsed calls in `.Traceback`, which `traceback` prints in a user-friendly format. The stack of deparsed calls always contains all function calls and all foreign function calls (such as `.Call`): if profiling is in progress it will include calls to some primitive functions. (Calls to builtins are included, but not to specials.)

Errors which are caught *via* `try` or `tryCatch` do not generate a traceback, so what is printed is the call sequence for the last uncaught error, and not necessarily for the last error.

Value

`traceback()` returns nothing, but prints the deparsed call stack deepest call first. The calls may print on more than one line, and the first line for each call is labelled by the frame number. The number of lines printed per call can be limited via `max.lines`.

Warning

It is undocumented where `.Traceback` is stored nor that it is visible, and this is subject to change. Prior to R 2.4.0 it was stored in the workspace, but no longer.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
foo <- function(x) { print(1); bar(2) }
bar <- function(x) { x + a.variable.which.does.not.exist }
## Not run:
foo(2) # gives a strange error
traceback()
## End(Not run)
## 2: bar(2)
## 1: foo(2)
bar
## Ah, this is the culprit ...
```

`tracemem` *Trace Copying of Objects*

Description

This function marks an object so that a message is printed whenever the internal function `duplicate` is called. This happens when two objects share the same memory and one of them is modified. It is a major cause of hard-to-predict memory use in R.

Usage

```
tracemem(x)
untracemem(x)
retracemem(x, previous = NULL)
```

Arguments

x	An R object, not a function or environment or NULL.
previous	A value as returned by tracemem or retracemem.

Details

This functionality is optional, determined at compilation, because it makes R run a little more slowly even when no objects are being traced. tracemem and untracemem give errors when R is not compiled with memory profiling; retracemem does not (so it can be left in code during development).

When an object is traced any copying of the object by the C function duplicate or by arithmetic or mathematical operations produces a message to standard output. The message consists of the string tracemem, the identifying strings for the object being copied and the new object being created, and a stack trace showing where the duplication occurred. retracemem() is used to indicate that a variable should be considered a copy of a previous variable (e.g. after subscripting).

The messages can be turned off with tracingState.

It is not possible to trace functions, as this would conflict with trace and it is not useful to trace NULL, environments, promises, weak references, or external pointer objects, as these are not duplicated.

Value

A character string for identifying the object in the trace output (an address in hex enclosed in angle brackets), or NULL (invisibly for untracemem.

See Also

trace, Rprofmem

http://developer.r-project.org/memory-profiling.html

Examples

```
## Not run:
a <- 1:10
tracemem(a)
## b and a share memory
b <- a
b[1] <- 1
untracemem(a)

## copying in lm
d <- stats::rnorm(10)
tracemem(d)
lm(d ~ a+log(b))

## f is not a copy and is not traced
f <- d[-1]
f+1
## indicate that f should be traced as a copy of d
retracemem(f, retracemem(d))
f+1
## End(Not run)
```

`transform` *Transform an Object, for Example a Data Frame*

Description

`transform` is a generic function, which—at least currently—only does anything useful with data frames. `transform.default` converts its first argument to a data frame if possible and calls `transform.data.frame`.

Usage

```
transform(`_data`, ...)
```

Arguments

`_data`	The object to be transformed
`...`	Further arguments of the form `tag=value`

Details

The `...` arguments to `transform.data.frame` are tagged vector expressions, which are evaluated in the data frame `_data`. The tags are matched against `names(_data)`, and for those that match, the value replace the corresponding variable in `_data`, and the others are appended to `_data`.

Value

The modified value of `_data`.

Note

Prior to R 2.3.0, the first argument was named `x`, but this caused trouble if people wanted to create a variable of that name. Names starting with an underscore are syntactically invalid, so the current choice should be less problematic.

If some of the values are not vectors of the appropriate length, you deserve whatever you get!

Author(s)

Peter Dalgaard

See Also

`subset`, `list`, `data.frame`

Examples

```
transform(airquality, Ozone = -Ozone)
transform(airquality, new = -Ozone, Temp = (Temp-32)/1.8)

attach(airquality)
transform(Ozone, logOzone = log(Ozone)) # marginally interesting ...
detach(airquality)
```

`Trig` *Trigonometric Functions*

Description

These functions give the obvious trigonometric functions. They respectively compute the cosine, sine, tangent, arc-cosine, arc-sine, arc-tangent, and the two-argument arc-tangent.

Usage

```
cos(x)
sin(x)
tan(x)
acos(x)
asin(x)
atan(x)
atan2(y, x)
```

Arguments

`x, y`	numeric or complex vectors.

Details

The arc-tangent of two arguments `atan2(y, x)` returns the angle between the x-axis and the vector from the origin to (x, y), i.e., for positive arguments `atan2(y, x) == atan(y/x)`.

Angles are in radians, not degrees (i.e., a right angle is $\pi/2$).

All except `atan2` are generic functions: methods can be defined for them individually or via the `Math` group generic.

Complex values

For the inverse trigonometric functions, branch cuts are defined as in Abramowitz and Stegun, figure 4.4, page 79. Continuity on the branch cuts is standard.

For `asin()` and `acos()`, there are two cuts, both along the real axis: $(-\infty, -1]$ and $[1, \infty)$. Functions `asin()` and `acos()` are continuous from above on the interval $(-\infty, -1]$ and continuous from below on $[1, \infty)$.

For `atan()` there are two cuts, both along the pure imaginary axis: $(-\infty i, -1i]$ and $[1i, \infty i)$. It is continuous from the left on the interval $(-\infty i, -1i]$ and from the right on the interval $[1i, \infty i)$.

S4 methods

All except `atan2` are S4 generic functions: methods can be defined for them individually or via the `Math` group generic.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Abramowitz, M. and Stegun, I. A. (1972). *Handbook of Mathematical Functions,* New York: Dover. Chapter 4. Elementary Transcendental Functions: Logarithmic, Exponential, Circular and Hyperbolic Functions

---

`try` *Try an Expression Allowing Error Recovery*

Description

`try` is a wrapper to run an expression that might fail and allow the user's code to handle error-recovery.

Usage

```
try(expr, silent = FALSE)
```

Arguments

`expr`	an R expression to try.
`silent`	logical: should the report of error messages be suppressed?

Details

`try` evaluates an expression and traps any errors that occur during the evaluation. If an error occurs then the error message is printed to the `stderr` connection unless `options("show.error.messages")` is false or the call includes `silent = TRUE`. The error message is also stored in a buffer where it can be retrieved by `geterrmessage`. (This should not be needed as the value returned in case of an error contains the error message.) `try` is implemented using `tryCatch`.

Value

The value of the expression if `expr` is evaluated without error, but an invisible object of class `"try-error"` containing the error message if it fails.

See Also

`options` for setting error handlers and suppressing the printing of error messages; `geterrmessage` for retrieving the last error message. `tryCatch` provides another means of catching and handling errors.

Examples

```
## this example will not work correctly in example(try), but
## it does work correctly if pasted in
options(show.error.messages = FALSE)
try(log("a"))
print(.Last.value)
options(show.error.messages = TRUE)

## alternatively,
print(try(log("a"), TRUE))

## run a simulation, keep only the results that worked.
set.seed(123)
x <- stats::rnorm(50)
doit <- function(x)
{
    x <- sample(x, replace=TRUE)
    if(length(unique(x)) > 30) mean(x)
    else stop("too few unique points")
}
## alternative 1
res <- lapply(1:100, function(i) try(doit(x), TRUE))
## alternative 2
## Not run:
res <- vector("list", 100)
for(i in 1:100) res[[i]] <- try(doit(x), TRUE)
## End(Not run)
unlist(res[sapply(res, function(x) !inherits(x, "try-error"))])
```

---

typeof *The Type of an Object*

Description

`typeof` determines the (R internal) type or storage mode of any object

Usage

```
typeof(x)
```

Arguments

`x`	any R object.

Value

A character string. The possible values are listed in the structure `TypeTable` in 'src/main/util.c'. Current values are the vector types `"logical"`, `"integer"`, `"double"`, `"complex"`, `"character"`, `"raw"` and `"list"`, `"NULL"`, `"closure"` (function), `"special"` and `"builtin"` (basic functions and operators), `"environment"`, `"S4"` (some S4 objects) and others that are unlikely to be seen at user level (`"symbol"`, `"pairlist"`, `"promise"`, `"language"`, `"char"`, `"..."`, `"any"`, `"expression"`, `"externalptr"`, `"bytecode"` and `"weakref"`).

See Also

mode, storage.mode.

isS4 to determine if an object has an S4 class.

Examples

```
typeof(2)
mode(2)
```

---

unique *Extract Unique Elements*

Description

unique returns a vector, data frame or array like x but with duplicate elements/rows removed.

Usage

```
unique(x, incomparables = FALSE, ...)

## Default S3 method:
unique(x, incomparables = FALSE, fromLast = FALSE, ...)

## S3 method for class 'matrix':
unique(x, incomparables = FALSE, MARGIN = 1,
       fromLast = FALSE, ...)

## S3 method for class 'array':
unique(x, incomparables = FALSE, MARGIN = 1,
       fromLast = FALSE, ...)
```

Arguments

x	a vector or a data frame or an array or NULL.
incomparables	a vector of values that cannot be compared. FALSE is a special value, meaning that all values can be compared, and may be the only value accepted for methods other than the default. It will be coerced internally to the same type as x.
fromLast	logical indicating if duplication should be considered from the last, i.e., the last (or rightmost) of identical elements will be kept. This only matters for names or dimnames.
...	arguments for particular methods.
MARGIN	the array margin to be held fixed: a single integer.

Details

This is a generic function with methods for vectors, data frames and arrays (including matrices).

The array method calculates for each element of the dimension specified by `MARGIN` if the remaining dimensions are identical to those for an earlier element (in row-major order). This would most commonly be used for matrices to find unique rows (the default) or columns (with `MARGIN = 2`).

Note that unlike the Unix command `uniq` this omits *duplicated* and not just *repeated* elements/rows. That is, an element is omitted if it is identical to any previous element and not just if it is the same as the immediately previous one. (For the latter, see `rle`).

Missing values are regarded as equal, but `NaN` is not equal to `NA_real_`.

Values in `incomparables` will never be marked as duplicated. This is intended to be used for a fairly small set of values and will not be efficient for a very large set.

Value

For a vector, an object of the same type of `x`, but with only one copy of each duplicated element. No attributes are copied (so the result has no names).

For a data frame, a data frame is returned with the same columns but possibly fewer rows (and with row names from the first occurrences of the unique rows).

A matrix or array is subsetted by `[, drop = FALSE]`, so dimensions and dimnames are copied appropriately, and the result always has the same number of dimensions as `x`.

Warning

Using this for lists is potentially slow, especially if the elements are not atomic vectors (see `vector`) or differ only in their attributes. In the worst case it is $O(n^2)$.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`duplicated` which gives the indices of duplicated elements.

`rle` which is the equivalent of the Unix `uniq -c` command.

Examples

```
x <- c(3:5, 11:8, 8 + 0:5)
(ux <- unique(x))
(u2 <- unique(x, fromLast = TRUE)) # different order
stopifnot(identical(sort(ux), sort(u2)))

length(unique(sample(100, 100, replace=TRUE)))
## approximately 100(1 - 1/e) = 63.21

unique(iris)
```

`unlink` *Delete Files and Directories*

Description

`unlink` deletes the file(s) or directories specified by `x`.

Usage

```
unlink(x, recursive = FALSE)
```

Arguments

`x`	a character vector with the names of the file(s) or directories to be deleted. Wildcards (normally '*' and '?') are allowed.
`recursive`	logical. Should directories be deleted recursively?

Details

If `recursive = FALSE` directories are not deleted, not even empty ones.

On most platforms 'file' includes symbolic links, fifos and sockets.

Wildcard expansion is done by the internal code of `Sys.glob`. Wildcards never match a leading '.' in the filename, and files '.' and '..' will never be considered for deletion. Wildcards will only be expanded if the system supports it. Most systems will support not only '*' and '?') but character classes such as '[a-z]' (see the `man` pages for `glob`). The metacharacters `* ? [` can occur in Unix filenames, and this makes it difficult to use `unlink` to delete such files (see `file.remove`), although escaping the metacharacters by backslashes usually works. As from R 2.8.0, if a metacharacter matches nothing it is considered as a literal character.

`recursive = TRUE` is not supported on all platforms, and may be ignored, with a warning.

Value

`0` for success, `1` for failure. Not deleting a non-existent file is not a failure, nor is being unable to delete a directory if `recursive = FALSE`. However, missing values in `x` result are regarded as failures.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`file.remove`.

`unlist` *Flatten Lists*

Description

Given a list structure `x`, `unlist` simplifies it to produce a vector which contains all the atomic components which occur in `x`.

Usage

```
unlist(x, recursive = TRUE, use.names = TRUE)
```

Arguments

`x`	an R object, typically a list or vector.
`recursive`	logical. Should unlisting be applied to list components of `x`?
`use.names`	logical. Should names be preserved?

Details

`unlist` is generic: you can write methods to handle specific classes of objects, see InternalMethods, and note, e.g., `relist` with the `unlist` method for `relistable` objects.

If `recursive = FALSE`, the function will not recurse beyond the first level items in `x`.

Factors are treated specially. If all non-list elements of `x` are factors (or ordered factors) then the result will be a factor with levels the union of the level sets of the elements, in the order the levels occur in the level sets of the elements (which means that if all the elements have the same level set, that is the level set of the result).

`x` can be an atomic vector, but then `unlist` does nothing useful, not even drop names.

By default, `unlist` tries to retain the naming information present in `x`. If `use.names = FALSE` all naming information is dropped.

Where possible the list elements are coerced to a common mode during the unlisting, and so the result often ends up as a character vector. Vectors will be coerced to the highest type of the components in the hierarchy NULL < raw < logical < integer < real < complex < character < list < expression: pairlists are treated as lists.

A list is a (generic) vector, and the simplified vector might still be a list (and might be unchanged). Non-vector elements of the list (for example language elements such as names, formulas and calls) are not coerced, and so a list containing one or more of these remains a list. (The effect of unlisting an `lm` fit is a list which has individual residuals as components.)

Value

`NULL` or an expression or a vector of an appropriate mode to hold the list components.

The output type is determined from the highest type of the components in the hierarchy NULL < raw < logical < integer < real < complex < character < list < expression, after coercion of pairlists to lists.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

c, as.list, relist.

Examples

```
unlist(options())
unlist(options(), use.names=FALSE)

l.ex <- list(a = list(1:5, LETTERS[1:5]), b = "Z", c = NA)
unlist(l.ex, recursive = FALSE)
unlist(l.ex, recursive = TRUE)

l1 <- list(a="a", b=2, c=pi+2i)
unlist(l1) # a character vector
l2 <- list(a="a", b=as.name("b"), c=pi+2i)
unlist(l2) # remains a list
```

---

unname *Remove 'names' or 'dimnames'*

Description

Remove the names or dimnames attribute of an R object.

Usage

```
unname(obj, force = FALSE)
```

Arguments

obj	an R object.
force	logical; if true, the dimnames are even removed from data.frames. *This argument is currently **experimental** and hence might change!*

Value

Object as obj but without names or dimnames.

Examples

```
require(graphics); require(stats)

## Answering a question on R-help (14 Oct 1999):
col3 <- 750+ 100*rt(1500, df = 3)
breaks <- factor(cut(col3,breaks=360+5*(0:155)))
z <- table(breaks)
z[1:5] # The names are larger than the data ...
barplot(unname(z), axes= FALSE)
```

`UseMethod` *Class Methods*

Description

R possesses a simple generic function mechanism which can be used for an object-oriented style of programming. Method dispatch takes place based on the class(es) of the first argument to the generic function or of the object supplied as an argument to `UseMethod` or `NextMethod`.

Usage

```
UseMethod(generic, object)

NextMethod(generic = NULL, object = NULL, ...)
```

Arguments

`generic`	a character string naming a function (and not a built-in operator). Required for `UseMethod`.
`object`	for `UseMethod`: an object whose class will determine the method to be dispatched. Defaults to the first argument of the enclosing function.
`...`	further arguments to be passed to the next method.

Details

An R object is a data object which has a `class` attribute (and this can be tested by `is.object`). A class attribute is a character vector giving the names of the classes from which the object *inherits*. If the object does not have a class attribute, it has an implicit class. Matrices and arrays have class `"matrix"` or`"array"` followed by the class of the underlying vector. Most vectors have class the result of `mode(x)`, except that integer vectors have class `c("integer", "numeric")` and real vectors have class `c("double", "numeric")`.

When a function calling `UseMethod("fun")` is applied to an object with class attribute `c("first", "second")`, the system searches for a function called `fun.first` and, if it finds it, applies it to the object. If no such function is found a function called `fun.second` is tried. If no class name produces a suitable function, the function `fun.default` is used, if it exists, or an error results.

Function `methods` can be used to find out about the methods for a particular generic function or class.

`UseMethod` is a primitive function so positional matching is used and names of supplied arguments are ignored. It is not the only means of dispatch of methods, for there are internal generic and group generic functions. `UseMethod` currently dispatches on the implicit class even for arguments that are not objects, but the other means of dispatch do not.

`NextMethod` invokes the next method (determined by the class vector, either of the object supplied to the generic, or of the first argument to the function containing `NextMethod` if a method was invoked directly). Normally `NextMethod` is used with only one argument, `generic`, but if further arguments are supplied these modify the call to the next method.

`NextMethod` should not be called except in methods called by `UseMethod` or from internal generics (see InternalGenerics). In particular it will not work inside anonymous calling functions (e.g. `get("print.ts")(AirPassengers)`).

Name spaces can register methods for generic functions. To support this, `UseMethod` and `NextMethod` search for methods in two places: first in the environment in which the generic function is called, and then in the registration data base for the environment in which the generic is defined (typically a name space). So methods for a generic function need to be available in the environment of the call to the generic, or they must be registered. (It does not matter whether they are visible in the environment in which the generic is defined.)

Technical Details

Now for some obscure details that need to appear somewhere. These comments will be slightly different than those in Chambers(1992). (See also the draft 'R Language Definition'.) `UseMethod` creates a new function call with arguments matched as they came in to the generic. Any local variables defined before the call to `UseMethod` are retained (unlike S). Any statements after the call to `UseMethod` will not be evaluated as `UseMethod` does not return. `UseMethod` can be called with more than two arguments: a warning will be given and additional arguments ignored. (They are not completely ignored in S.) If it is called with just one argument, the class of the first argument of the enclosing function is used as `object`: unlike S this is the first actual argument passed and not the current value of the object of that name.

`NextMethod` works by creating a special call frame for the next method. If no new arguments are supplied, the arguments will be the same in number, order and name as those to the current method but their values will be promises to evaluate their name in the current method and environment. Any named arguments matched to `...` are handled specially: they either replace existing arguments of the same name or are appended to the argument list. They are passed on as the promise that was supplied as an argument to the current environment. (S does this differently!) If they have been evaluated in the current (or a previous environment) they remain evaluated. (This is a complex area, and subject to change: see the draft 'R Language Definition'.)

The search for methods for `NextMethod` is slightly different from that for `UseMethod`. Finding no `fun.default` is not necessarily an error, as the search continues to the generic itself. This is to pick up an internal generic like `[` which has no separate default method, and succeeds only if the generic is a primitive function or a wrapper for a `.Internal` function of the same name. (When a primitive is called as the default method, argument matching may not work as described above due to the different semantics of primitives.)

You will see objects such as `.Generic`, `.Method`, and `.Class` used in methods. These are set in the environment within which the method is evaluated by the dispatch mechanism, which is as follows:

1. Find the context for the calling function (the generic): this gives us the unevaluated arguments for the original call.
2. Evaluate the object (usually an argument) to be used for dispatch, and find a method (possibly the default method) or throw an error.
3. Create an environment for evaluating the method and insert special variables (see below) into that environment. Also copy any variables in the environment of the generic that are not formal (or actual) arguments.
4. Fix up the argument list to be the arguments of the call matched to the formals of the method.

`.Generic` is a length-one character vector naming the generic function.

`.Method` is a character vector (normally of length one) naming the method function. (For functions in the group generic `Ops` it is of length two.)

`.Class` is a character vector of classes used to find the next method. `NextMethod` adds an attribute `"previous"` to `.Class` giving the `.Class` last used for dispatch, and shifts `.Class` along to that used for dispatch.

`.GenericCallEnv` and `.GenericDefEnv` are the environments of the call to be generic and defining the generic respectively. (The latter is used to find methods registered for the generic.)

Note that `.Class` is set when the generic is called, and is unchanged if the class of the dispatching argument is changed in a method. It is possible to change the method that `NextMethod` would dispatch by manipulating `.Class`, but 'this is not recommended unless you understand the inheritance mechanism thoroughly' (Chambers & Hastie, 1992, p. 469).

Note

This scheme is called *S3* (S version 3). For new projects, it is recommended to use the more flexible and robust *S4* scheme provided in the **methods** package.

References

Chambers, J. M. (1992) *Classes and methods: object-oriented programming in S.* Appendix A of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

The draft 'R Language Definition'.

`methods`, `class`, `getS3method`, `is.object`.

`UserHooks` *Functions to Get and Set Hooks for Load, Attach, Detach and Unload*

Description

These functions allow users to set actions to be taken before packages are attached/detached and name spaces are (un)loaded.

Usage

```
getHook(hookName)
setHook(hookName, value,
        action = c("append", "prepend", "replace"))

packageEvent(pkgname,
             event = c("onLoad", "attach", "detach", "onUnload"))
```

Arguments

`hookName`	character string: the hook name
`pkgname`	character string: the package/name space name. If versioned install has been used, `pkgname` should be the unversioned name of the package (but any version information will be stripped).
`event`	character string: an event for the package
`value`	A function, or for `action="replace"`, `NULL`.
`action`	The action to be taken. The names can be abbreviated.

Details

setHook provides a general mechanism for users to register hooks, a list of functions to be called from system (or user) functions. The initial set of hooks is associated with events on packages/name spaces: these hooks are named via calls to packageEvent.

To remove a hook completely, call setHook(hookName, NULL, "replace").

When an R package is attached by library, it can call initialization code via a function .First.lib, and when it is detach-ed it can tidy up via a function .Last.lib. Users can add their own initialization code via the hooks provided by these functions, functions which will be called as funname(pkgname, pkgpath) inside a try call. (The attach hook is called after .First.lib and the detach hook before .Last.lib.)

If a package has a name space, there are two further actions, when the name space is loaded (before being attached and after .onLoad is called) and when it is unloaded (after being detached and before .onUnload). Note that code in these hooks is run without the package being on the search path, so objects in the package need to be referred to using the double colon operator as in the example. (Unlike .onLoad, the user hook is run after the name space has been sealed.)

Hooks are normally run in the order shown by getHook, but the "detach" and "onUnload" hooks are run in reverse order so the default for package events is to add hooks 'inside' existing ones.

Note that when an R session is finished, packages are not detached and name spaces are not unloaded, so the corresponding hooks will not be run.

The hooks are stored in the environment .userHooksEnv in the base package, with 'mangled' names.

Value

For getHook function, a list of functions (possible empty). For setHook function, no return value. For packageEvent, the derived hook name (a character string).

See Also

library, detach, loadNamespace.

Other hooks may be added later: plot.new and persp already have them.

Examples

```
setHook(packageEvent("grDevices", "onLoad"),
        function(...) grDevices::ps.options(horizontal=FALSE))
```

---

utf8Conversion *Convert to or from UTF-8-encoded Character Vectors*

Description

Conversion of UTF-8 encoded character vectors to and from integer vectors.

Usage

```
utf8ToInt(x)
intToUtf8(x, multiple = FALSE)
```

Arguments

x	object to be converted.
multiple	logical: should the conversion be to a single character string or multiple individual characters?

Details

These will work in any locale, including on machines that do not otherwise support multi-byte character sets.

Value

utf8ToInt converts a length-one character string encoded in UTF-8 to an integer vector of (numeric) UTF-8 code points.

intToUtf8 converts a vector of (numeric) UTF-8 code points either to a single character string or a character vector of single characters. (For a single character string 0 is silently omitted: otherwise 0 is mapped to "". Non-integral numeric values are truncated to integers.) The Encoding is declared as "UTF-8".

Examples

```
## Not run:
## will only display in some locales and fonts
intToUtf8(0x03B2L) # Greek beta
## End(Not run)
```

---

vector *Vectors*

Description

vector produces a vector of the given length and mode.

as.vector, a generic, attempts to coerce its argument into a vector of mode mode (the default is to coerce to whichever mode is most convenient).

is.vector returns TRUE if x is a vector (of mode logical, integer, real, complex, character, raw or list if not specified) or expression and FALSE otherwise.

Usage

```
vector(mode = "logical", length = 0)
as.vector(x, mode = "any")
is.vector(x, mode = "any")
```

Arguments

mode	A character string giving an atomic mode or "list", or (not for vector) "any".
length	A non-negative integer specifying the desired length.
x	An object.

Details

The atomic modes are "logical", "integer", "numeric", "complex", "character" and "raw".

is.vector returns FALSE if x has any attributes except names. (This is incompatible with S.) On the other hand, as.vector removes *all* attributes including names for results of atomic mode.

Note that factors are *not* vectors; is.vector returns FALSE and as.vector converts to a character vector for mode = "any".

Value

For vector, a vector of the given length and mode. Logical vector elements are initialized to FALSE, numeric vector elements to 0, character vector elements to "", raw vector elements to nul bytes and list elements to NULL.

All attributes are removed from the answer if it is of an atomic mode.

Note

as.vector and is.vector are quite distinct from the meaning of the formal class "vector" in the **methods** package, and hence as(x, "vector") and is(x, "vector").

modes of "symbol", "pairlist" and "expression" are allowed but have long been undocumented.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

c, is.numeric, is.list, etc.

Examples

```
df <- data.frame(x=1:3, y=5:7)
## Not run:
## Error:
  as.vector(data.frame(x=1:3, y=5:7), mode="numeric")
## End(Not run)

x <- c(a = 1, b = 2)
is.vector(x)
as.vector(x)
all.equal(x, as.vector(x)) ## FALSE

###-- All the following are TRUE:
is.list(df)
! is.vector(df)
! is.vector(df, mode="list")

is.vector(list(), mode="list")
is.vector(NULL,   mode="NULL")
```

warning *Warning Messages*

Description

Generates a warning message that corresponds to its argument(s) and (optionally) the expression or function from which it was called.

Usage

```
warning(..., call. = TRUE, immediate. = FALSE, domain = NULL)
suppressWarnings(expr)
```

Arguments

...	zero or more objects which can be coerced to character (and which are pasted together with no separator) or a single condition object.
call.	logical, indicating if the call should become part of the warning message.
immediate.	logical, indicating if the call should be output immediately, even if getOption("warn") <= 0.
expr	expression to evaluate.
domain	see gettext. If NA, messages will not be translated.

Details

The result *depends* on the value of options("warn") and on handlers established in the executing code.

If a condition object is supplied it should be the only argument, and further arguments will be ignored, with a message.

warning signals a warning condition by (effectively) calling signalCondition. If there are no handlers or if all handlers return, then the value of warn = getOption("warn") is used to determine the appropriate action. If warn is negative warnings are ignored; if it is zero they are stored and printed after the top–level function has completed; if it is one they are printed as they occur and if it is 2 (or larger) warnings are turned into errors. Calling warning(immediate. = TRUE) turns warn <= 0 into warn = 1 for this call only.

If warn is zero (the default), a read-only variable last.warning is created. It contains the warnings which can be printed via a call to warnings.

Warnings will be truncated to getOption("warning.length") characters, default 1000, indicated by [... truncated].

While the warning is being processed, a muffleWarning restart is available. If this restart is invoked with invokeRestart, then warning returns immediately.

An attempt is made to coerce other types of inputs to warning to character vectors.

suppressWarnings evaluates its expression in a context that ignores all warnings.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`stop` for fatal errors, `message` for diagnostic messages, `warnings`, and `options` with argument `warn=`.

`gettext` for the mechanisms for the automated translation of messages.

Examples

```
testit <- function() warning("testit")
testit() ## shows call
testit <- function() warning("problem in testit", call. = FALSE)
testit() ## no call
suppressWarnings(warning("testit"))
```

`warnings` *Print Warning Messages*

Description

`warnings` and its `print` method print the variable `last.warning` in a pleasing form.

Usage

```
warnings(...)
```

Arguments

`...`	arguments to be passed to `cat`.

Details

See the description of `options("warn")` for the circumstances under which there is a `last.warning` object and `warnings()` is used. In essence this is if `options(warn = 0)` and `warning` has been called at least once.

It is possible that `last.warning` refers to the last recorded warning and not to the last warning, for example if `options(warn)` has been changed or if a catastrophic error occurred.

Warning

It is undocumented where `last.warning` is stored nor that it is visible, and this is subject to change. Prior to R 2.4.0 it was stored in the workspace, but no longer.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`warning`.

Examples

```
## NB this example is intended to be pasted in,
##    rather than run by example()
ow <- options("warn")
for(w in -1:1) {
   options(warn = w); cat("\n warn =",w,"\n")
   for(i in 1:3) { cat(i,"..\n"); m <- matrix(1:7, 3,4) }
}
warnings()
options(ow) # reset
```

---

weekdays *Extract Parts of a POSIXt or Date Object*

Description

Extract the weekday, month or quarter, or the Julian time (days since some origin). These are generic functions: the methods for the internal date-time classes are documented here.

Usage

```
weekdays(x, abbreviate)
## S3 method for class 'POSIXt':
weekdays(x, abbreviate = FALSE)
## S3 method for class 'Date':
weekdays(x, abbreviate = FALSE)

months(x, abbreviate)
## S3 method for class 'POSIXt':
months(x, abbreviate = FALSE)
## S3 method for class 'Date':
months(x, abbreviate = FALSE)

quarters(x, abbreviate)
## S3 method for class 'POSIXt':
quarters(x, ...)
## S3 method for class 'Date':
quarters(x, ...)

julian(x, ...)
## S3 method for class 'POSIXt':
julian(x, origin = as.POSIXct("1970-01-01", tz="GMT"), ...)
## S3 method for class 'Date':
julian(x, origin = as.Date("1970-01-01"), ...)
```

Arguments

x	an object inheriting from class "POSIXt" or "Date".
abbreviate	logical. Should the names be abbreviated?
origin	an length-one object inheriting from class "POSIXt" or "Date".
...	arguments for other methods.

Value

`weekdays` and `months` return a character vector of names in the locale in use.

`quarters` returns a character vector of `"Q1"` to `"Q4"`.

`julian` returns the number of days (possibly fractional) since the origin, with the origin as a `"origin"` attribute.

Note

Other components such as the day of the month or the year are very easy to compute: just use `as.POSIXlt` and extract the relevant component.

See Also

`DateTimeClasses`, `Date`

Examples

```
weekdays(.leap.seconds)
months(.leap.seconds)
quarters(.leap.seconds)

## Julian Day Number (JDN, http://en.wikipedia.org/wiki/Julian_day)
## is the number of days since noon UTC on the first day of 4317 BC.
julian(Sys.Date(), -2440588) # for a day
floor(as.numeric(julian(Sys.time())) + 2440587.5) # for a date-time
```

`which` *Which indices are TRUE?*

Description

Give the `TRUE` indices of a logical object, allowing for array indices.

Usage

```
which(x, arr.ind = FALSE)
```

Arguments

`x`	a `logical` vector or array. `NA`s are allowed and omitted (treated as if `FALSE`).
`arr.ind`	logical; should **arr**ay **ind**ices be returned when `x` is an array?

Value

If `arr.ind == FALSE` (the default), an integer vector with `length` equal to `sum(x)`, i.e., to the number of `TRUE`s in `x`; Basically, the result is `(1:length(x))[x]`.

If `arr.ind == TRUE` and `x` is an `array` (has a `dim` attribute), the result is a matrix whose rows each are the indices of one element of `x`; see Examples below.

Author(s)

Werner Stahel and Peter Holzer ⟨holzer@stat.math.ethz.ch⟩, for the array case.

See Also

Logic, which.min for the index of the minimum or maximum, and match for the first index of an element in a vector, i.e., for a scalar a, match(a,x) is equivalent to min(which(x == a)) but much more efficient.

Examples

```
which(LETTERS == "R")
which(ll <- c(TRUE,FALSE,TRUE,NA,FALSE,FALSE,TRUE))#> 1 3 7
names(ll) <- letters[seq(ll)]
which(ll)
which((1:12)%%2 == 0) # which are even?
which(1:10 > 3, arr.ind=TRUE)

( m <- matrix(1:12,3,4) )
which(m %% 3 == 0)
which(m %% 3 == 0, arr.ind=TRUE)
rownames(m) <- paste("Case",1:3, sep="_")
which(m %% 5 == 0, arr.ind=TRUE)

dim(m) <- c(2,2,3); m
which(m %% 3 == 0, arr.ind=FALSE)
which(m %% 3 == 0, arr.ind=TRUE)

vm <- c(m)
dim(vm) <- length(vm) #-- funny thing with  length(dim(...)) == 1
which(vm %% 3 == 0, arr.ind=TRUE)
```

---

which.min *Where is the Min() or Max() ?*

Description

Determines the location, i.e., index of the (first) minimum or maximum of a numeric vector.

Usage

```
which.min(x)
which.max(x)
```

Arguments

x	numeric (integer or double) vector, whose min or max is searched for.

Value

Missing and NaN values are discarded.

an integer of length 1 or 0 (iff x has no non-NAs), giving the index of the *first* minimum or maximum respectively of x.

If this extremum is unique (or empty), the results are the same as (but more efficient than) which(x == min(x)) or which(x == max(x)) respectively.

Author(s)

Martin Maechler

See Also

`which`, `max.col`, `max`, etc.

`which.is.max` in package **nnet** differs in breaking ties at random (and having a 'fuzz' in the definition of ties).

Examples

```
x <- c(1:4,0:5,11)
which.min(x)
which.max(x)

## it *does* work with NA's present, by discarding them:
presidents[1:30]
range(presidents, na.rm = TRUE)
which.min(presidents) # 28
which.max(presidents) #  2
```

---

with *Evaluate an Expression in a Data Environment*

Description

Evaluate an R expression in an environment constructed from data, possibly modifying the original data.

Usage

```
with(data, expr, ...)
within(data, expr, ...)
```

Arguments

`data`	data to use for constructing an environment. For the default `with` method this may be an environment, a list, a data frame, or an integer as in `sys.call`. For `within`, it can be a list or a data frame.
`expr`	expression to evaluate.
`...`	arguments to be passed to future methods.

Details

`with` is a generic function that evaluates `expr` in a local environment constructed from `data`. The environment has the caller's environment as its parent. This is useful for simplifying calls to modeling functions. (Note: if `data` is already an environment then this is used with its existing parent.)

Note that assignments within `expr` take place in the constructed environment and not in the user's workspace.

`within` is similar, except that it examines the environment after the evaluation of `expr` and makes the corresponding modifications to `data` (this may fail in the data frame case if objects are created which cannot be stored in a data frame), and returns it. `within` can be used as an alternative to `transform`.

Value

For `with`, the value of the evaluated `expr`. For `within`, the modified object.

See Also

`evalq`, `attach`, `assign`, `transform`.

Examples

```
require(stats); require(graphics)
#examples from glm:
## Not run:
library(MASS)
with(anorexia, {
    anorex.1 <- glm(Postwt ~ Prewt + Treat + offset(Prewt),
                    family = gaussian)
    summary(anorex.1)
})
## End(Not run)


aq <- within(airquality, {     # Notice that multiple vars can be changed
    lOzone<-log(Ozone)
    Month<-factor(month.abb[Month])
    cTemp <- round((Temp - 32) * 5/9, 1) # From Fahrenheit to Celsius
    rm(Day, Temp)
})
head(aq)

with(data.frame(u = c(5,10,15,20,30,40,60,80,100),
                lot1 = c(118,58,42,35,27,25,21,19,18),
                lot2 = c(69,35,26,21,18,16,13,12,12)),
    list(summary(glm(lot1 ~ log(u), family = Gamma)),
         summary(glm(lot2 ~ log(u), family = Gamma))))

# example from boxplot:
with(ToothGrowth, {
    boxplot(len ~ dose, boxwex = 0.25, at = 1:3 - 0.2,
            subset = (supp == "VC"), col = "yellow",
            main = "Guinea Pigs' Tooth Growth",
            xlab = "Vitamin C dose mg",
            ylab = "tooth length", ylim = c(0,35))
    boxplot(len ~ dose, add = TRUE, boxwex = 0.25, at = 1:3 + 0.2,
            subset = supp == "OJ", col = "orange")
    legend(2, 9, c("Ascorbic acid", "Orange juice"),
           fill = c("yellow", "orange"))
})

# alternate form that avoids subset argument:
with(subset(ToothGrowth, supp == "VC"),
     boxplot(len ~ dose, boxwex = 0.25, at = 1:3 - 0.2,
```

```
              col = "yellow", main = "Guinea Pigs' Tooth Growth",
              xlab = "Vitamin C dose mg",
              ylab = "tooth length", ylim = c(0,35)))
with(subset(ToothGrowth,  supp == "OJ"),
     boxplot(len ~ dose, add = TRUE, boxwex = 0.25, at = 1:3 + 0.2,
              col = "orange"))
legend(2, 9, c("Ascorbic acid", "Orange juice"),
       fill = c("yellow", "orange"))
```

`write` *Write Data to a File*

Description

The data (usually a matrix) `x` are written to file `file`. If `x` is a two-dimensional matrix you need to transpose it to get the columns in `file` the same as those in the internal representation.

Usage

```
write(x, file = "data",
      ncolumns = if(is.character(x)) 1 else 5,
      append = FALSE, sep = " ")
```

Arguments

`x`	the data to be written out.
`file`	A connection, or a character string naming the file to write to. If `""`, print to the standard output connection. If it is `"|cmd"`, the output is piped to the command given by 'cmd'.
`ncolumns`	the number of columns to write the data in.
`append`	if `TRUE` the data `x` are appended to the connection.
`sep`	a string used to separate columns. Using `sep = "\t"` gives tab delimited output; default is `" "`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`write` is a wrapper for `cat`, which gives further details on the format used.

`save` for writing any R objects, `write.table` for data frames, and `scan` for reading data.

Examples

```
# create a 2 by 5 matrix
x <- matrix(1:10,ncol=5)

# the file data contains x, two rows, five cols
# 1 3 5 7 9 will form the first row
write(t(x))

# Writing to the "console" 'tab-delimited'
# two rows, five cols but the first row is 1 2 3 4 5
write(x, "", sep = "\t")
unlink("data") # tidy up
```

---

writeLines *Write Lines to a Connection*

Description

Write text lines to a connection.

Usage

```
writeLines(text, con = stdout(), sep = "\n")
```

Arguments

text	A character vector
con	A connection object or a character string.
sep	character. A string to be written to the connection after each line of text.

Details

If the con is a character string, the function calls file to obtain a file connection which is opened for the duration of the function call.

If the connection is open it is written from its current position. If it is not open, it is opened for the duration of the call in "wt" mode and then closed again.

Normally writeLines is used with a text-mode connection, and the default separator is converted to the normal separator for that platform (LF on Unix/Linux, CRLF on Windows). For more control, open a binary connection and specify the precise value you want written to the file in sep. For even more control, use writeChar on a binary connection.

See Also

connections, writeChar, writeBin, readLines, cat

xtfrm *Auxiliary Function for Sorting and Ranking*

Description

A generic auxiliary function that produces a numeric vector which will sort in the same order as `x`.

Usage

```
xtfrm(x)
```

Arguments

`x`	an R object.

Details

This is a special case of ranking, but as a less general function than `rank` is more suitable to be made generic. The default method is equivalent to `rank(x, ties.method="min", na.last="keep")`, so `NA` values are given rank `NA` and all tied values are given equal integer rank.

The `factor` method extracts the codes. The `Surv` method sorts first on times and then on status code(s).

The default method will make use of ==, > and `is.na` methods for the class of `x`, but might be rather slow when doing so.

Value

A numeric (usually integer) vector of the same length as `x`.

See Also

`rank`, `sort`, `order`.

zpackages *Listing of Packages*

Description

`.packages` returns information about package availability.

Usage

```
.packages(all.available = FALSE, lib.loc = NULL)
```

Arguments

`all.available`	logical; if `TRUE` return a character vector of all available packages in `lib.loc`.
`lib.loc`	a character vector describing the location of R library trees to search through, or `NULL`. The default value of `NULL` corresponds to all libraries currently known.

Details

`.packages()` returns the names of the currently attached packages *invisibly* whereas `.packages(all.available = TRUE)` gives (visibly) *all* packages available in the library location path `lib.loc`. If versioned installs have been used, the names returned will be of the form `pkgname_version`.

For a package to be regarded as being available it must have a 'DESCRIPTION' file containing a valid `version` field.

Value

A character vector of package base names, invisible unless `all.available = TRUE`.

Author(s)

R core; Guido Masarotto for the `all.available=TRUE` part of `.packages`.

See Also

`library`, `.libPaths`.

Examples

```
(.packages())                # maybe just "base"
.packages(all.available = TRUE) # return all available as character vector
require(splines)
(.packages())                # "splines", too
detach("package:splines")
```

---

zutils *Miscellaneous Internal/Programming Utilities*

Description

Miscellaneous internal/programming utilities.

Usage

```
.standard_regexps()
```

Details

`.standard_regexps` returns a list of 'standard' regexps, including elements named `valid_package_name` and `valid_package_version` with the obvious meanings. The regexps are not anchored.

Chapter 2

The `datasets` package

`datasets-package` *The R Datasets Package*

Description

Base R datasets

Details

This package contains a variety of datasets. For a complete list, use `library(help="datasets")`.

Author(s)

R Development Core Team and contributors worldwide

Maintainer: R Core Team ⟨R-core@r-project.org⟩

`ability.cov` *Ability and Intelligence Tests*

Description

Six tests were given to 112 individuals. The covariance matrix is given in this object.

Usage

```
ability.cov
```

Details

The tests are described as

general: a non-verbal measure of general intelligence using Cattell's culture-fair test.

picture: a picture-completion test

blocks: block design

maze: mazes

reading: reading comprehension

vocab: vocabulary

Bartholomew gives both covariance and correlation matrices, but these are inconsistent. Neither are in the original paper.

Source

Bartholomew, D. J. (1987) *Latent Variable Analysis and Factor Analysis.* Griffin.

Bartholomew, D. J. and Knott, M. (1990) *Latent Variable Analysis and Factor Analysis.* Second Edition, Arnold.

References

Smith, G. A. and Stanley G. (1983) Clocking g: relating intelligence and measures of timed performance. *Intelligence*, **7**, 353–368.

Examples

```
require(stats)
(ability.FA <- factanal(factors = 1, covmat=ability.cov))
update(ability.FA, factors=2)
update(ability.FA, factors=2, rotation="promax")
```

---

airmiles *Passenger Miles on Commercial US Airlines, 1937–1960*

Description

The revenue passenger miles flown by commercial airlines in the United States for each year from 1937 to 1960.

Usage

```
airmiles
```

Format

A time series of 24 observations; yearly, 1937–1960.

Source

F.A.A. Statistical Handbook of Aviation.

References

Brown, R. G. (1963) *Smoothing, Forecasting and Prediction of Discrete Time Series*. Prentice-Hall.

Examples

```
require(graphics)
plot(airmiles, main = "airmiles data",
     xlab = "Passenger-miles flown by U.S. commercial airlines", col = 4)
```

AirPassengers *Monthly Airline Passenger Numbers 1949-1960*

Description

The classic Box & Jenkins airline data. Monthly totals of international airline passengers, 1949 to 1960.

Usage

```
AirPassengers
```

Format

A monthly time series, in thousands.

Source

Box, G. E. P., Jenkins, G. M. and Reinsel, G. C. (1976) *Time Series Analysis, Forecasting and Control.* Third Edition. Holden-Day. Series G.

Examples

```
## Not run:
## These are quite slow and so not run by example(AirPassengers)

## The classic 'airline model', by full ML
(fit <- arima(log10(AirPassengers), c(0, 1, 1),
              seasonal = list(order=c(0, 1 ,1), period=12)))
update(fit, method = "CSS")
update(fit, x=window(log10(AirPassengers), start = 1954))
pred <- predict(fit, n.ahead = 24)
tl <- pred$pred - 1.96 * pred$se
tu <- pred$pred + 1.96 * pred$se
ts.plot(AirPassengers, 10^tl, 10^tu, log = "y", lty = c(1,2,2))

## full ML fit is the same if the series is reversed, CSS fit is not
ap0 <- rev(log10(AirPassengers))
attributes(ap0) <- attributes(AirPassengers)
arima(ap0, c(0, 1, 1), seasonal = list(order=c(0, 1 ,1), period=12))
arima(ap0, c(0, 1, 1), seasonal = list(order=c(0, 1 ,1), period=12),
      method = "CSS")

## Structural Time Series
```

```
ap <- log10(AirPassengers) - 2
(fit <- StructTS(ap, type= "BSM"))
par(mfrow=c(1,2))
plot(cbind(ap, fitted(fit)), plot.type = "single")
plot(cbind(ap, tsSmooth(fit)), plot.type = "single")
## End(Not run)
```

`airquality` *New York Air Quality Measurements*

Description

Daily air quality measurements in New York, May to September 1973.

Usage

```
airquality
```

Format

A data frame with 154 observations on 6 variables.

`[,1]`	`Ozone`	numeric	Ozone (ppb)
`[,2]`	`Solar.R`	numeric	Solar R (lang)
`[,3]`	`Wind`	numeric	Wind (mph)
`[,4]`	`Temp`	numeric	Temperature (degrees F)
`[,5]`	`Month`	numeric	Month (1–12)
`[,6]`	`Day`	numeric	Day of month (1–31)

Details

Daily readings of the following air quality values for May 1, 1973 (a Tuesday) to September 30, 1973.

- `Ozone`: Mean ozone in parts per billion from 1300 to 1500 hours at Roosevelt Island
- `Solar.R`: Solar radiation in Langleys in the frequency band 4000–7700 Angstroms from 0800 to 1200 hours at Central Park
- `Wind`: Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport
- `Temp`: Maximum daily temperature in degrees Fahrenheit at La Guardia Airport.

Source

The data were obtained from the New York State Department of Conservation (ozone data) and the National Weather Service (meteorological data).

References

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis*. Belmont, CA: Wadsworth.

Examples

```
require(graphics)
pairs(airquality, panel = panel.smooth, main = "airquality data")
```

`anscombe` *Anscombe's Quartet of "Identical" Simple Linear Regressions*

Description

Four x-y datasets which have the same traditional statistical properties (mean, variance, correlation, regression line, etc.), yet are quite different.

Usage

```
anscombe
```

Format

A data frame with 11 observations on 8 variables.

x1 == x2 == x3	the integers 4:14, specially arranged
x4	values 8 and 19
y1, y2, y3, y4	numbers in (3, 12.5) with mean 7.5 and sdev 2.03

Source

Tufte, Edward R. (1989) *The Visual Display of Quantitative Information*, 13–14. Graphics Press.

References

Anscombe, Francis J. (1973) Graphs in statistical analysis. *American Statistician*, **27**, 17–21.

Examples

```
require(stats); require(graphics)
summary(anscombe)

##-- now some "magic" to do the 4 regressions in a loop:
ff <- y ~ x
for(i in 1:4) {
  ff[2:3] <- lapply(paste(c("y","x"), i, sep=""), as.name)
  ## or   ff[[2]] <- as.name(paste("y", i, sep=""))
  ##      ff[[3]] <- as.name(paste("x", i, sep=""))
  assign(paste("lm.",i,sep=""), lmi <- lm(ff, data= anscombe))
  print(anova(lmi))
}

## See how close they are (numerically!)
sapply(objects(pattern="lm\\.[1-4]$"), function(n) coef(get(n)))
lapply(objects(pattern="lm\\.[1-4]$"),
       function(n) coef(summary(get(n))))

## Now, do what you should have done in the first place: PLOTS
```

```
op <- par(mfrow=c(2,2), mar=.1+c(4,4,1,1), oma= c(0,0,2,0))
for(i in 1:4) {
  ff[2:3] <- lapply(paste(c("y","x"), i, sep=""), as.name)
  plot(ff, data =anscombe, col="red", pch=21, bg = "orange", cex = 1.2,
       xlim=c(3,19), ylim=c(3,13))
  abline(get(paste("lm.",i,sep="")), col="blue")
}
mtext("Anscombe's 4 Regression data sets", outer = TRUE, cex=1.5)
par(op)
```

`attenu` *The Joyner–Boore Attenuation Data*

Description

This data gives peak accelerations measured at various observation stations for 23 earthquakes in California. The data have been used by various workers to estimate the attenuating affect of distance on ground acceleration.

Usage

```
attenu
```

Format

A data frame with 182 observations on 5 variables.

[,1]	event	numeric	Event Number
[,2]	mag	numeric	Moment Magnitude
[,3]	station	factor	Station Number
[,4]	dist	numeric	Station-hypocenter distance (km)
[,5]	accel	numeric	Peak acceleration (g)

Source

Joyner, W.B., D.M. Boore and R.D. Porcella (1981). Peak horizontal acceleration and velocity from strong-motion records including records from the 1979 Imperial Valley, California earthquake. USGS Open File report 81-365. Menlo Park, Ca.

References

Boore, D. M. and Joyner, W.B.(1982) The empirical prediction of ground motion, *Bull. Seism. Soc. Am.*, **72**, S269–S268.

Bolt, B. A. and Abrahamson, N. A. (1982) New attenuation relations for peak and expected accelerations of strong ground motion, *Bull. Seism. Soc. Am.*, **72**, 2307–2321.

Bolt B. A. and Abrahamson, N. A. (1983) Reply to W. B. Joyner & D. M. Boore's "Comments on: New attenuation relations for peak and expected accelerations for peak and expected accelerations of strong ground motion", *Bull. Seism. Soc. Am.*, **73**, 1481–1483.

Brillinger, D. R. and Preisler, H. K. (1984) An exploratory analysis of the Joyner-Boore attenuation data, *Bull. Seism. Soc. Am.*, **74**, 1441–1449.

Brillinger, D. R. and Preisler, H. K. (1984) *Further analysis of the Joyner-Boore attenuation data*. Manuscript.

Examples

```
require(graphics)
## check the data class of the variables
sapply(attenu, data.class)
summary(attenu)
pairs(attenu, main = "attenu data")
coplot(accel ~ dist | as.factor(event), data = attenu, show.given = FALSE)
coplot(log(accel) ~ log(dist) | as.factor(event),
       data = attenu, panel = panel.smooth, show.given = FALSE)
```

---

attitude *The Chatterjee–Price Attitude Data*

Description

From a survey of the clerical employees of a large financial organization, the data are aggregated from the questionnaires of the approximately 35 employees for each of 30 (randomly selected) departments. The numbers give the percent proportion of favourable responses to seven questions in each department.

Usage

```
attitude
```

Format

A dataframe with 30 observations on 7 variables. The first column are the short names from the reference, the second one the variable names in the data frame:

Y	rating	numeric	Overall rating
X[1]	complaints	numeric	Handling of employee complaints
X[2]	privileges	numeric	Does not allow special privileges
X[3]	learning	numeric	Opportunity to learn
X[4]	raises	numeric	Raises based on performance
X[5]	critical	numeric	Too critical
X[6]	advancel	numeric	Advancement

Source

Chatterjee, S. and Price, B. (1977) *Regression Analysis by Example*. New York: Wiley. (Section 3.7, p.68ff of 2nd ed.(1991).)

Examples

```
require(stats); require(graphics)
pairs(attitude, main = "attitude data")
summary(attitude)
summary(fm1 <- lm(rating ~ ., data = attitude))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
```

```
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
summary(fm2 <- lm(rating ~ complaints, data = attitude))
plot(fm2)
par(opar)
```

`austres` *Quarterly Time Series of the Number of Australian Residents*

Description

Numbers (in thousands) of Australian residents measured quarterly from March 1971 to March 1994. The object is of class `"ts"`.

Usage

```
austres
```

Source

P. J. Brockwell and R. A. Davis (1996) *Introduction to Time Series and Forecasting.* Springer

---

`beavers` *Body Temperature Series of Two Beavers*

Description

Reynolds (1994) describes a small part of a study of the long-term temperature dynamics of beaver *Castor canadensis* in north-central Wisconsin. Body temperature was measured by telemetry every 10 minutes for four females, but data from a one period of less than a day for each of two animals is used there.

Usage

```
beaver1
beaver2
```

Format

The `beaver1` data frame has 114 rows and 4 columns on body temperature measurements at 10 minute intervals.

The `beaver2` data frame has 100 rows and 4 columns on body temperature measurements at 10 minute intervals.

The variables are as follows:

day Day of observation (in days since the beginning of 1990), December 12–13 (`beaver1`) and November 3–4 (`beaver2`).

time Time of observation, in the form `0330` for 3:30am

temp Measured body temperature in degrees Celsius.

activ Indicator of activity outside the retreat.

Note

The observation at 22:20 is missing in `beaver1`.

Source

P. S. Reynolds (1994) Time-series analyses of beaver body temperatures. Chapter 11 of Lange, N., Ryan, L., Billard, L., Brillinger, D., Conquest, L. and Greenhouse, J. eds (1994) *Case Studies in Biometry.* New York: John Wiley and Sons.

Examples

```
require(graphics)
(yl <- range(beaver1$temp, beaver2$temp))

beaver.plot <- function(bdat, ...) {
  nam <- deparse(substitute(bdat))
  with(bdat, {
    # Hours since start of day:
    hours <- time %/% 100 + 24*(day - day[1]) + (time %% 100)/60
    plot (hours, temp, type = "l", ...,
          main = paste(nam, "body temperature"))
    abline(h = 37.5, col = "gray", lty = 2)
    is.act <- activ == 1
    points(hours[is.act], temp[is.act], col = 2, cex = .8)
  })
}
op <- par(mfrow = c(2,1), mar = c(3,3,4,2), mgp = .9* 2:0)
 beaver.plot(beaver1, ylim = yl)
 beaver.plot(beaver2, ylim = yl)
par(op)
```

BJsales *Sales Data with Leading Indicator*

Description

The sales time series `BJsales` and leading indicator `BJsales.lead` each contain 150 observations. The objects are of class `"ts"`.

Usage

```
BJsales
BJsales.lead
```

Source

The data are given in Box & Jenkins (1976). Obtained from the Time Series Data Library at http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/

References

G. E. P. Box and G. M. Jenkins (1976): *Time Series Analysis, Forecasting and Control*, Holden-Day, San Francisco, p. 537.

P. J. Brockwell and R. A. Davis (1991): *Time Series: Theory and Methods*, Second edition, Springer Verlag, NY, pp. 414.

`BOD` *Biochemical Oxygen Demand*

Description

The `BOD` data frame has 6 rows and 2 columns giving the biochemical oxygen demand versus time in an evaluation of water quality.

Usage

```
BOD
```

Format

This data frame contains the following columns:

Time A numeric vector giving the time of the measurement (days).

demand A numeric vector giving the biochemical oxygen demand (mg/l).

Source

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley, Appendix A1.4.

Originally from Marske (1967), *Biochemical Oxygen Demand Data Interpretation Using Sum of Squares Surface* M.Sc. Thesis, University of Wisconsin – Madison.

Examples

```
require(stats)
# simplest form of fitting a first-order model to these data
fm1 <- nls(demand ~ A*(1-exp(-exp(lrc)*Time)), data = BOD,
   start = c(A = 20, lrc = log(.35)))
coef(fm1)
print(fm1)
# using the plinear algorithm
fm2 <- nls(demand ~ (1-exp(-exp(lrc)*Time)), data = BOD,
   start = c(lrc = log(.35)), algorithm = "plinear", trace = TRUE)
# using a self-starting model
fm3 <- nls(demand ~ SSasympOrig(Time, A, lrc), data = BOD)
summary( fm3 )
```

`cars` *Speed and Stopping Distances of Cars*

Description

The data give the speed of cars and the distances taken to stop. Note that the data were recorded in the 1920s.

Usage

```
cars
```

Format

A data frame with 50 observations on 2 variables.

[,1]	speed	numeric	Speed (mph)
[,2]	dist	numeric	Stopping distance (ft)

Source

Ezekiel, M. (1930) *Methods of Correlation Analysis*. Wiley.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(stats); require(graphics)
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     las = 1)
lines(lowess(cars$speed, cars$dist, f = 2/3, iter = 3), col = "red")
title(main = "cars data")
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     las = 1, log = "xy")
title(main = "cars data (logarithmic scales)")
lines(lowess(cars$speed, cars$dist, f = 2/3, iter = 3), col = "red")
summary(fm1 <- lm(log(dist) ~ log(speed), data = cars))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)

## An example of polynomial regression
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
    las = 1, xlim = c(0, 25))
d <- seq(0, 25, length.out = 200)
for(degree in 1:4) {
  fm <- lm(dist ~ poly(speed, degree), data = cars)
  assign(paste("cars", degree, sep="."), fm)
  lines(d, predict(fm, data.frame(speed=d)), col = degree)
}
anova(cars.1, cars.2, cars.3, cars.4)
```

`ChickWeight` *Weight versus age of chicks on different diets*

Description

The `ChickWeight` data frame has 578 rows and 4 columns from an experiment on the effect of diet on early growth of chicks.

Usage

```
ChickWeight
```

Format

This data frame contains the following columns:

weight a numeric vector giving the body weight of the chick (gm).

Time a numeric vector giving the number of days since birth when the measurement was made.

Chick an ordered factor with levels `18` < ... < `48` giving a unique identifier for the chick. The ordering of the levels groups chicks on the same diet together and orders them according to their final weight (lightest to heaviest) within diet.

Diet a factor with levels 1,...,4 indicating which experimental diet the chick received.

Details

The body weights of the chicks were measured at birth and every second day thereafter until day 20. They were also measured on day 21. There were four groups on chicks on different protein diets.

Source

Crowder, M. and Hand, D. (1990), *Analysis of Repeated Measures*, Chapman and Hall (example 5.3)

Hand, D. and Crowder, M. (1996), *Practical Longitudinal Data Analysis*, Chapman and Hall (table A.2)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats); require(graphics)
coplot(weight ~ Time | Chick, data = ChickWeight,
    type = "b", show.given = FALSE)
## fit a representative chick
fm1 <- nls(weight ~ SSlogis( Time, Asym, xmid, scal ),
    data = ChickWeight, subset = Chick == 1)
summary( fm1 )
```

chickwts *Chicken Weights by Feed Type*

Description

An experiment was conducted to measure and compare the effectiveness of various feed supplements on the growth rate of chickens.

Usage

```
chickwts
```

Format

A data frame with 71 observations on 2 variables.

weight a numeric variable giving the chick weight.

feed a factor giving the feed type.

Details

Newly hatched chicks were randomly allocated into six groups, and each group was given a different feed supplement. Their weights in grams after six weeks are given along with feed types.

Source

Anonymous (1948) *Biometrika*, **35**, 214.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(stats); require(graphics)
boxplot(weight ~ feed, data = chickwts, col = "lightgray",
    varwidth = TRUE, notch = TRUE, main = "chickwt data",
    ylab = "Weight at six weeks (gm)")
anova(fm1 <- lm(weight ~ feed, data = chickwts))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)
```

CO2 *Carbon Dioxide uptake in grass plants*

Description

The CO2 data frame has 84 rows and 5 columns of data from an experiment on the cold tolerance of the grass species *Echinochloa crus-galli*.

Usage

```
CO2
```

Format

This data frame contains the following columns:

Plant an ordered factor with levels Qn1 < Qn2 < Qn3 < ... < Mc1 giving a unique identifier for each plant.

Type a factor with levels Quebec Mississippi giving the origin of the plant

Treatment a factor with levels nonchilled chilled

conc a numeric vector of ambient carbon dioxide concentrations (mL/L).

uptake a numeric vector of carbon dioxide uptake rates (μmol/m^2 sec).

Details

The CO_2 uptake of six plants from Quebec and six plants from Mississippi was measured at several levels of ambient CO_2 concentration. Half the plants of each type were chilled overnight before the experiment was conducted.

Source

Potvin, C., Lechowicz, M. J. and Tardif, S. (1990) "The statistical analysis of ecophysiological response curves obtained from experiments involving repeated measures", *Ecology*, **71**, 1389–1400.

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats); require(graphics)
coplot(uptake ~ conc | Plant, data = CO2, show.given = FALSE, type = "b")
## fit the data for the first plant
fm1 <- nls(uptake ~ SSasymp(conc, Asym, lrc, c0),
   data = CO2, subset = Plant == 'Qn1')
summary(fm1)
## fit each plant separately
fmlist <- list()
for (pp in levels(CO2$Plant)) {
  fmlist[[pp]] <- nls(uptake ~ SSasymp(conc, Asym, lrc, c0),
      data = CO2, subset = Plant == pp)
}
## check the coefficients by plant
sapply(fmlist, coef)
```

co2 *Mauna Loa Atmospheric CO2 Concentration*

Description

Atmospheric concentrations of CO_2 are expressed in parts per million (ppm) and reported in the preliminary 1997 SIO manometric mole fraction scale.

Usage

```
co2
```

Format

A time series of 468 observations; monthly from 1959 to 1997.

Details

The values for February, March and April of 1964 were missing and have been obtained by interpolating linearly between the values for January and May of 1964.

Source

Keeling, C. D. and Whorf, T. P., Scripps Institution of Oceanography (SIO), University of California, La Jolla, California USA 92093-0220.

ftp://cdiac.esd.ornl.gov/pub/maunaloa-co2/maunaloa.co2.

References

Cleveland, W. S. (1993) *Visualizing Data*. New Jersey: Summit Press.

Examples

```
require(graphics)
plot(co2, ylab = expression("Atmospheric concentration of CO"[2]),
     las = 1)
title(main = "co2 data set")
```

crimtab *Student's 3000 Criminals Data*

Description

Data of 3000 male criminals over 20 years old undergoing their sentences in the chief prisons of England and Wales.

Usage

```
data(crimtab)
```

Format

A `table` object of `integer` counts, of dimension 42×22 with a total count, `sum(crimtab)` of 3000.

The 42 `rownames` (`"9.4"`, `"9.5"`, ...) correspond to midpoints of intervals of finger lengths whereas the 22 column names (`colnames`) (`"142.24"`, `"144.78"`, ...) correspond to (body) heights of 3000 criminals, see also below.

Details

Student is the pseudonym of William Sealy Gosset. In his 1908 paper he wrote (on page 13) at the beginning of section VI entitled *Practical Test of the forgoing Equations*:

"Before I had succeeded in solving my problem analytically, I had endeavoured to do so empirically. The material used was a correlation table containing the height and left middle finger measurements of 3000 criminals, from a paper by W. R. MacDonell (*Biometrika*, Vol. I., p. 219). The measurements were written out on 3000 pieces of cardboard, which were then very thoroughly shuffled and drawn at random. As each card was drawn its numbers were written down in a book, which thus contains the measurements of 3000 criminals in a random order. Finally, each consecutive set of 4 was taken as a sample—750 in all—and the mean, standard deviation, and correlation of each sample determined. The difference between the mean of each sample and the mean of the population was then divided by the standard deviation of the sample, giving us the z of Section III."

The table is in fact page 216 and not page 219 in MacDonell(1902). In the MacDonell table, the middle finger lengths were given in mm and the heights in feet/inches intervals, they are both converted into cm here. The midpoints of intervals were used, e.g., where MacDonell has $4'7''9/16 - -8''9/16$, we have 142.24 which is 2.54*56 = 2.54*($4'8''$).

MacDonell credited the source of data (page 178) as follows: *The data on which the memoir is based were obtained, through the kindness of Dr Garson, from the Central Metric Office, New Scotland Yard...* He pointed out on page 179 that : *The forms were drawn at random from the mass on the office shelves; we are therefore dealing with a random sampling.*

Source

`http://pbil.univ-lyon1.fr/R/donnees/criminals1902.txt` thanks to Jean R. Lobry and Anne-Béatrice Dufour.

References

Garson, J.G. (1900) The metric system of identification of criminals, as used in in Great Britain and Ireland. *The Journal of the Anthropological Institute of Great Britain and Ireland* **30**, 161–198.

MacDonell, W.R. (1902) On criminal anthropometry and the identification of criminals. *Biometrika* **1**, 2, 177–227.

Student (1908) The probable error of a mean. *Biometrika* **6**, 1–25.

Examples

```
require(stats)
dim(crimtab)
utils::str(crimtab)
## for nicer printing:
local({cT <- crimtab
       colnames(cT) <- substring(colnames(cT), 2,3)
       print(cT, zero.print = " ")
})
```

```
## Repeat Student's experiment:

# 1) Reconstitute 3000 raw data for heights in inches and rounded to
#    nearest integer as in Student's paper:

(heIn <- round(as.numeric(colnames(crimtab)) / 2.54))
d.hei <- data.frame(height = rep(heIn, colSums(crimtab)))

# 2) shuffle the data:

set.seed(1)
d.hei <- d.hei[sample(1:3000), , drop = FALSE]

# 3) Make 750 samples each of size 4:

d.hei$sample <- as.factor(rep(1:750, each = 4))

# 4) Compute the means and standard deviations (n) for the 750 samples:

h.mean <- with(d.hei, tapply(height, sample, FUN = mean))
h.sd   <- with(d.hei, tapply(height, sample, FUN = sd)) * sqrt(3/4)

# 5) Compute the difference between the mean of each sample and
#    the mean of the population and then divide by the
#    standard deviation of the sample:

zobs <- (h.mean - mean(d.hei[,"height"]))/h.sd

# 6) Replace infinite values by +/- 6 as in Student's paper:

zobs[infZ <- is.infinite(zobs)] # 3 of them
zobs[infZ] <- 6 * sign(zobs[infZ])

# 7) Plot the distribution:

require(grDevices); require(graphics)
hist(x = zobs, probability = TRUE, xlab = "Student's z",
     col = grey(0.8), border = grey(0.5),
     main = "Distribution of Student's z score  for 'crimtab' data")
```

`discoveries` *Yearly Numbers of Important Discoveries*

Description

The numbers of "great" inventions and scientific discoveries in each year from 1860 to 1959.

Usage

```
discoveries
```

Format

A time series of 100 values.

Source

The World Almanac and Book of Facts, 1975 Edition, pages 315–318.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(graphics)
plot(discoveries, ylab = "Number of important discoveries",
     las = 1)
title(main = "discoveries data set")
```

DNase *Elisa assay of DNase*

Description

The `DNase` data frame has 176 rows and 3 columns of data obtained during development of an ELISA assay for the recombinant protein DNase in rat serum.

Usage

```
DNase
```

Format

This data frame contains the following columns:

Run an ordered factor with levels `10` < ... < `3` indicating the assay run.

conc a numeric vector giving the known concentration of the protein.

density a numeric vector giving the measured optical density (dimensionless) in the assay. Duplicate optical density measurements were obtained.

Source

Davidian, M. and Giltinan, D. M. (1995) *Nonlinear Models for Repeated Measurement Data*, Chapman & Hall (section 5.2.4, p. 134)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats); require(graphics)
coplot(density ~ conc | Run, data = DNase,
       show.given = FALSE, type = "b")
coplot(density ~ log(conc) | Run, data = DNase,
       show.given = FALSE, type = "b")
## fit a representative run
fm1 <- nls(density ~ SSlogis( log(conc), Asym, xmid, scal ),
    data = DNase, subset = Run == 1)
## compare with a four-parameter logistic
```

```
fm2 <- nls(density ~ SSfpl( log(conc), A, B, xmid, scal ),
    data = DNase, subset = Run == 1)
summary(fm2)
anova(fm1, fm2)
```

`esoph` *Smoking, Alcohol and (O)esophageal Cancer*

Description

Data from a case-control study of (o)esophageal cancer in Ile-et-Vilaine, France.

Usage

```
esoph
```

Format

A data frame with records for 88 age/alcohol/tobacco combinations.

[,1]	"agegp"	Age group	1 25–34 years
			2 35–44
			3 45–54
			4 55–64
			5 65–74
			6 75+
[,2]	"alcgp"	Alcohol consumption	1 0–39 gm/day
			2 40–79
			3 80–119
			4 120+
[,3]	"tobgp"	Tobacco consumption	1 0– 9 gm/day
			2 10–19
			3 20–29
			4 30+
[,4]	"ncases"	Number of cases	
[,5]	"ncontrols"	Number of controls	

Author(s)

Thomas Lumley

Source

Breslow, N. E. and Day, N. E. (1980) *Statistical Methods in Cancer Research. 1: The Analysis of Case-Control Studies.* IARC Lyon / Oxford University Press.

Examples

```
require(stats)
require(graphics) # for mosaicplot
summary(esoph)
## effects of alcohol, tobacco and interaction, age-adjusted
```

```
model1 <- glm(cbind(ncases, ncontrols) ~ agegp + tobgp * alcgp,
              data = esoph, family = binomial())
anova(model1)
## Try a linear effect of alcohol and tobacco
model2 <- glm(cbind(ncases, ncontrols) ~ agegp + unclass(tobgp)
                                         + unclass(alcgp),
              data = esoph, family = binomial())
summary(model2)
## Re-arrange data for a mosaic plot
ttt <- table(esoph$agegp, esoph$alcgp, esoph$tobgp)
ttt[ttt == 1] <- esoph$ncases
tt1 <- table(esoph$agegp, esoph$alcgp, esoph$tobgp)
tt1[tt1 == 1] <- esoph$ncontrols
tt <- array(c(ttt, tt1), c(dim(ttt),2),
            c(dimnames(ttt), list(c("Cancer", "control"))))
mosaicplot(tt, main = "esoph data set", color = TRUE)
```

`euro` *Conversion Rates of Euro Currencies*

Description

Conversion rates between the various Euro currencies.

Usage

```
euro
euro.cross
```

Format

`euro` is a named vector of length 11, `euro.cross` a matrix of size 11 by 11, with dimnames.

Details

The data set `euro` contains the value of 1 Euro in all currencies participating in the European monetary union (Austrian Schilling ATS, Belgian Franc BEF, German Mark DEM, Spanish Peseta ESP, Finnish Markka FIM, French Franc FRF, Irish Punt IEP, Italian Lira ITL, Luxembourg Franc LUF, Dutch Guilder NLG and Portuguese Escudo PTE). These conversion rates were fixed by the European Union on December 31, 1998. To convert old prices to Euro prices, divide by the respective rate and round to 2 digits.

The data set `euro.cross` contains conversion rates between the various Euro currencies, i.e., the result of `outer(1 / euro, euro)`.

Examples

```
cbind(euro)

## These relations hold:
euro == signif(euro,6) # [6 digit precision in Euro's definition]
all(euro.cross == outer(1/euro, euro))

## Convert 20 Euro to Belgian Franc
```

```
20 * euro["BEF"]
## Convert 20 Austrian Schilling to Euro
20 / euro["ATS"]
## Convert 20 Spanish Pesetas to Italian Lira
20 * euro.cross["ESP", "ITL"]

require(graphics)
dotchart(euro,
         main = "euro data: 1 Euro in currency unit")
dotchart(1/euro,
         main = "euro data: 1 currency unit in Euros")
dotchart(log(euro, 10),
         main = "euro data: log10(1 Euro in currency unit)")
```

---

eurodist *Distances Between European Cities*

Description

The data give the road distances (in km) between 21 cities in Europe. The data are taken from a table in *The Cambridge Encyclopaedia*.

Usage

```
eurodist
```

Format

A `dist` object based on 21 objects. (You must have the **stats** package loaded to have the methods for this kind of object available).

Source

Crystal, D. Ed. (1990) *The Cambridge Encyclopaedia*. Cambridge: Cambridge University Press,

---

EuStockMarkets *Daily Closing Prices of Major European Stock Indices, 1991–1998*

Description

Contains the daily closing prices of major European stock indices: Germany DAX (Ibis), Switzerland SMI, France CAC, and UK FTSE. The data are sampled in business time, i.e., weekends and holidays are omitted.

Usage

```
EuStockMarkets
```

Format

A multivariate time series with 1860 observations on 4 variables. The object is of class `"mts"`.

Source

The data were kindly provided by Erste Bank AG, Vienna, Austria.

`faithful` *Old Faithful Geyser Data*

Description

Waiting time between eruptions and the duration of the eruption for the Old Faithful geyser in Yellowstone National Park, Wyoming, USA.

Usage

```
faithful
```

Format

A data frame with 272 observations on 2 variables.

[,1]	eruptions	numeric	Eruption time in mins
[,2]	waiting	numeric	Waiting time to next eruption (in mins)

Details

A closer look at `faithful$eruptions` reveals that these are heavily rounded times originally in seconds, where multiples of 5 are more frequent than expected under non-human measurement. For a better version of the eruption times, see the example below.

There are many versions of this dataset around: Azzalini and Bowman (1990) use a more complete version.

Source

W. Härdle.

References

Härdle, W. (1991) *Smoothing Techniques with Implementation in S*. New York: Springer.

Azzalini, A. and Bowman, A. W. (1990). A look at some data on the Old Faithful geyser. *Applied Statistics* **39**, 357–365.

See Also

`geyser` in package **MASS** for the Azzalini–Bowman version.

Examples

```
require(stats); require(graphics)
f.tit <-  "faithful data: Eruptions of Old Faithful"

ne60 <- round(e60 <- 60 * faithful$eruptions)
all.equal(e60, ne60)             # relative diff. ~ 1/10000
```

```
table(zapsmall(abs(e60 - ne60))) # 0, 0.02 or 0.04
faithful$better.eruptions <- ne60 / 60
te <- table(ne60)
te[te >= 4]                      # (too) many multiples of 5 !
plot(names(te), te, type="h", main = f.tit, xlab = "Eruption time (sec)")

plot(faithful[, -3], main = f.tit,
     xlab = "Eruption time (min)",
     ylab = "Waiting time to next eruption (min)")
lines(lowess(faithful$eruptions, faithful$waiting, f = 2/3, iter = 3),
      col = "red")
```

`Formaldehyde` *Determination of Formaldehyde*

Description

These data are from a chemical experiment to prepare a standard curve for the determination of formaldehyde by the addition of chromatropic acid and concentrated sulpuric acid and the reading of the resulting purple color on a spectrophotometer.

Usage

```
Formaldehyde
```

Format

A data frame with 6 observations on 2 variables.

[,1]	carb	numeric	Carbohydrate (ml)
[,2]	optden	numeric	Optical Density

Source

Bennett, N. A. and N. L. Franklin (1954) *Statistical Analysis in Chemistry and the Chemical Industry*. New York: Wiley.

References

McNeil, D. R. (1977) *Interactive Data Analysis.* New York: Wiley.

Examples

```
require(stats); require(graphics)
plot(optden ~ carb, data = Formaldehyde,
     xlab = "Carbohydrate (ml)", ylab = "Optical Density",
     main = "Formaldehyde data", col = 4, las = 1)
abline(fm1 <- lm(optden ~ carb, data = Formaldehyde))
summary(fm1)
opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(fm1)
par(opar)
```

`freeny` *Freeny's Revenue Data*

Description

Freeny's data on quarterly revenue and explanatory variables.

Usage

```
freeny
freeny.x
freeny.y
```

Format

There are three 'freeny' data sets.

`freeny.y` is a time series with 39 observations on quarterly revenue from (1962,2Q) to (1971,4Q).

`freeny.x` is a matrix of explanatory variables. The columns are `freeny.y` lagged 1 quarter, price index, income level, and market potential.

Finally, `freeny` is a data frame with variables `y`, `lag.quarterly.revenue`, `price.index`, `income.level`, and `market.potential` obtained from the above two data objects.

Source

A. E. Freeny (1977) *A Portable Linear Regression Package with Test Programs*. Bell Laboratories memorandum.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
require(stats); require(graphics)
summary(freeny)
pairs(freeny, main = "freeny data")
# gives warning: freeny$y has class "ts"

summary(fm1 <- lm(y ~ ., data = freeny))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)
```

HairEyeColor *Hair and Eye Color of Statistics Students*

Description

Distribution of hair and eye color and sex in 592 statistics students.

Usage

```
HairEyeColor
```

Format

A 3-dimensional array resulting from cross-tabulating 592 observations on 3 variables. The variables and their levels are as follows:

No	Name	Levels
1	Hair	Black, Brown, Red, Blond
2	Eye	Brown, Blue, Hazel, Green
3	Sex	Male, Female

Details

The Hair × Eye table comes rom a survey of students at the University of Delaware reported by Snee (1974). The split by Sex was added by Friendly (1992a) for didactic purposes.

This data set is useful for illustrating various techniques for the analysis of contingency tables, such as the standard chi-squared test or, more generally, log-linear modelling, and graphical methods such as mosaic plots, sieve diagrams or association plots.

Source

http://euclid.psych.yorku.ca/ftp/sas/vcd/catdata/haireye.sas

Snee (1974) gives the two-way table aggregated over Sex. The Sex split of the 'Brown hair, Brown eye' cell was changed in R 2.6.0 to agree with that used by Friendly (2000).

References

Snee, R. D. (1974) Graphical display of two-way contingency tables. *The American Statistician*, **28**, 9–12.

Friendly, M. (1992a) Graphical methods for categorical data. *SAS User Group International Conference Proceedings*, **17**, 190–200. http://www.math.yorku.ca/SCS/sugi/sugi17-paper.html

Friendly, M. (1992b) Mosaic displays for loglinear models. *Proceedings of the Statistical Graphics Section*, American Statistical Association, pp. 61–68. http://www.math.yorku.ca/SCS/Papers/asa92.html

Friendly, M. (2000) *Visualizing Categorical Data.* SAS Institute, ISBN 1-58025-660-0.

See Also

chisq.test, loglin, mosaicplot

Examples

```
require(graphics)
## Full mosaic
mosaicplot(HairEyeColor)
## Aggregate over sex (as in Snee's original data)
x <- apply(HairEyeColor, c(1, 2), sum)
x
mosaicplot(x, main = "Relation between hair and eye color")
```

Harman23.cor *Harman Example 2.3*

Description

A correlation matrix of eight physical measurements on 305 girls between ages seven and seventeen.

Usage

```
Harman23.cor
```

Source

Harman, H. H. (1976) *Modern Factor Analysis*, Third Edition Revised, University of Chicago Press, Table 2.3.

Examples

```
require(stats)
(Harman23.FA <- factanal(factors = 1, covmat = Harman23.cor))
for(factors in 2:4) print(update(Harman23.FA, factors = factors))
```

Harman74.cor *Harman Example 7.4*

Description

A correlation matrix of 24 psychological tests given to 145 seventh and eight-grade children in a Chicago suburb by Holzinger and Swineford.

Usage

```
Harman74.cor
```

Source

Harman, H. H. (1976) *Modern Factor Analysis*, Third Edition Revised, University of Chicago Press, Table 7.4.

Examples

```
require(stats)
(Harman74.FA <- factanal(factors = 1, covmat = Harman74.cor))
for(factors in 2:5) print(update(Harman74.FA, factors = factors))
Harman74.FA <- factanal(factors = 5, covmat = Harman74.cor,
                        rotation="promax")
print(Harman74.FA$loadings, sort = TRUE)
```

Indometh *Pharmacokinetics of Indomethicin*

Description

The `Indometh` data frame has 66 rows and 3 columns of data on the pharmacokinetics of indomethicin.

Usage

```
Indometh
```

Format

This data frame contains the following columns:

Subject an ordered factor with containing the subject codes. The ordering is according to increasing maximum response.

time a numeric vector of times at which blood samples were drawn (hr).

conc a numeric vector of plasma concentrations of indomethicin (mcg/ml).

Details

Each of the six subjects were given an intravenous injection of indomethicin.

Source

Kwan, Breault, Umbenhauer, McMahon and Duggan (1976), Kinetics of Indomethicin absorption, elimination, and enterohepatic circulation in man. *Journal of Pharmacokinetics and Biopharmaceutics*, **4**, 255–280.

Davidian, M. and Giltinan, D. M. (1995) *Nonlinear Models for Repeated Measurement Data*, Chapman & Hall (section 5.2.4, p. 134)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats)
fm1 <- nls(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2),
           data = Indometh, subset = Subject == 1)
summary(fm1)
```

`infert` *Infertility after Spontaneous and Induced Abortion*

Description

This is a matched case-control study dating from before the availability of conditional logistic regression.

Usage

```
infert
```

Format

1.	Education	0 = 0-5 years
		1 = 6-11 years
		2 = 12+ years
2.	age	age in years of case
3.	parity	count
4.	number of prior	0 = 0
	induced abortions	1 = 1
		2 = 2 or more
5.	case status	1 = case
		0 = control
6.	number of prior	0 = 0
	spontaneous abortions	1 = 1
		2 = 2 or more
7.	matched set number	1-83
8.	stratum number	1-63

Note

One case with two prior spontaneous abortions and two prior induced abortions is omitted.

Source

Trichopoulos et al. (1976) *Br. J. of Obst. and Gynaec.* **83**, 645–650.

Examples

```
require(stats)
model1 <- glm(case ~ spontaneous+induced, data=infert,family=binomial())
summary(model1)
## adjusted for other potential confounders:
summary(model2 <- glm(case ~ age+parity+education+spontaneous+induced,
                data=infert,family=binomial()))
## Really should be analysed by conditional logistic regression
## which is in the survival package
if(require(survival)){
```

```
  model3 <- clogit(case~spontaneous+induced+strata(stratum),data=infert)
  print(summary(model3))
  detach()# survival (conflicts)
}
```

`InsectSprays` *Effectiveness of Insect Sprays*

Description

The counts of insects in agricultural experimental units treated with different insecticides.

Usage

```
InsectSprays
```

Format

A data frame with 72 observations on 2 variables.

[,1]	count	numeric	Insect count
[,2]	spray	factor	The type of spray

Source

Beall, G., (1942) The Transformation of data from entomological field experiments, *Biometrika*, **29**, 243–262.

References

McNeil, D. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(stats); require(graphics)
boxplot(count ~ spray, data = InsectSprays,
        xlab = "Type of spray", ylab = "Insect count",
        main = "InsectSprays data", varwidth = TRUE, col = "lightgray")
fm1 <- aov(count ~ spray, data = InsectSprays)
summary(fm1)
opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(fm1)
fm2 <- aov(sqrt(count) ~ spray, data = InsectSprays)
summary(fm2)
plot(fm2)
par(opar)
```

`iris` *Edgar Anderson's Iris Data*

Description

This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are *Iris setosa*, *versicolor*, and *virginica*.

Usage

```
iris
iris3
```

Format

`iris` is a data frame with 150 cases (rows) and 5 variables (columns) named `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`, and `Species`.

`iris3` gives the same data arranged as a 3-dimensional array of size 50 by 4 by 3, as represented by S-PLUS. The first dimension gives the case number within the species subsample, the second the measurements with names `Sepal L.`, `Sepal W.`, `Petal L.`, and `Petal W.`, and the third the species.

Source

Fisher, R. A. (1936) The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, **7**, Part II, 179–188.

The data were collected by Anderson, Edgar (1935). The irises of the Gaspe Peninsula, *Bulletin of the American Iris Society*, **59**, 2–5.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (has `iris3` as `iris`.)

See Also

`matplot` some examples of which use `iris`.

Examples

```
dni3 <- dimnames(iris3)
ii <- data.frame(matrix(aperm(iris3, c(1,3,2)), ncol=4,
                        dimnames = list(NULL, sub(" L.",".Length",
                                                  sub(" W.",".Width", dni3[[2]])))),
    Species = gl(3, 50, labels=sub("S", "s", sub("V", "v", dni3[[3]]))))
all.equal(ii, iris) # TRUE
```

islands *Areas of the World's Major Landmasses*

Description

The areas in thousands of square miles of the landmasses which exceed 10,000 square miles.

Usage

```
islands
```

Format

A named vector of length 48.

Source

The World Almanac and Book of Facts, 1975, page 406.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(graphics)
dotchart(log(islands, 10),
   main = "islands data: log10(area) (log10(sq. miles))")
dotchart(log(islands[order(islands)], 10),
   main = "islands data: log10(area) (log10(sq. miles))")
```

JohnsonJohnson *Quarterly Earnings per Johnson & Johnson Share*

Description

Quarterly earnings (dollars) per Johnson & Johnson share 1960–80.

Usage

```
JohnsonJohnson
```

Format

A quarterly time series

Source

Shumway, R. H. and Stoffer, D. S. (2000) *Time Series Analysis and its Applications*. Second Edition. Springer. Example 1.1.

Examples

```
require(stats); require(graphics)
JJ <- log10(JohnsonJohnson)
plot(JJ)
(fit <- StructTS(JJ, type="BSM"))
tsdiag(fit)
sm <- tsSmooth(fit)
plot(cbind(JJ, sm[, 1], sm[, 3]-0.5), plot.type = "single",
     col = c("black", "green", "blue"))
abline(h = -0.5, col = "grey60")

monthplot(fit)
```

LakeHuron *Level of Lake Huron 1875–1972*

Description

Annual measurements of the level, in feet, of Lake Huron 1875–1972.

Usage

```
LakeHuron
```

Format

A time series of length 98.

Source

Brockwell, P. J. & Davis, R. A. (1991). *Time Series and Forecasting Methods*. Second edition. Springer, New York. Series A, page 555.

Brockwell, P. J. & Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. Springer, New York. Sections 5.1 and 7.6.

---

lh *Luteinizing Hormone in Blood Samples*

Description

A regular time series giving the luteinizing hormone in blood samples at 10 mins intervals from a human female, 48 samples.

Usage

```
lh
```

Source

P.J. Diggle (1990) *Time Series: A Biostatistical Introduction.* Oxford, table A.1, series 3

`LifeCycleSavings` *Intercountry Life-Cycle Savings Data*

Description

Data on the savings ratio 1960–1970.

Usage

```
LifeCycleSavings
```

Format

A data frame with 50 observations on 5 variables.

[,1]	sr	numeric	aggregate personal savings
[,2]	pop15	numeric	% of population under 15
[,3]	pop75	numeric	% of population over 75
[,4]	dpi	numeric	real per-capita disposable income
[,5]	ddpi	numeric	% growth rate of dpi

Details

Under the life-cycle savings hypothesis as developed by Franco Modigliani, the savings ratio (aggregate personal saving divided by disposable income) is explained by per-capita disposable income, the percentage rate of change in per-capita disposable income, and two demographic variables: the percentage of population less than 15 years old and the percentage of the population over 75 years old. The data are averaged over the decade 1960–1970 to remove the business cycle or other short-term fluctuations.

Source

The data were obtained from Belsley, Kuh and Welsch (1980). They in turn obtained the data from Sterling (1977).

References

Sterling, Arnie (1977) Unpublished BS Thesis. Massachusetts Institute of Technology.

Belsley, D. A., Kuh. E. and Welsch, R. E. (1980) *Regression Diagnostics*. New York: Wiley.

Examples

```
require(stats); require(graphics)
pairs(LifeCycleSavings, panel = panel.smooth,
      main = "LifeCycleSavings data")
fm1 <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)
summary(fm1)
```

Loblolly *Growth of Loblolly pine trees*

Description

The `Loblolly` data frame has 84 rows and 3 columns of records of the growth of Loblolly pine trees.

Usage

```
Loblolly
```

Format

This data frame contains the following columns:

height a numeric vector of tree heights (ft).

age a numeric vector of tree ages (yr).

Seed an ordered factor indicating the seed source for the tree. The ordering is according to increasing maximum height.

Source

Kung, F. H. (1986), Fitting logistic growth curve with predetermined carrying capacity, in *Proceedings of the Statistical Computing Section, American Statistical Association*, 340–343.

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats); require(graphics)
plot(height ~ age, data = Loblolly, subset = Seed == 329,
     xlab = "Tree age (yr)", las = 1,
     ylab = "Tree height (ft)",
     main = "Loblolly data and fitted curve (Seed 329 only)")
fm1 <- nls(height ~ SSasymp(age, Asym, R0, lrc),
           data = Loblolly, subset = Seed == 329)
summary(fm1)
age <- seq(0, 30, length.out = 101)
lines(age, predict(fm1, list(age = age)))
```

longley *Longley's Economic Regression Data*

Description

A macroeconomic data set which provides a well-known example for a highly collinear regression.

Usage

```
longley
```

Format

A data frame with 7 economical variables, observed yearly from 1947 to 1962 ($n = 16$).

GNP.deflator: GNP implicit price deflator ($1954 = 100$)

GNP: Gross National Product.

Unemployed: number of unemployed.

Armed.Forces: number of people in the armed forces.

Population: 'noninstitutionalized' population $\geq$ 14 years of age.

Year: the year (time).

Employed: number of people employed.

The regression `lm(Employed ~ .)` is known to be highly collinear.

Source

J. W. Longley (1967) An appraisal of least-squares programs from the point of view of the user. *Journal of the American Statistical Association*, **62**, 819–841.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
require(stats); require(graphics)
## give the data set in the form it is used in S-PLUS:
longley.x <- data.matrix(longley[, 1:6])
longley.y <- longley[, "Employed"]
pairs(longley, main = "longley data")
summary(fm1 <- lm(Employed ~ ., data = longley))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)
```

`lynx` *Annual Canadian Lynx trappings 1821–1934*

Description

Annual numbers of lynx trappings for 1821–1934 in Canada. Taken from Brockwell & Davis (1991), this appears to be the series considered by Campbell & Walker (1977).

Usage

```
lynx
```

Source

Brockwell, P. J. and Davis, R. A. (1991) *Time Series and Forecasting Methods.* Second edition. Springer. Series G (page 557).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Campbell, M. J.and A. M. Walker (1977). A Survey of statistical work on the Mackenzie River series of annual Canadian lynx trappings for the years 1821–1934 and a new analysis. *Journal of the Royal Statistical Society series A*, **140**, 411–431.

`morley` *Michaelson-Morley Speed of Light Data*

Description

The classical data of Michaelson and Morley on the speed of light. The data consists of five experiments, each consisting of 20 consecutive 'runs'. The response is the speed of light measurement, suitably coded.

Usage

```
morley
```

Format

A data frame contains the following components:

`Expt` The experiment number, from 1 to 5.

`Run` The run number within each experiment.

`Speed` Speed-of-light measurement.

Details

The data is here viewed as a randomized block experiment with 'experiment' and 'run' as the factors. 'run' may also be considered a quantitative variate to account for linear (or polynomial) changes in the measurement over the course of a single experiment.

Source

A. J. Weekes (1986) *A Genstat Primer*. London: Edward Arnold.

Examples

```
require(stats); require(graphics)
morley$Expt <- factor(morley$Expt)
morley$Run <- factor(morley$Run)

xtabs(~ Expt + Run, data = morley)# 5 x 20 balanced (two-way)
plot(Speed ~ Expt, data = morley,
     main = "Speed of Light Data", xlab = "Experiment No.")
fm <- aov(Speed ~ Run + Expt, data = morley)
summary(fm)
fm0 <- update(fm, . ~ . - Run)
anova(fm0, fm)
```

mtcars *Motor Trend Car Road Tests*

Description

The data was extracted from the 1974 *Motor Trend* US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

Usage

```
mtcars
```

Format

A data frame with 32 observations on 11 variables.

[, 1]	mpg	Miles/(US) gallon
[, 2]	cyl	Number of cylinders
[, 3]	disp	Displacement (cu.in.)
[, 4]	hp	Gross horsepower
[, 5]	drat	Rear axle ratio
[, 6]	wt	Weight (lb/1000)
[, 7]	qsec	1/4 mile time
[, 8]	vs	V/S
[, 9]	am	Transmission (0 = automatic, 1 = manual)
[,10]	gear	Number of forward gears
[,11]	carb	Number of carburetors

Source

Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics*, **37**, 391–411.

Examples

```
require(graphics)
pairs(mtcars, main = "mtcars data")
coplot(mpg ~ disp | as.factor(cyl), data = mtcars,
       panel = panel.smooth, rows = 1)
```

nhtemp *Average Yearly Temperatures in New Haven*

Description

The mean annual temperature in degrees Fahrenheit in New Haven, Connecticut, from 1912 to 1971.

Usage

```
nhtemp
```

Format

A time series of 60 observations.

Source

Vaux, J. E. and Brinker, N. B. (1972) *Cycles*, **1972**, 117–121.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(stats); require(graphics)
plot(nhtemp, main = "nhtemp data",
  ylab = "Mean annual temperature in New Haven, CT (deg. F)")
```

Nile *Flow of the River Nile*

Description

Measurements of the annual flow of the river Nile at Ashwan 1871–1970.

Usage

```
Nile
```

Format

A time series of length 100.

Source

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press. http://www.ssfpack.com/dkbook/

References

Balke, N. S. (1993) Detecting level shifts in time series. *Journal of Business and Economic Statistics* **11**, 81–92.

Cobb, G. W. (1978) The problem of the Nile: conditional solution to a change-point problem. *Biometrika* **65**, 243–51.

Examples

```
require(stats); require(graphics)
par(mfrow = c(2,2))
plot(Nile)
acf(Nile)
pacf(Nile)
ar(Nile) # selects order 2
cpgram(ar(Nile)$resid)
par(mfrow = c(1,1))
arima(Nile, c(2, 0, 0))

## Now consider missing values, following Durbin & Koopman
NileNA <- Nile
NileNA[c(21:40, 61:80)] <- NA
arima(NileNA, c(2, 0, 0))
plot(NileNA)
pred <-
   predict(arima(window(NileNA, 1871, 1890), c(2,0,0)), n.ahead = 20)
lines(pred$pred, lty = 3, col = "red")
lines(pred$pred + 2*pred$se, lty=2, col="blue")
lines(pred$pred - 2*pred$se, lty=2, col="blue")
pred <-
   predict(arima(window(NileNA, 1871, 1930), c(2,0,0)), n.ahead = 20)
lines(pred$pred, lty = 3, col = "red")
lines(pred$pred + 2*pred$se, lty=2, col="blue")
lines(pred$pred - 2*pred$se, lty=2, col="blue")

## Structural time series models
par(mfrow = c(3, 1))
plot(Nile)
## local level model
(fit <- StructTS(Nile, type = "level"))
lines(fitted(fit), lty = 2)              # contemporaneous smoothing
lines(tsSmooth(fit), lty = 2, col = 4)   # fixed-interval smoothing
plot(residuals(fit)); abline(h = 0, lty = 3)
## local trend model
(fit2 <- StructTS(Nile, type = "trend")) ## constant trend fitted
pred <- predict(fit, n.ahead = 30)
## with 50% confidence interval
ts.plot(Nile, pred$pred,
        pred$pred + 0.67*pred$se, pred$pred -0.67*pred$se)

## Now consider missing values
plot(NileNA)
(fit3 <- StructTS(NileNA, type = "level"))
lines(fitted(fit3), lty = 2)
lines(tsSmooth(fit3), lty = 3)
plot(residuals(fit3)); abline(h = 0, lty = 3)
```

`nottem` *Average Monthly Temperatures at Nottingham, 1920–1939*

Description

A time series object containing average air temperatures at Nottingham Castle in degrees Fahrenheit for 20 years.

Usage

```
nottem
```

Source

Anderson, O. D. (1976) *Time Series Analysis and Forecasting: The Box-Jenkins approach.* Butterworths. Series R.

Examples

```
## Not run:
require(stats); require(graphics)
nott <- window(nottem, end=c(1936,12))
fit <- arima(nott,order=c(1,0,0), list(order=c(2,1,0), period=12))
nott.fore <- predict(fit, n.ahead=36)
ts.plot(nott, nott.fore$pred, nott.fore$pred+2*nott.fore$se,
        nott.fore$pred-2*nott.fore$se, gpars=list(col=c(1,1,4,4)))
## End(Not run)
```

---

occupationalStatus *Occupational Status of Fathers and their Sons*

Description

Cross-classification of a sample of British males according to each subject's occupational status and his father's occupational status.

Usage

```
occupationalStatus
```

Format

A `table` of counts, with classifying factors `origin` (father's occupational status; levels `1:8`) and `destination` (son's occupational status; levels `1:8`).

Source

Goodman, L. A. (1979) Simple Models for the Analysis of Association in Cross-Classifications having Ordered Categories. *J. Am. Stat. Assoc.*, **74** (367), 537–552.

The data set has been in package **gnm** and been provided by the package authors.

Examples

```
require(stats); require(graphics)

plot(occupationalStatus)

##  Fit a uniform association model separating diagonal effects
Diag <- as.factor(diag(1:8))
Rscore <- scale(as.numeric(row(occupationalStatus)), scale = FALSE)
Cscore <- scale(as.numeric(col(occupationalStatus)), scale = FALSE)
modUnif <- glm(Freq ~ origin + destination + Diag + Rscore:Cscore,
               family = poisson, data = occupationalStatus)

summary(modUnif)
plot(modUnif) # 4 plots, with warning about  h_ii ~= 1
```

`Orange` *Growth of Orange Trees*

Description

The `Orange` data frame has 35 rows and 3 columns of records of the growth of orange trees.

Usage

```
Orange
```

Format

This data frame contains the following columns:

Tree an ordered factor indicating the tree on which the measurement is made. The ordering is according to increasing maximum diameter.

age a numeric vector giving the age of the tree (days since 1968/12/31)

circumference a numeric vector of trunk circumferences (mm). This is probably "circumference at breast height", a standard measurement in forestry.

Source

Draper, N. R. and Smith, H. (1998), *Applied Regression Analysis (3rd ed)*, Wiley (exercise 24.N).

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

Examples

```
require(stats); require(graphics)
coplot(circumference ~ age | Tree, data = Orange, show.given = FALSE)
fm1 <- nls(circumference ~ SSlogis(age, Asym, xmid, scal),
           data = Orange, subset = Tree == 3)
plot(circumference ~ age, data = Orange, subset = Tree == 3,
     xlab = "Tree age (days since 1968/12/31)",
     ylab = "Tree circumference (mm)", las = 1,
     main = "Orange tree data and fitted model (Tree 3 only)")
age <- seq(0, 1600, length.out = 101)
lines(age, predict(fm1, list(age = age)))
```

`OrchardSprays` *Potency of Orchard Sprays*

Description

An experiment was conducted to assess the potency of various constituents of orchard sprays in repelling honeybees, using a Latin square design.

Usage

```
OrchardSprays
```

Format

A data frame with 64 observations on 4 variables.

[,1]	rowpos	numeric	Row of the design
[,2]	colpos	numeric	Column of the design
[,3]	treatment	factor	Treatment level
[,4]	decrease	numeric	Response

Details

Individual cells of dry comb were filled with measured amounts of lime sulphur emulsion in sucrose solution. Seven different concentrations of lime sulphur ranging from a concentration of 1/100 to 1/1,562,500 in successive factors of 1/5 were used as well as a solution containing no lime sulphur.

The responses for the different solutions were obtained by releasing 100 bees into the chamber for two hours, and then measuring the decrease in volume of the solutions in the various cells.

An 8×8 Latin square design was used and the treatments were coded as follows:

A	highest level of lime sulphur
B	next highest level of lime sulphur
.	
.	
.	
G	lowest level of lime sulphur
H	no lime sulphur

Source

Finney, D. J. (1947) *Probit Analysis*. Cambridge.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(graphics)
pairs(OrchardSprays, main = "OrchardSprays data")
```

PlantGrowth *Results from an Experiment on Plant Growth*

Description

Results from an experiment to compare yields (as measured by dried weight of plants) obtained under a control and two different treatment conditions.

Usage

```
PlantGrowth
```

Format

A data frame of 30 cases on 2 variables.

[, 1]	weight	numeric
[, 2]	group	factor

The levels of `group` are 'ctrl', 'trt1', and 'trt2'.

Source

Dobson, A. J. (1983) *An Introduction to Statistical Modelling*. London: Chapman and Hall.

Examples

```
## One factor ANOVA example from Dobson's book, cf. Table 7.4:
require(stats); require(graphics)
boxplot(weight ~ group, data = PlantGrowth, main = "PlantGrowth data",
        ylab = "Dried weight of plants", col = "lightgray",
        notch = TRUE, varwidth = TRUE)
anova(lm(weight ~ group, data = PlantGrowth))
```

precip *Annual Precipitation in US Cities*

Description

The average amount of precipitation (rainfall) in inches for each of 70 United States (and Puerto Rico) cities.

Usage

```
precip
```

Format

A named vector of length 70.

Source

Statistical Abstracts of the United States, 1975.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(graphics)
dotchart(precip[order(precip)], main = "precip data")
title(sub = "Average annual precipitation (in.)")
```

---

presidents *Quarterly Approval Ratings of US Presidents*

Description

The (approximately) quarterly approval rating for the President of the United states from the first quarter of 1945 to the last quarter of 1974.

Usage

```
presidents
```

Format

A time series of 120 values.

Details

The data are actually a fudged version of the approval ratings. See McNeil's book for details.

Source

The Gallup Organisation.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(stats); require(graphics)
plot(presidents, las = 1, ylab = "Approval rating (%)",
     main = "presidents data")
```

`pressure` *Vapor Pressure of Mercury as a Function of Temperature*

Description

Data on the relation between temperature in degrees Celsius and vapor pressure of mercury in millimeters (of mercury).

Usage

```
pressure
```

Format

A data frame with 19 observations on 2 variables.

[, 1]	temperature	numeric	temperature (deg C)
[, 2]	pressure	numeric	pressure (mm)

Source

Weast, R. C., ed. (1973) *Handbook of Chemistry and Physics*. CRC Press.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(graphics)
plot(pressure, xlab = "Temperature (deg C)",
     ylab = "Pressure (mm of Hg)",
     main = "pressure data: Vapor Pressure of Mercury")
plot(pressure, xlab = "Temperature (deg C)",  log = "y",
     ylab = "Pressure (mm of Hg)",
     main = "pressure data: Vapor Pressure of Mercury")
```

`Puromycin` *Reaction velocity of an enzymatic reaction*

Description

The `Puromycin` data frame has 23 rows and 3 columns of the reaction velocity versus substrate concentration in an enzymatic reaction involving untreated cells or cells treated with Puromycin.

Usage

```
Puromycin
```

Format

This data frame contains the following columns:

conc a numeric vector of substrate concentrations (ppm)

rate a numeric vector of instantaneous reaction rates (counts/min/min)

state a factor with levels `treated` `untreated`

Details

Data on the velocity of an enzymatic reaction were obtained by Treloar (1974). The number of counts per minute of radioactive product from the reaction was measured as a function of substrate concentration in parts per million (ppm) and from these counts the initial rate (or velocity) of the reaction was calculated (counts/min/min). The experiment was conducted once with the enzyme treated with Puromycin, and once with the enzyme untreated.

Source

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley, Appendix A1.3.

Treloar, M. A. (1974), *Effects of Puromycin on Galactosyltransferase in Golgi Membranes*, M.Sc. Thesis, U. of Toronto.

Examples

```
require(stats); require(graphics)
plot(rate ~ conc, data = Puromycin, las = 1,
     xlab = "Substrate concentration (ppm)",
     ylab = "Reaction velocity (counts/min/min)",
     pch = as.integer(Puromycin$state),
     col = as.integer(Puromycin$state),
     main = "Puromycin data and fitted Michaelis-Menten curves")
## simplest form of fitting the Michaelis-Menten model to these data
fm1 <- nls(rate ~ Vm * conc/(K + conc), data = Puromycin,
           subset = state == "treated",
           start = c(Vm = 200, K = 0.05), trace = TRUE)
fm2 <- nls(rate ~ Vm * conc/(K + conc), data = Puromycin,
           subset = state == "untreated",
           start = c(Vm = 160, K = 0.05), trace = TRUE)
summary(fm1)
summary(fm2)
## using partial linearity
fm3 <- nls(rate ~ conc/(K + conc), data = Puromycin,
           subset = state == "treated", start = c(K = 0.05),
           algorithm = "plinear", trace = TRUE)
## using a self-starting model
fm4 <- nls(rate ~ SSmicmen(conc, Vm, K), data = Puromycin,
           subset = state == "treated")
summary(fm4)
## add fitted lines to the plot
conc <- seq(0, 1.2, length.out = 101)
lines(conc, predict(fm1, list(conc = conc)), lty = 1, col = 1)
lines(conc, predict(fm2, list(conc = conc)), lty = 2, col = 2)
legend(0.8, 120, levels(Puromycin$state),
       col = 1:2, lty = 1:2, pch = 1:2)
```

quakes *Locations of Earthquakes off Fiji*

Description

The data set give the locations of 1000 seismic events of MB > 4.0. The events occurred in a cube near Fiji since 1964.

Usage

```
quakes
```

Format

A data frame with 1000 observations on 5 variables.

[,1]	lat	numeric	Latitude of event
[,2]	long	numeric	Longitude
[,3]	depth	numeric	Depth (km)
[,4]	mag	numeric	Richter Magnitude
[,5]	stations	numeric	Number of stations reporting

Details

There are two clear planes of seismic activity. One is a major plate junction; the other is the Tonga trench off New Zealand. These data constitute a subsample from a larger dataset of containing 5000 observations.

Source

This is one of the Harvard PRIM-H project data sets. They in turn obtained it from Dr. John Woodhouse, Dept. of Geophysics, Harvard University.

Examples

```
require(graphics)
pairs(quakes, main = "Fiji Earthquakes, N = 1000", cex.main=1.2, pch=".")
```

randu *Random Numbers from Congruential Generator RANDU*

Description

400 triples of successive random numbers were taken from the VAX FORTRAN function RANDU running under VMS 1.5.

Usage

```
randu
```

Format

A data frame with 400 observations on 3 variables named `x`, `y` and `z` which give the first, second and third random number in the triple.

Details

In three dimensional displays it is evident that the triples fall on 15 parallel planes in 3-space. This can be shown theoretically to be true for all triples from the RANDU generator.

These particular 400 triples start 5 apart in the sequence, that is they are ((U[5i+1], U[5i+2], U[5i+3]), i= 0, . . . , 399), and they are rounded to 6 decimal places.

Under VMS versions 2.0 and higher, this problem has been fixed.

Source

David Donoho

Examples

```
## Not run:
## We could re-generate the dataset by the following R code
seed <- as.double(1)
RANDU <- function() {
    seed <<- ((2^16 + 3) * seed) %% (2^31)
    seed/(2^31)
}
for(i in 1:400) {
    U <- c(RANDU(), RANDU(), RANDU(), RANDU(), RANDU())
    print(round(U[1:3], 6))
}
## End(Not run)
```

`rivers` *Lengths of Major North American Rivers*

Description

This data set gives the lengths (in miles) of 141 “major” rivers in North America, as compiled by the US Geological Survey.

Usage

```
rivers
```

Format

A vector containing 141 observations.

Source

World Almanac and Book of Facts, 1975, page 406.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

`rock` *Measurements on Petroleum Rock Samples*

Description

Measurements on 48 rock samples from a petroleum reservoir.

Usage

```
rock
```

Format

A data frame with 48 rows and 4 numeric columns.

[,1]	area	area of pores space, in pixels out of 256 by 256
[,2]	peri	perimeter in pixels
[,3]	shape	perimeter/sqrt(area)
[,4]	perm	permeability in milli-Darcies

Details

Twelve core samples from petroleum reservoirs were sampled by 4 cross-sections. Each core sample was measured for permeability, and each cross-section has total area of pores, total perimeter of pores, and shape.

Source

Data from BP Research, image analysis by Ronit Katz, U. Oxford.

`sleep` *Student's Sleep Data*

Description

Data which show the effect of two soporific drugs (increase in hours of sleep compared to control) on 10 patients.

Usage

```
sleep
```

Format

A data frame with 20 observations on 2 variables.

[, 1]	extra	numeric	increase in hours of sleep
[, 2]	group	factor	drug given

Source

Cushny, A. R. and Peebles, A. R. (1905) The action of optical isomers: II hyoscines. *The Journal of Physiology* **32**, 501–510.

Student (1908) The probable error of the mean. *Biometrika*, **6**, 20.

References

Scheffé, Henry (1959) *The Analysis of Variance*. New York, NY: Wiley.

Examples

```
require(stats)
## Student's paired t-test
t.test(extra ~ group, data = sleep, paired = TRUE)
```

---

stackloss *Brownlee's Stack Loss Plant Data*

Description

Operational data of a plant for the oxidation of ammonia to nitric acid.

Usage

```
stackloss

stack.x
stack.loss
```

Format

`stackloss` is a data frame with 21 observations on 4 variables.

[,1]	`Air Flow`	Flow of cooling air
[,2]	`Water Temp`	Cooling Water Inlet Temperature
[,3]	`Acid Conc.`	Concentration of acid [per 1000, minus 500]
[,4]	`stack.loss`	Stack loss

For compatibility with S-PLUS, the data sets `stack.x`, a matrix with the first three (independent) variables of the data frame, and `stack.loss`, the numeric vector giving the fourth (dependent) variable, are provided as well.

Details

"Obtained from 21 days of operation of a plant for the oxidation of ammonia (NH_3) to nitric acid (HNO_3). The nitric oxides produced are absorbed in a countercurrent absorption tower". (Brownlee, cited by Dodge, slightly reformatted by MM.)

`Air Flow` represents the rate of operation of the plant. `Water Temp` is the temperature of cooling water circulated through coils in the absorption tower. `Acid Conc.` is the concentration of the acid circulating, minus 50, times 10: that is, 89 corresponds to 58.9 per cent acid. `stack.loss` (the dependent variable) is 10 times the percentage of the ingoing ammonia to the plant that escapes

from the absorption column unabsorbed; that is, an (inverse) measure of the over-all efficiency of the plant.

Source

Brownlee, K. A. (1960, 2nd ed. 1965) *Statistical Theory and Methodology in Science and Engineering*. New York: Wiley. pp. 491–500.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dodge, Y. (1996) The guinea pig of multiple regression. In: *Robust Statistics, Data Analysis, and Computer Intensive Methods; In Honor of Peter Huber's 60th Birthday*, 1996, *Lecture Notes in Statistics* **109**, Springer-Verlag, New York.

Examples

```
require(stats)
summary(lm.stack <- lm(stack.loss ~ stack.x))
```

---

state *US State Facts and Figures*

Description

Data sets related to the 50 states of the United States of America.

Usage

```
state.abb
state.area
state.center
state.division
state.name
state.region
state.x77
```

Details

R currently contains the following "state" data sets. Note that all data are arranged according to alphabetical order of the state names.

`state.abb`: character vector of 2-letter abbreviations for the state names.

`state.area`: numeric vector of state areas (in square miles).

`state.center`: list with components named `x` and `y` giving the approximate geographic center of each state in negative longitude and latitude. Alaska and Hawaii are placed just off the West Coast.

`state.division`: factor giving state divisions (New England, Middle Atlantic, South Atlantic, East South Central, West South Central, East North Central, West North Central, Mountain, and Pacific).

state.name: character vector giving the full state names.

state.region: factor giving the region (Northeast, South, North Central, West) that each state belongs to.

state.x77: matrix with 50 rows and 8 columns giving the following statistics in the respective columns.

Population: population estimate as of July 1, 1975

Income: per capita income (1974)

Illiteracy: illiteracy (1970, percent of population)

Life Exp: life expectancy in years (1969–71)

Murder: murder and non-negligent manslaughter rate per 100,000 population (1976)

HS Grad: percent high-school graduates (1970)

Frost: mean number of days with minimum temperature below freezing (1931–1960) in capital or large city

Area: land area in square miles

Source

U.S. Department of Commerce, Bureau of the Census (1977) *Statistical Abstract of the United States*.

U.S. Department of Commerce, Bureau of the Census (1977) *County and City Data Book*.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

---

sunspot.month *Monthly Sunspot Data, 1749–1997*

Description

Monthly numbers of sunspots.

Usage

```
sunspot.month
```

Format

The univariate time series `sunspot.year` and `sunspot.month` contain 289 and 2988 observations, respectively. The objects are of class `"ts"`.

Source

World Data Center-C1 For Sunspot Index Royal Observatory of Belgium, Av. Circulaire, 3, B-1180 BRUSSELS http://www.oma.be/KSB-ORB/SIDC/sidc_txt.html

See Also

`sunspot.month` is a longer version of `sunspots` that runs until 1988 rather than 1983.

Examples

```
require(stats); require(graphics)
## Compare the monthly series
plot (sunspot.month, main = "sunspot.month [stats]", col = 2)
lines(sunspots) # "very barely" see something

## Now look at the difference :
all(tsp(sunspots)     [c(1,3)] ==
    tsp(sunspot.month)[c(1,3)]) ## Start & Periodicity are the same
n1 <- length(sunspots)
table(eq <- sunspots == sunspot.month[1:n1]) #>  132  are different !
i <- which(!eq)
rug(time(eq)[i])
s1 <- sunspots[i] ; s2 <- sunspot.month[i]
cbind(i = i, sunspots = s1, ss.month = s2,
      perc.diff = round(100*2*abs(s1-s2)/(s1+s2), 1))
```

---

sunspot.year *Yearly Sunspot Data, 1700–1988*

Description

Yearly numbers of sunspots.

Usage

```
sunspot.year
```

Format

The univariate time series `sunspot.year` contains 289 observations, and is of class `"ts"`.

Source

H. Tong (1996) *Non-Linear Time Series*. Clarendon Press, Oxford, p. 471.

---

sunspots *Monthly Sunspot Numbers, 1749–1983*

Description

Monthly mean relative sunspot numbers from 1749 to 1983. Collected at Swiss Federal Observatory, Zurich until 1960, then Tokyo Astronomical Observatory.

Usage

```
sunspots
```

Format

A time series of monthly data from 1749 to 1983.

Source

Andrews, D. F. and Herzberg, A. M. (1985) *Data: A Collection of Problems from Many Fields for the Student and Research Worker*. New York: Springer-Verlag.

See Also

`sunspot.month` has a longer (and a bit different) series.

Examples

```
require(graphics)
plot(sunspots, main = "sunspots data", xlab = "Year",
     ylab = "Monthly sunspot numbers")
```

---

swiss — *Swiss Fertility and Socioeconomic Indicators (1888) Data*

Description

Standardized fertility measure and socio-economic indicators for each of 47 French-speaking provinces of Switzerland at about 1888.

Usage

```
swiss
```

Format

A data frame with 47 observations on 6 variables, *each* of which is in percent, i.e., in $[0, 100]$.

[,1]	Fertility	I_g, 'common standardized fertility measure'
[,2]	Agriculture	% of males involved in agriculture as occupation
[,3]	Examination	% draftees receiving highest mark on army examination
[,4]	Education	% education beyond primary school for draftees.
[,5]	Catholic	% 'catholic' (as opposed to 'protestant').
[,6]	Infant.Mortality	live births who live less than 1 year.

All variables but 'Fertility' give proportions of the population.

Details

(paraphrasing Mosteller and Tukey):

Switzerland, in 1888, was entering a period known as the *demographic transition*; i.e., its fertility was beginning to fall from the high level typical of underdeveloped countries.

The data collected are for 47 French-speaking "provinces" at about 1888.

Here, all variables are scaled to $[0, 100]$, where in the original, all but `"Catholic"` were scaled to $[0, 1]$.

Note

Files for all 182 districts in 1888 and other years have been available at http://opr.princeton.edu/archive/eufert/switz.html or http://opr.princeton.edu/archive/pefp/switz.asp.

They state that variables `Examination` and `Education` are averages for 1887, 1888 and 1889.

Source

Project "16P5", pages 549–551 in

Mosteller, F. and Tukey, J. W. (1977) *Data Analysis and Regression: A Second Course in Statistics*. Addison-Wesley, Reading Mass.

indicating their source as "Data used by permission of Franice van de Walle. Office of Population Research, Princeton University, 1976. Unpublished data assembled under NICHD contract number No 1-HD-O-2077."

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
require(stats); require(graphics)
pairs(swiss, panel = panel.smooth, main = "swiss data",
      col = 3 + (swiss$Catholic > 50))
summary(lm(Fertility ~ . , data = swiss))
```

`Theoph` *Pharmacokinetics of theophylline*

Description

The `Theoph` data frame has 132 rows and 5 columns of data from an experiment on the pharmacokinetics of theophylline.

Usage

```
Theoph
```

Format

This data frame contains the following columns:

Subject an ordered factor with levels `1`, ..., `12` identifying the subject on whom the observation was made. The ordering is by increasing maximum concentration of theophylline observed.

Wt weight of the subject (kg).

Dose dose of theophylline administered orally to the subject (mg/kg).

Time time since drug administration when the sample was drawn (hr).

conc theophylline concentration in the sample (mg/L).

Details

Boeckmann, Sheiner and Beal (1994) report data from a study by Dr. Robert Upton of the kinetics of the anti-asthmatic drug theophylline. Twelve subjects were given oral doses of theophylline then serum concentrations were measured at 11 time points over the next 25 hours.

These data are analyzed in Davidian and Giltinan (1995) and Pinheiro and Bates (2000) using a two-compartment open pharmacokinetic model, for which a self-starting model function, `SSfol`, is available.

Source

Boeckmann, A. J., Sheiner, L. B. and Beal, S. L. (1994), *NONMEM Users Guide: Part V*, NONMEM Project Group, University of California, San Francisco.

Davidian, M. and Giltinan, D. M. (1995) *Nonlinear Models for Repeated Measurement Data*, Chapman & Hall (section 5.5, p. 145 and section 6.6, p. 176)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer (Appendix A.29)

See Also

`SSfol`

Examples

```
require(stats); require(graphics)
coplot(conc ~ Time | Subject, data = Theoph, show.given = FALSE)
Theoph.4 <- subset(Theoph, Subject == 4)
fm1 <- nls(conc ~ SSfol(Dose, Time, lKe, lKa, lCl),
           data = Theoph.4)
summary(fm1)
plot(conc ~ Time, data = Theoph.4,
     xlab = "Time since drug administration (hr)",
     ylab = "Theophylline concentration (mg/L)",
     main = "Observed concentrations and fitted model",
     sub  = "Theophylline data - Subject 4 only",
     las = 1, col = 4)
xvals <- seq(0, par("usr")[2], length.out = 55)
lines(xvals, predict(fm1, newdata = list(Time = xvals)),
      col = 4)
```

Titanic *Survival of passengers on the Titanic*

Description

This data set provides information on the fate of passengers on the fatal maiden voyage of the ocean liner 'Titanic', summarized according to economic status (class), sex, age and survival.

Usage

```
Titanic
```

Format

A 4-dimensional array resulting from cross-tabulating 2201 observations on 4 variables. The variables and their levels are as follows:

No	Name	Levels
1	Class	1st, 2nd, 3rd, Crew
2	Sex	Male, Female
3	Age	Child, Adult
4	Survived	No, Yes

Details

The sinking of the Titanic is a famous event, and new books are still being published about it. Many well-known facts—from the proportions of first-class passengers to the 'women and children first' policy, and the fact that that policy was not entirely successful in saving the women and children in the third class—are reflected in the survival rates for various classes of passenger.

These data were originally collected by the British Board of Trade in their investigation of the sinking. Note that there is not complete agreement among primary sources as to the exact numbers on board, rescued, or lost.

Due in particular to the very successful film 'Titanic', the last years saw a rise in public interest in the Titanic. Very detailed data about the passengers is now available on the Internet, at sites such as *Encyclopedia Titanica* (http://www.rmplc.co.uk/eduweb/sites/phind).

Source

Dawson, Robert J. MacG. (1995), The 'Unusual Episode' Data Revisited. *Journal of Statistics Education*, **3**. http://www.amstat.org/publications/jse/v3n3/datasets.dawson.html

The source provides a data set recording class, sex, age, and survival status for each person on board of the Titanic, and is based on data originally collected by the British Board of Trade and reprinted in:

British Board of Trade (1990), *Report on the Loss of the 'Titanic' (S.S.)*. British Board of Trade Inquiry Report (reprint). Gloucester, UK: Allan Sutton Publishing.

Examples

```
require(graphics)
mosaicplot(Titanic, main = "Survival on the Titanic")
## Higher survival rates in children?
apply(Titanic, c(3, 4), sum)
## Higher survival rates in females?
apply(Titanic, c(2, 4), sum)
## Use loglm() in package 'MASS' for further analysis ...
```

`ToothGrowth` *The Effect of Vitamin C on Tooth Growth in Guinea Pigs*

Description

The response is the length of odontoblasts (teeth) in each of 10 guinea pigs at each of three dose levels of Vitamin C (0.5, 1, and 2 mg) with each of two delivery methods (orange juice or ascorbic acid).

Usage

```
ToothGrowth
```

Format

A data frame with 60 observations on 3 variables.

[,1]	len	numeric	Tooth length
[,2]	supp	factor	Supplement type (VC or OJ).
[,3]	dose	numeric	Dose in milligrams.

Source

C. I. Bliss (1952) *The Statistics of Bioassay*. Academic Press.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(graphics)
coplot(len ~ dose | supp, data = ToothGrowth, panel = panel.smooth,
       xlab = "ToothGrowth data: length vs dose, given type of supplement")
```

---

treering *Yearly Treering Data, -6000–1979*

Description

Contains normalized tree-ring widths in dimensionless units.

Usage

```
treering
```

Format

A univariate time series with 7981 observations. The object is of class `"ts"`.

Each tree ring corresponds to one year.

Details

The data were recorded by Donald A. Graybill, 1980, from Gt Basin Bristlecone Pine 2805M, 3726-11810 in Methuselah Walk, California.

Source

Time Series Data Library: http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/, series 'CA535.DAT'

References

For background on Bristlecone pines and Methuselah Walk, see http://www.sonic.net/bristlecone/; for some photos see http://www.ltrr.arizona.edu/~hallman/sitephotos/meth.html

---

trees — *Girth, Height and Volume for Black Cherry Trees*

Description

This data set provides measurements of the girth, height and volume of timber in 31 felled black cherry trees. Note that girth is the diameter of the tree (in inches) measured at 4 ft 6 in above the ground.

Usage

```
trees
```

Format

A data frame with 31 observations on 3 variables.

[,1]	Girth	numeric	Tree diameter in inches
[,2]	Height	numeric	Height in ft
[,3]	Volume	numeric	Volume of timber in cubic ft

Source

Ryan, T. A., Joiner, B. L. and Ryan, B. F. (1976) *The Minitab Student Handbook*. Duxbury Press.

References

Atkinson, A. C. (1985) *Plots, Transformations and Regression*. Oxford University Press.

Examples

```
require(stats); require(graphics)
pairs(trees, panel = panel.smooth, main = "trees data")
plot(Volume ~ Girth, data = trees, log = "xy")
coplot(log(Volume) ~ log(Girth) | Height, data = trees,
       panel = panel.smooth)
summary(fm1 <- lm(log(Volume) ~ log(Girth), data = trees))
summary(fm2 <- update(fm1, ~ . + log(Height), data = trees))
step(fm2)
## i.e., Volume ~= c * Height * Girth^2  seems reasonable
```

`UCBAdmissions` *Student Admissions at UC Berkeley*

Description

Aggregate data on applicants to graduate school at Berkeley for the six largest departments in 1973 classified by admission and sex.

Usage

```
UCBAdmissions
```

Format

A 3-dimensional array resulting from cross-tabulating 4526 observations on 3 variables. The variables and their levels are as follows:

No	Name	Levels
1	Admit	Admitted, Rejected
2	Gender	Male, Female
3	Dept	A, B, C, D, E, F

Details

This data set is frequently used for illustrating Simpson's paradox, see Bickel et al. (1975). At issue is whether the data show evidence of sex bias in admission practices. There were 2691 male applicants, of whom 1198 (44.5%) were admitted, compared with 1835 female applicants of whom 557 (30.4%) were admitted. This gives a sample odds ratio of 1.83, indicating that males were almost twice as likely to be admitted. In fact, graphical methods (as in the example below) or log-linear modelling show that the apparent association between admission and sex stems from differences in the tendency of males and females to apply to the individual departments (females used to apply *more* to departments with higher rejection rates).

This data set can also be used for illustrating methods for graphical display of categorical data, such as the general-purpose mosaic plot or the ourfold display for 2-by-2-by-k tables. See the home page of Michael Friendly (`http://www.math.yorku.ca/SCS/friendly.html`) for further information.

References

Bickel, P. J., Hammel, E. A., and O'Connell, J. W. (1975) Sex bias in graduate admissions: Data from Berkeley. *Science*, **187**, 398–403.

Examples

```
require(graphics)
## Data aggregated over departments
apply(UCBAdmissions, c(1, 2), sum)
mosaicplot(apply(UCBAdmissions, c(1, 2), sum),
           main = "Student admissions at UC Berkeley")
## Data for individual departments
opar <- par(mfrow = c(2, 3), oma = c(0, 0, 2, 0))
for(i in 1:6)
```

```
  mosaicplot(UCBAdmissions[,,i],
    xlab = "Admit", ylab = "Sex",
    main = paste("Department", LETTERS[i]))
mtext(expression(bold("Student admissions at UC Berkeley")),
      outer = TRUE, cex = 1.5)
par(opar)
```

`UKDriverDeaths` *Road Casualties in Great Britain 1969–84*

Description

`UKDriverDeaths` is a time series giving the monthly totals of car drivers in Great Britain killed or seriously injured Jan 1969 to Dec 1984. Compulsory wearing of seat belts was introduced on 31 Jan 1983.

`Seatbelts` is more information on the same problem.

Usage

```
UKDriverDeaths
Seatbelts
```

Format

`Seatbelts` is a multiple time series, with columns

`DriversKilled` car drivers killed.

`drivers` same as `UKDriverDeaths`.

`front` front-seat passengers killed or seriously injured.

`rear` rear-seat passengers killed or seriously injured.

`kms` distance driven.

`PetrolPrice` petrol price.

`VanKilled` number of van ('light goods vehicle') drivers.

`law` 0/1: was the law in effect that month?

Source

Harvey, A.C. (1989) *Forecasting, Structural Time Series Models and the Kalman Filter.* Cambridge University Press, pp. 519–523.

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press. http://www.ssfpack.com/dkbook/

References

Harvey, A. C. and Durbin, J. (1986) The effects of seat belt legislation on British road casualties: A case study in structural time series modelling. *Journal of the Royal Statistical Society* series B, **149**, 187–227.

Examples

```
require(stats); require(graphics)
## work with pre-seatbelt period to identify a model, use logs
work <- window(log10(UKDriverDeaths), end = 1982+11/12)
par(mfrow = c(3,1))
plot(work); acf(work); pacf(work)
par(mfrow = c(1,1))
(fit <- arima(work, c(1,0,0), seasonal = list(order= c(1,0,0))))
z <- predict(fit, n.ahead = 24)
ts.plot(log10(UKDriverDeaths), z$pred, z$pred+2*z$se, z$pred-2*z$se,
        lty = c(1,3,2,2), col = c("black", "red", "blue", "blue"))

## now see the effect of the explanatory variables
X <- Seatbelts[, c("kms", "PetrolPrice", "law")]
X[, 1] <- log10(X[, 1]) - 4
arima(log10(Seatbelts[, "drivers"]), c(1,0,0),
      seasonal = list(order= c(1,0,0)), xreg = X)
```

UKgas *UK Quarterly Gas Consumption*

Description

Quarterly UK gas consumption from 1960Q1 to 1986Q4, in millions of therms.

Usage

```
UKgas
```

Format

A quarterly time series of length 108.

Source

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press. http://www.ssfpack.com/dkbook/

Examples

```
## maybe str(UKgas) ; plot(UKgas) ...
```

UKLungDeaths *Monthly Deaths from Lung Diseases in the UK*

Description

Three time series giving the monthly deaths from bronchitis, emphysema and asthma in the UK, 1974–1979, both sexes (`ldeaths`), males (`mdeaths`) and females (`fdeaths`).

Usage

```
ldeaths
fdeaths
mdeaths
```

Source

P. J. Diggle (1990) *Time Series: A Biostatistical Introduction.* Oxford, table A.3

Examples

```
require(stats); require(graphics) # for time
plot(ldeaths)
plot(mdeaths, fdeaths)
## Better labels:
yr <- floor(tt <- time(mdeaths))
plot(mdeaths, fdeaths,
     xy.labels = paste(month.abb[12*(tt - yr)], yr-1900, sep="'"))
```

USAccDeaths *Accidental Deaths in the US 1973–1978*

Description

A time series giving the monthly totals of accidental deaths in the USA. The values for the first six months of 1979 are 7798 7406 8363 8460 9217 9316.

Usage

```
USAccDeaths
```

Source

P. J. Brockwell and R. A. Davis (1991) *Time Series: Theory and Methods.* Springer, New York.

USArrests *Violent Crime Rates by US State*

Description

This data set contains statistics, in arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973. Also given is the percent of the population living in urban areas.

Usage

```
USArrests
```

Format

A data frame with 50 observations on 4 variables.

[,1]	Murder	numeric	Murder arrests (per 100,000)
[,2]	Assault	numeric	Assault arrests (per 100,000)
[,3]	UrbanPop	numeric	Percent urban population
[,4]	Rape	numeric	Rape arrests (per 100,000)

Source

World Almanac and Book of facts 1975. (Crime rates).

Statistical Abstracts of the United States 1975. (Urban rates).

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

See Also

The `state` data sets.

Examples

```
require(graphics)
pairs(USArrests, panel = panel.smooth, main = "USArrests data")
```

USJudgeRatings *Lawyers' Ratings of State Judges in the US Superior Court*

Description

Lawyers' ratings of state judges in the US Superior Court.

Usage

```
USJudgeRatings
```

Format

A data frame containing 43 observations on 12 numeric variables.

[,1]	CONT	Number of contacts of lawyer with judge.
[,2]	INTG	Judicial integrity.
[,3]	DMNR	Demeanor.
[,4]	DILG	Diligence.
[,5]	CFMG	Case flow managing.
[,6]	DECI	Prompt decisions.
[,7]	PREP	Preparation for trial.
[,8]	FAMI	Familiarity with law.
[,9]	ORAL	Sound oral rulings.
[,10]	WRIT	Sound written rulings.
[,11]	PHYS	Physical ability.
[,12]	RTEN	Worthy of retention.

Source

New Haven Register, 14 January, 1977 (from John Hartigan).

Examples

```
require(graphics)
pairs(USJudgeRatings, main = "USJudgeRatings data")
```

USPersonalExpenditure

Personal Expenditure Data

Description

This data set consists of United States personal expenditures (in billions of dollars) in the categories; food and tobacco, household operation, medical and health, personal care, and private education for the years 1940, 1945, 1950, 1955 and 1960.

Usage

```
USPersonalExpenditure
```

Format

A matrix with 5 rows and 5 columns.

Source

The World Almanac and Book of Facts, 1962, page 756.

References

Tukey, J. W. (1977) *Exploratory Data Analysis*. Addison-Wesley.

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(stats) # for medpolish
USPersonalExpenditure
medpolish(log10(USPersonalExpenditure))
```

---

uspop *Populations Recorded by the US Census*

Description

This data set gives the population of the United States (in millions) as recorded by the decennial census for the period 1790–1970.

Usage

```
uspop
```

Format

A time series of 19 values.

Source

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(graphics)
plot(uspop, log = "y", main = "uspop data", xlab = "Year",
     ylab = "U.S. Population (millions)")
```

VADeaths *Death Rates in Virginia (1940)*

Description

Death rates per 1000 in Virginia in 1940.

Usage

```
VADeaths
```

Format

A matrix with 5 rows and 4 columns.

Details

The death rates are measured per 1000 population per year. They are cross-classified by age group (rows) and population group (columns). The age groups are: 50–54, 55–59, 60–64, 65–69, 70–74 and the population groups are Rural/Male, Rural/Female, Urban/Male and Urban/Female.

This provides a rather nice 3-way analysis of variance example.

Source

Molyneaux, L., Gilliam, S. K., and Florant, L. C.(1947) Differences in Virginia death rates by color, sex, age, and rural or urban residence. *American Sociological Review*, **12**, 525–535.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(stats); require(graphics)
n <- length(dr <- c(VADeaths))
nam <- names(VADeaths)
d.VAD <- data.frame(
 Drate = dr,
 age = rep(ordered(rownames(VADeaths)),length.out=n),
 gender= gl(2,5,n, labels= c("M", "F")),
 site =  gl(2,10,  labels= c("rural", "urban")))
coplot(Drate ~ as.numeric(age) | gender * site, data = d.VAD,
       panel = panel.smooth, xlab = "VADeaths data - Given: gender")
summary(aov.VAD <- aov(Drate ~ .^2, data = d.VAD))
opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(aov.VAD)
par(opar)
```

---

volcano *Topographic Information on Auckland's Maunga Whau Volcano*

Description

Maunga Whau (Mt Eden) is one of about 50 volcanos in the Auckland volcanic field. This data set gives topographic information for Maunga Whau on a 10m by 10m grid.

Usage

```
volcano
```

Format

A matrix with 87 rows and 61 columns, rows corresponding to grid lines running east to west and columns to grid lines running south to north.

Source

Digitized from a topographic map by Ross Ihaka. These data should not be regarded as accurate.

See Also

`filled.contour` for a nice plot.

Examples

```
require(grDevices); require(graphics)
filled.contour(volcano, color.palette = terrain.colors, asp = 1)
title(main = "volcano data: filled contour map")
```

---

warpbreaks *The Number of Breaks in Yarn during Weaving*

Description

This data set gives the number of warp breaks per loom, where a loom corresponds to a fixed length of yarn.

Usage

```
warpbreaks
```

Format

A data frame with 54 observations on 3 variables.

[,1]	breaks	numeric	The number of breaks
[,2]	wool	factor	The type of wool (A or B)
[,3]	tension	factor	The level of tension (L, M, H)

There are measurements on 9 looms for each of the six types of warp (AL, AM, AH, BL, BM, BH).

Source

Tippett, L. H. C. (1950) *Technological Applications of Statistics*. Wiley. Page 106.

References

Tukey, J. W. (1977) *Exploratory Data Analysis*. Addison-Wesley.

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

See Also

xtabs for ways to display these data as a table.

Examples

```
require(stats); require(graphics)
summary(warpbreaks)
opar <- par(mfrow = c(1,2), oma = c(0, 0, 1.1, 0))
plot(breaks ~ tension, data = warpbreaks, col = "lightgray",
     varwidth = TRUE, subset = wool == "A", main = "Wool A")
plot(breaks ~ tension, data = warpbreaks, col = "lightgray",
     varwidth = TRUE, subset = wool == "B", main = "Wool B")
mtext("warpbreaks data", side = 3, outer = TRUE)
par(opar)
summary(fm1 <- lm(breaks ~ wool*tension, data = warpbreaks))
```

```
anova(fm1)
```

`women` *Average Heights and Weights for American Women*

Description

This data set gives the average heights and weights for American women aged 30–39.

Usage

```
women
```

Format

A data frame with 15 observations on 2 variables.

`[,1]`	`height`	numeric	Height (in)
`[,2]`	`weight`	numeric	Weight (lbs)

Details

The data set appears to have been taken from the American Society of Actuaries *Build and Blood Pressure Study* for some (unknown to us) earlier year.

The World Almanac notes: "The figures represent weights in ordinary indoor clothing and shoes, and heights with shoes".

Source

The World Almanac and Book of Facts, 1975.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(graphics)
plot(women, xlab = "Height (in)", ylab = "Weight (lb)",
     main = "women data: American women aged 30-39")
```

`WorldPhones` *The World's Telephones*

Description

The number of telephones in various regions of the world (in thousands).

Usage

```
phones
```

Format

A matrix with 7 rows and 8 columns. The columns of the matrix give the figures for a given region, and the rows the figures for a year.

The regions are: North America, Europe, Asia, South America, Oceania, Africa, Central America.

The years are: 1951, 1956, 1957, 1958, 1959, 1960, 1961.

Source

AT&T (1961) *The World's Telephones*.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

Examples

```
require(graphics)
matplot(rownames(WorldPhones), WorldPhones, type = "b", log = "y",
        xlab = "Year", ylab = "Number of telephones (1000's)")
legend(1951.5, 80000, colnames(WorldPhones), col = 1:6, lty = 1:5,
       pch = rep(21, 7))
title(main = "World phones data: log scale for response")
```

WWWusage *Internet Usage per Minute*

Description

A time series of the numbers of users connected to the Internet through a server every minute.

Usage

```
WWWusage
```

Format

A time series of length 100.

Source

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press. http://www.ssfpack.com/dkbook/

References

Makridakis, S., Wheelwright, S. C. and Hyndman, R. J. (1998) *Forecasting: Methods and Applications.* Wiley.

Examples

```
require(graphics)
work <- diff(WWWusage)
par(mfrow = c(2,1)); plot(WWWusage); plot(work)
## Not run:
require(stats)
aics <- matrix(, 6, 6, dimnames=list(p=0:5, q=0:5))
for(q in 1:5) aics[1, 1+q] <- arima(WWWusage, c(0,1,q),
    optim.control = list(maxit = 500))$aic
for(p in 1:5)
   for(q in 0:5) aics[1+p, 1+q] <- arima(WWWusage, c(p,1,q),
       optim.control = list(maxit = 500))$aic
round(aics - min(aics, na.rm=TRUE), 2)
## End(Not run)
```

Chapter 3

The `grDevices` package

`grDevices-package` *The R Graphics Devices and Support for Colours and Fonts*

Description

Graphics devices and support for base and grid graphics

Details

This package contains functions which support both base and grid graphics.

For a complete list of functions, use `library(help="grDevices")`.

Author(s)

R Development Core Team and contributors worldwide

Maintainer: R Core Team ⟨R-core@r-project.org⟩

`as.graphicsAnnot` *Coerce an Object for Graphics Annotation*

Description

Coerce an R object into a form suitable for graphics annotation.

Usage

```
as.graphicsAnnot(x)
```

Arguments

`x`	an R object

Details

Expressions, calls and names (as used by plotmath) are passed through unchanged. All other objects with an S3 class (as determined by `is.object`) are coerced by `as.character` to character vectors.

All the **graphics** and **grid** functions which use this coerce calls and names to expressions internally.

Value

A language object or a character vector.

`boxplot.stats` *Box Plot Statistics*

Description

This function is typically called by another function to gather the statistics necessary for producing box plots, but may be invoked separately.

Usage

```
boxplot.stats(x, coef = 1.5, do.conf = TRUE, do.out = TRUE)
```

Arguments

`x`	a numeric vector for which the boxplot will be constructed (`NA`s and `NaN`s are allowed and omitted).
`coef`	this determines how far the plot 'whiskers' extend out from the box. If `coef` is positive, the whiskers extend to the most extreme data point which is no more than `coef` times the length of the box away from the box. A value of zero causes the whiskers to extend to the data extremes (and no outliers be returned).
`do.conf,do.out`	logicals; if `FALSE`, the `conf` or `out` component respectively will be empty in the result.

Details

The two 'hinges' are versions of the first and third quartile, i.e., close to `quantile(x, c(1,3)/4)`. The hinges equal the quartiles for odd n (where `n <- length(x)`) and differ for even n. Whereas the quartiles only equal observations for `n %% 4 == 1` ($n \equiv 1 \bmod 4$), the hinges do so *additionally* for `n %% 4 == 2` ($n \equiv 2 \bmod 4$), and are in the middle of two observations otherwise.

The notches (if requested) extend to `+/-1.58 IQR/sqrt(n)`. This seems to be based on the same calculations as the formula with 1.57 in Chambers *et al.* (1983, p. 62), given in McGill *et al.* (1978, p. 16). They are based on asymptotic normality of the median and roughly equal sample sizes for the two medians being compared, and are said to be rather insensitive to the underlying distributions of the samples. The idea appears to be to give roughly a 95% confidence interval for the difference in two medians.

Value

List with named components as follows:

`stats`	a vector of length 5, containing the extreme of the lower whisker, the lower 'hinge', the median, the upper 'hinge' and the extreme of the upper whisker.
`n`	the number of non-`NA` observations in the sample.
`conf`	the lower and upper extremes of the 'notch' (`if(do.conf)`). See the details.
`out`	the values of any data points which lie beyond the extremes of the whiskers (`if(do.out)`).

Note that `$stats` and `$conf` are sorted in *in*creasing order, unlike S, and that `$n` and `$out` include any `+- Inf` values.

References

Tukey, J. W. (1977) *Exploratory Data Analysis.* Section 2C.

McGill, R., Tukey, J. W. and Larsen, W. A. (1978) Variations of box plots. *The American Statistician* **32**, 12–16.

Velleman, P. F. and Hoaglin, D. C. (1981) *Applications, Basics and Computing of Exploratory Data Analysis.* Duxbury Press.

Emerson, J. D and Strenio, J. (1983). Boxplots and batch comparison. Chapter 3 of *Understanding Robust and Exploratory Data Analysis*, eds. D. C. Hoaglin, F. Mosteller and J. W. Tukey. Wiley.

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis.* Wadsworth & Brooks/Cole.

See Also

`fivenum`, `boxplot`, `bxp`.

Examples

```
require(stats)
x <- c(1:100, 1000)
(b1 <- boxplot.stats(x))
(b2 <- boxplot.stats(x, do.conf=FALSE, do.out=FALSE))
stopifnot(b1 $ stats == b2 $ stats) # do.out=F is still robust
boxplot.stats(x, coef = 3, do.conf=FALSE)
## no outlier treatment:
boxplot.stats(x, coef = 0)

boxplot.stats(c(x, NA)) # slight change : n is 101
(r <- boxplot.stats(c(x, -1:1/0)))
stopifnot(r$out == c(1000, -Inf, Inf))
```

cairo *Cairo-based SVG, PDF and PostScript Graphics Devices*

Description

Graphics devices for SVG, PDF and PostScript graphics files.

Usage

```
svg(filename = if(onefile) "Rplots.svg" else "Rplot%03d.svg",
    width = 7, height = 7, pointsize = 12,
    onefile = FALSE, bg = "white", antialias)

cairo_pdf(filename = if(onefile) "Rplots.pdf" else "Rplot%03d.pdf",
          width = 7, height = 7, pointsize = 12,
          onefile = FALSE, bg = "white", antialias)

cairo_ps(filename = if(onefile) "Rplots.ps" else "Rplot%03d.ps",
         width = 7, height = 7, pointsize = 12,
         onefile = FALSE, bg = "white", antialias)
```

Arguments

filename	the name of the output file. The page number is substituted if a C integer format is included in the character string, as in the default. (The result must be less than PATH_MAX characters long, and may be truncated if not. See postscript for further details.) Tilde expansion is performed where supported by the platform.
width	the width of the device in inches.
height	the height of the device in inches.
pointsize	the default pointsize of plotted text (in big points).
onefile	should all plots appear in one file or in separate files?
bg	the initial background colour: can be overridden by setting par("bg").
antialias	the type of anti-aliasing (if any) to be used. See X11.

Details

SVG (Scalar Vector Graphics) is a W3C standard for vector graphics. See http://www.w3.org/Graphics/SVG/. The output is SVG version 1.1 for onefile = FALSE (the default), otherwise SVG 1.2. (Very few SVG viewers are capable of displaying multi-page SVG files.)

Note that unlike postscript and pdf, cairo_pdf and cairo_ps sometimes record *bitmaps* and not vector graphics: a resolution of 72dpi is used. On the other hand, they can (on suitable platforms) include a much wider range of UTF-8 glyphs, and embed the fonts used. They are somewhat experimental.

R can be compiled without support for any of these devices: this will be reported if you attempt to use them on a system where they are not supported. They all require cairo version 1.2 or later.

If you plot more than one page on one of these devices and do not include something like %d for the sequence number in file (or set onefile=TRUE) the file will contain the last page plotted.

The cairo_ps output is not yet encapsulated (that is coming in cairo 1.6).

There is full support of transparency, but using this is one of the things liable to trigger bitmap output (and will always do so for `cairo_ps`).

Value

A plot device is opened: nothing is returned to the R interpreter.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the "R Internals Manual".

- The default device size is in pixels (`svg`) or inches.
- Font sizes are in big points.
- The default font family is Helvetica.
- Line widths are multiples of 1/96 inch.
- Circle radii have a minumum of 1/72 inch.
- Colours are interpreted by the viewing application.

See Also

`Devices`, `dev.print`, `pdf`, `postscript`

`capabilities` to see if cairo is supported.

`check.options` *Set Options with Consistency Checks*

Description

Utility function for setting options with some consistency checks. The `attributes` of the new settings in `new` are checked for consistency with the *model* (often default) list in `name.opt`.

Usage

```
check.options(new, name.opt, reset = FALSE, assign.opt = FALSE,
              envir = .GlobalEnv,
              check.attributes = c("mode", "length"),
              override.check = FALSE)
```

Arguments

`new`	a *named* list
`name.opt`	character with the name of R object containing the default list.
`reset`	logical; if `TRUE`, reset the options from `name.opt`. If there is more than one R object with name `name.opt`, remove the first one in the `search()` path.
`assign.opt`	logical; if `TRUE`, assign the ...
`envir`	the `environment` used for `get` and `assign`.
`check.attributes`	character containing the attributes which `check.options` should check.

override.check
: logical vector of length length(new) (or 1 which entails recycling). For those new[i] where override.check[i] == TRUE, the checks are overridden and the changes made anyway.

Value

A list of components with the same names as the one called name.opt. The values of the components are changed from the new list, as long as these pass the checks (when these are not overridden according to override.check).

Note

Option "names" is exempt from all the checks or warnings, as in the application it can be NULL or a variable-length character vector.

Author(s)

Martin Maechler

See Also

ps.options and pdf.options, which use check.options.

Examples

```
(L1 <- list(a=1:3, b=pi, ch="CH"))
check.options(list(a=0:2), name.opt = "L1")
check.options(NULL, reset = TRUE, name.opt = "L1")
```

---

chull *Compute Convex Hull of a Set of Points*

Description

Computes the subset of points which lie on the convex hull of the set of points specified.

Usage

```
chull(x, y = NULL)
```

Arguments

x, y
: coordinate vectors of points. This can be specified as two vectors x and y, a 2-column matrix x, a list x with two components, etc, see xy.coords.

Details

xy.coords is used to interpret the specification of the points. The algorithm is that given by Eddy (1977).

'Peeling' as used in the S function chull can be implemented by calling chull recursively.

Value

An integer vector giving the indices of the points lying on the convex hull, in clockwise order.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Eddy, W. F. (1977) A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software*, **3**, 398–403.

Eddy, W. F. (1977) Algorithm 523. CONVEX, A new convex hull algorithm for planar sets[Z]. *ACM Transactions on Mathematical Software*, **3**, 411–412.

See Also

xy.coords,polygon

Examples

```
require(stats)
X <- matrix(rnorm(2000), ncol = 2)
chull(X)
## Not run:
  # Example usage from graphics package
  plot(X, cex = 0.5)
  hpts <- chull(X)
  hpts <- c(hpts, hpts[1])
  lines(X[hpts, ])
## End(Not run)
```

---

cm *Unit Transformation*

Description

Translates from inches to cm (centimeters).

Usage

```
cm(x)
```

Arguments

x	numeric vector

Examples

```
cm(1)# = 2.54

## Translate *from* cm *to* inches:

10 / cm(1) # -> 10cm  are 3.937 inches
```

`col2rgb` *Color to RGB Conversion*

Description

R color to RGB (red/green/blue) conversion.

Usage

```
col2rgb(col, alpha = FALSE)
```

Arguments

`col`	vector of any of the three kind of R colors, i.e., either a color name (an element of `colors()`), a hexadecimal string of the form `"#rrggbb"`, or an integer `i` meaning `palette()[i]`. Non-string values are coerced to integer.
`alpha`	logical value indicating whether alpha channel values should be returned.

Details

For integer colors, `0` is shorthand for the current `par("bg")` (and hence is only relevant to base graphics), and `NA` means transparent.

For character colors, `"NA"` is equivalent to `NA` above.

Value

an integer matrix with three or four rows and number of columns the length (and names if any) as `col`.

Author(s)

Martin Maechler

See Also

`rgb`, `colors`, `palette`, etc.

Examples

```
col2rgb("peachpuff")
col2rgb(c(blu = "royalblue", reddish = "tomato")) # names kept

col2rgb(1:8)# the ones from the palette() :

col2rgb(paste("gold", 1:4, sep=""))

col2rgb("#08a0ff")
## all three kind of colors mixed :
col2rgb(c(red="red", palette= 1:3, hex="#abcdef"))

##-- NON-INTRODUCTORY examples --

grC <- col2rgb(paste("gray",0:100,sep=""))
```

```
table(print(diff(grC["red",])))# '2' or '3': almost equidistant
## The 'named' grays are in between {"slate gray" is not gray, strictly}
col2rgb(c(g66="gray66", darkg= "dark gray", g67="gray67",
          g74="gray74", gray =      "gray", g75="gray75",
          g82="gray82", light="light gray", g83="gray83"))

crgb <- col2rgb(cc <- colors())
colnames(crgb) <- cc
t(crgb)## The whole table

ccodes <- c(256^(2:0) %*% crgb)## = internal codes
## How many names are 'aliases' of each other:
table(tcc <- table(ccodes))
length(uc <- unique(sort(ccodes))) # 502
## All the multiply named colors:
mult <- uc[tcc >= 2]
cl <- lapply(mult, function(m) cc[ccodes == m])
names(cl) <- apply(col2rgb(sapply(cl, function(x)x[1])),
                   2, function(n)paste(n, collapse=","))
utils::str(cl)
## Not run:
 if(require(xgobi)) { ## Look at the color cube dynamically :
   tc <- t(crgb[, !duplicated(ccodes)])
   table(is.gray <- tc[,1] == tc[,2] & tc[,2] == tc[,3])# (397, 105)
   xgobi(tc, color = c("gold", "gray")[1 + is.gray])
 }
## End(Not run)
```

`colorRamp` *Color interpolation*

Description

These functions return functions that interpolate a set of given colors to create new color palettes (like `topo.colors`) and color ramps, functions that map the interval $[0, 1]$ to colors (like `grey`).

Usage

```
colorRamp(colors, bias = 1, space = c("rgb", "Lab"),
          interpolate = c("linear", "spline"))
colorRampPalette(colors, ...)
```

Arguments

`colors`	Colors to interpolate
`bias`	A positive number. Higher values give more widely spaced colors at the high end.
`space`	Interpolation in RGB or CIE Lab color spaces
`interpolate`	Use spline or linear interpolation.
`...`	arguments to pass to `colorRamp`.

Details

The CIE Lab color space is approximately perceptually uniform, and so gives smoother and more uniform color ramps. On the other hand, palettes that vary from one hue to another via white may have a more symmetrical appearance in RGB space.

The conversion formulas in this function do not appear to be completely accurate and the color ramp may not reach the extreme values in Lab space. Future changes in the R color model may change the colors produced with `space="Lab"`.

Value

`colorRamp` returns a function that maps values between 0 and 1 to colors. `colorRampPalette` returns a function that takes an integer argument and returns that number of colors interpolating the given sequence (similar to `heat.colors` or `terrain.colors`.

See Also

Good starting points for interpolation are the "sequential" and "diverging" ColorBrewer palettes in the RColorBrewer package

Examples

```
require(graphics)

## Here space="rgb" gives palettes that vary only in saturation,
## as intended.
## With space="Lab" the steps are more uniform, but the hues
## are slightly purple.
filled.contour(volcano,
               color.palette =
                   colorRampPalette(c("red", "white", "blue")),
               asp = 1)
filled.contour(volcano,
               color.palette =
                   colorRampPalette(c("red", "white", "blue"),
                                    space = "Lab"),
               asp = 1)

## Interpolating a 'sequential' ColorBrewer palette
YlOrBr <- c("#FFFFD4", "#FED98E", "#FE9929", "#D95F0E", "#993404")
filled.contour(volcano,
               color.palette = colorRampPalette(YlOrBr, space = "Lab"),
               asp = 1)
filled.contour(volcano,
               color.palette = colorRampPalette(YlOrBr, space = "Lab",
                                                bias = 0.5),
               asp = 1)

## 'jet.colors' is "as in Matlab"
## (and hurting the eyes by over-saturation)
jet.colors <-
  colorRampPalette(c("#00007F", "blue", "#007FFF", "cyan",
                     "#7FFF7F", "yellow", "#FF7F00", "red", "#7F0000"))
filled.contour(volcano, color = jet.colors, asp = 1)

## space="Lab" helps when colors don't form a natural sequence
```

```
m <- outer(1:20,1:20,function(x,y) sin(sqrt(x*y)/3))
rgb.palette <- colorRampPalette(c("red", "orange", "blue"),
                                space = "rgb")
Lab.palette <- colorRampPalette(c("red", "orange", "blue"),
                                space = "Lab")
filled.contour(m, col = rgb.palette(20))
filled.contour(m, col = Lab.palette(20))
```

`colors` *Color Names*

Description

Returns the built-in color names which R knows about.

Usage

```
colors()
colours()
```

Details

These color names can be used with a `col=` specification in graphics functions.

An even wider variety of colors can be created with primitives `rgb` and `hsv` or the derived `rainbow`, `heat.colors`, etc.

Value

A character vector containing all the built-in color names.

See Also

`palette` for setting the 'palette' of colors for `par(col=`*<num>*`)`; `rgb`, `hsv`, `hcl`, `gray`; `rainbow` for a nice example; and `heat.colors`, `topo.colors` for images.

`col2rgb` for translating to RGB numbers and extended examples.

Examples

```
cl <- colors()
length(cl); cl[1:20]
```

`contourLines` *Calculate Contour Lines*

Description

Calculate contour lines for a given set of data.

Usage

```
contourLines(x = seq(0, 1, length.out = nrow(z)),
             y = seq(0, 1, length.out = ncol(z)),
             z, nlevels = 10,
             levels = pretty(range(z, na.rm=TRUE), nlevels))
```

Arguments

`x,y`	locations of grid lines at which the values in `z` are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If `x` is a `list`, its components `x$x` and `x$y` are used for `x` and `y`, respectively. If the list has component `z` this is used for `z`.
`z`	a matrix containing the values to be plotted (`NA`s are allowed). Note that `x` can be used instead of `z` for convenience.
`nlevels`	number of contour levels desired **iff** `levels` is not supplied.
`levels`	numeric vector of levels at which to draw contour lines.

Details

`contourLines` draws nothing, but returns a set of contour lines.

There is currently no documentation about the algorithm. The source code is in '*R_HOME*/src/main/plot3d.c'.

Value

A list of contours. Each contour is a list with elements:

`level`	The contour level.
`x`	The x-coordinates of the contour.
`y`	The y-coordinates of the contour.

See Also

`options("max.countour.segments")` for the maximal complexity of a single contour line.

`contour`.

Examples

```
x <- 10*1:nrow(volcano)
y <- 10*1:ncol(volcano)
contourLines(x, y, volcano)
```

convertColor *Convert between colour spaces*

Description

Convert colours between standard colour space representations. This function is experimental.

Usage

```
convertColor(color, from, to, from.ref.white, to.ref.white,
             scale.in=1, scale.out=1, clip=TRUE)
```

Arguments

color	A matrix whose rows specify colors.
from,to	Input and output color spaces. See 'Details' below.
from.ref.white,to.ref.white	Reference whites or NULL if these are built in to the definition, as for RGB spaces. D65 is the default, see 'Details' for others.
scale.in, scale.out	Input is divided by scale.in, output is multiplied by scale.out. Use NULL to suppress scaling when input or output is not numeric.
clip	If TRUE, truncate RGB output to [0,1], FALSE return out-of-range RGB, NA set out of range colors to NaN.

Details

Color spaces are specified by objects of class colorConverter, created by colorConverter or make.rgb. Built-in color spaces may be referenced by strings: "XYZ", "sRGB", "Apple RGB", "CIE RGB", "Lab", "Luv". The converters for these colour spaces are in the object colorspaces.

The "sRGB" color space is that used by standard PC monitors. "Apple RGB" is used by Apple monitors. "Lab" and "Luv" are approximately perceptually uniform spaces standardized by the Commission Internationale d'Eclairage. XYZ is a 1931 CIE standard capable of representing all visible colors (and then some), but not in a perceptually uniform way.

The Lab and Luv spaces describe colors of objects, and so require the specification of a reference 'white light' color. Illuminant D65 is a standard indirect daylight, Illuminant D50 is close to direct sunlight, and Illuminant A is the light from a standard incandescent bulb. Other standard CIE illuminants supported are B, C, E and D55. RGB colour spaces are defined relative to a particular reference white, and can be only approximately translated to other reference whites. The Bradford chromatic adaptation algorithm is used for this.

The RGB color spaces are specific to a particular class of display. An RGB space cannot represent all colors, and the clip option controls what is done to out-of-range colors.

Value

A 3-row matrix whose columns specify the colors.

References

For all the conversion equations http://www.brucelindbloom.com/

For the white points http://www.efg2.com/Lab/Graphics/Colors/Chromaticity.htm

See Also

col2rgb and colors for ways to specify colors in graphics.

make.rgb for specifying other colour spaces.

Examples

```
require(graphics); require(stats) # for na.omit
par(mfrow=c(2,2))
## The displayable colors from four planes of Lab space
ab <- expand.grid(a=(-10:15)*10,b=(-15:10)*10)

Lab <- cbind(L=20,ab)
srgb <- convertColor(Lab,from="Lab",to="sRGB",clip=NA)
clipped <- attr(na.omit(srgb),"na.action")
srgb[clipped,] <- 0
cols <- rgb(srgb[,1],srgb[,2],srgb[,3])
image((-10:15)*10,(-15:10)*10,matrix(1:(26*26),ncol=26),col=cols,
  xlab="a",ylab="b",main="Lab: L=20")

Lab <- cbind(L=40,ab)
srgb <- convertColor(Lab,from="Lab",to="sRGB",clip=NA)
clipped <- attr(na.omit(srgb),"na.action")
srgb[clipped,] <- 0
cols <- rgb(srgb[,1],srgb[,2],srgb[,3])
image((-10:15)*10,(-15:10)*10,matrix(1:(26*26),ncol=26),col=cols,
  xlab="a",ylab="b",main="Lab: L=40")

Lab <- cbind(L=60,ab)
srgb <- convertColor(Lab,from="Lab",to="sRGB",clip=NA)
clipped <- attr(na.omit(srgb),"na.action")
srgb[clipped,] <- 0
cols <- rgb(srgb[,1],srgb[,2],srgb[,3])
image((-10:15)*10,(-15:10)*10,matrix(1:(26*26),ncol=26),col=cols,
  xlab="a",ylab="b",main="Lab: L=60")

Lab <- cbind(L=80,ab)
srgb <- convertColor(Lab,from="Lab",to="sRGB",clip=NA)
clipped <- attr(na.omit(srgb),"na.action")
srgb[clipped,] <- 0
cols <- rgb(srgb[,1],srgb[,2],srgb[,3])
image((-10:15)*10,(-15:10)*10,matrix(1:(26*26),ncol=26),col=cols,
  xlab="a",ylab="b",main="Lab: L=80")

(cols <- t(col2rgb(palette())))
(lab <- convertColor(cols,from="sRGB",to="Lab",scale.in=255))
round(convertColor(lab,from="Lab",to="sRGB",scale.out=255))
```

dev.interactive *Is the Current Graphics Device Interactive ?*

Description

Test if the current graphics device (or that which would be opened) is interactive.

Usage

```
dev.interactive(orNone = FALSE)

deviceIsInteractive(name = NULL)
```

Arguments

orNone	logical; if TRUE, the function also returns TRUE when .Device == "null device" and getOption("device") is among the known interactive devices.
name	one or more device names as a character vector, or NULL to give the existing list.

Details

The X11 (Unix), windows (Windows) and quartz (MacOS X, on-screen types only) are regarded as interactive, together with JavaGD (from the package of the same name) and CairoWin and CairoX11 (from package **Cairo**). Packages can add their devices to the list by calling deviceIsInteractive.

Value

dev.interactive() returns a logical, TRUE if and only if an interactive (screen) device is in use.

deviceIsInteractive returns the updated list of known interactive devices, invisibly unless name = NULL.

See Also

Devices for the available devices on your platform.

Examples

```
dev.interactive()
print(deviceIsInteractive(NULL))
```

dev.size *Find Size of Device Surface*

Description

Find the dimensions of the device surface of the current device.

Usage

```
dev.size(units = c("in", "cm", "px"))
```

Arguments

units	the units in which to return the value – inches, cm, or pixels (device units).

Value

A two-element numeric vector giving width and height of the current device; a new device is opened if there is none, similarly to `dev.new()`.

See Also

The size information in inches can be obtained by `par("din")`, but this provides a way to access it independent of the graphics sub-system in use.

Examples

```
dev.size("cm")
```

dev.xxx *Control Multiple Devices*

Description

These functions provide control over multiple graphics devices.

Usage

```
dev.cur()
dev.list()
dev.next(which = dev.cur())
dev.prev(which = dev.cur())
dev.off(which = dev.cur())
dev.set(which = dev.next())
dev.new(...)
graphics.off()
```

Arguments

which	An integer specifying a device number.
...	arguments to be passed to the device selected.

Details

Only one device is the 'active' device: this is the device in which all graphics operations occur. There is a `"null device"` which is always open but is really a placeholder: any attempt to use it will open a new device specified by `getOption("device")`.

Devices are associated with a name (e.g., `"X11"` or `"postscript"`) and a number in the range 1 to 63; the `"null device"` is always device 1. Once a device has been opened the null device is not considered as a possible active device. There is a list of open devices, and this is considered as a circular list not including the null device. `dev.next` and `dev.prev` select the next open device in the appropriate direction, unless no device is open.

`dev.off` shuts down the specified (by default the current) device. If the current device is shut down and any other devices are open, the next open device is made current. It is an error to attempt to shut down device 1. `graphics.off()` shuts down all open graphics devices.

`dev.set` makes the specified device the active device. If there is no device with that number, it is equivalent to `dev.next`. If `which = 1` it opens a new device and selects that.

`dev.new` opens a new device. Normally R will open a new device automatically when needed, but this enables you to open further devices in a platform-independent way. (For which device is used see `getOption("device")`.) Note that care is needed with file-based devices such as `pdf` and `postscript` and in that case file names such as 'Rplots.pdf', 'Rplots1.pdf', ..., 'Rplots999.pdf' are tried in turn. Only named arguments are passed to the device, and then only if they match the argument list of the device. Even so, case is needed with the interpretation of e.g. `width`, and for the standard bitmap devices `units="in", res=72` is forced if neither is supplied but both `width` and `height` are.

Value

`dev.cur` returns a length-one named integer vector giving the number and name of the active device, or 1, the null device, if none is active.

`dev.list` returns the numbers of all open devices, except device 1, the null device. This is a numeric vector with a `names` attribute giving the device names, or `NULL` is there is no open device.

`dev.next` and `dev.prev` return the number and name of the next / previous device in the list of devices. This will be the null device if and only if there are no open devices.

`dev.off` returns the number and name of the new active device (after the specified device has been shut down).

`dev.set` returns the number and name of the new active device.

`dev.new` returns the return value of the device opened, usually invisible `NULL`.

See Also

`Devices`, such as `postscript`, etc.

`layout` and its links for setting up plotting regions on the current device.

Examples

```
## Not run:
## Unix-specific example
x11()
plot(1:10)
x11()
plot(rnorm(10))
```

```
dev.set(dev.prev())
abline(0,1)# through the 1:10 points
dev.set(dev.next())
abline(h=0, col="gray")# for the residual plot
dev.set(dev.prev())
dev.off(); dev.off()#- close the two X devices
## End(Not run)
```

---

dev2 *Copy Graphics Between Multiple Devices*

Description

`dev.copy` copies the graphics contents of the current device to the device specified by `which` or to a new device which has been created by the function specified by `device` (it is an error to specify both `which` and `device`). (If recording is off on the current device, there are no contents to copy: this will result in no plot or an empty plot.) The device copied to becomes the current device.

`dev.print` copies the graphics contents of the current device to a new device which has been created by the function specified by `device` and then shuts the new device.

`dev.copy2eps` is similar to `dev.print` but produces an EPSF output file in portrait orientation (`horizontal = FALSE`). `dev.copy2pdf` is the analogue for PDF output.

`dev.control` allows the user to control the recording of graphics operations in a device. If `displaylist` is `"inhibit"` (`"enable"`) then recording is turned off (on). It is only safe to change this at the beginning of a plot (just before or just after a new page). Initially recording is on for screen devices, and off for print devices.

Usage

```
dev.copy(device, ..., which = dev.next())
dev.print(device = postscript, ...)
dev.copy2eps(...)
dev.copy2pdf(..., out.type = "pdf")
dev.control(displaylist = c("inhibit", "enable"))
```

Arguments

`device`	A device function (e.g., `x11`, `postscript`, ...)
`...`	Arguments to the `device` function above: for `dev.copy2eps` arguments to `postscript` and for `dev.copy2pdf`, arguments to `pdf`. For `dev.print`, this includes `which` and by default any `postscript` arguments.
`which`	A device number specifying the device to copy to.
`out.type`	The name of the output device: can be `"pdf"`, or `"quartz"` (some Mac OS X builds) or `"cairo"` (some Unix-alikes, see `cairo_pdf`).
`displaylist`	A character string: the only valid values are `"inhibit"` and `"enable"`.

Details

Note that these functions copy the *device region* and not a plot: the background colour of the device surface is part of what is copied. Most screen devices default to a transparent background, which is probably not what is needed when copying to a device such as `png`.

For `dev.copy2eps` and `dev.copy2pdf`, `width` and `height` are taken from the current device unless otherwise specified. If just one of `width` and `height` is specified, the other is adjusted to preserve the aspect ratio of the device being copied. The default file name is `Rplot.eps` or `Rplot.pdf`, and can be overridden by specifying a `file` argument.

Copying to devices such as `postscript` and `pdf` which need font families pre-specified needs extra care – R is unaware of which families were used in a plot and so they will need to manually specified by the `fonts` argument passed as part of `...`. Similarly, if the device to be copied from was opened with a `family` argument, a suitable `family` argument will need to be included in `...`.

The default for `dev.print` is to produce and print a postscript copy, if `options("printcmd")` is set suitably.

`dev.print` is most useful for producing a postscript print (its default) when the following applies. Unless `file` is specified, the plot will be printed. Unless `width`, `height` and `pointsize` are specified the plot dimensions will be taken from the current device, shrunk if necessary to fit on the paper. (`pointsize` is rescaled if the plot is shrunk.) If `horizontal` is not specified and the plot can be printed at full size by switching its value this is done instead of shrinking the plot region.

If `dev.print` is used with a specified `device` (even `postscript`) it sets the width and height in the same way as `dev.copy2eps`. This will not be appropriate unless the device specifies dimensions in inches, in particular not for `png` and `jpeg`.

Value

`dev.copy` returns the name and number of the device which has been copied to.

`dev.print`, `dev.copy2eps` and `dev.copy2pdf` return the name and number of the device which has been copied from.

Note

Most devices (including all screen devices) have a display list which records all of the graphics operations that occur in the device. `dev.copy` copies graphics contents by copying the display list from one device to another device. Also, automatic redrawing of graphics contents following the resizing of a device depends on the contents of the display list.

After the command `dev.control("inhibit")`, graphics operations are not recorded in the display list so that `dev.copy` and `dev.print` will not copy anything and the contents of a device will not be redrawn automatically if the device is resized.

The recording of graphics operations is relatively expensive in terms of memory so the command `dev.control("inhibit")` can be useful if memory usage is an issue.

See Also

`dev.cur` and other `dev.xxx` functions.

Examples

```
## Not run:
x11()
```

```
plot(rnorm(10), main="Plot 1")
dev.copy(device=x11)
mtext("Copy 1", 3)
dev.print(width=6, height=6, horizontal=FALSE) # prints it
dev.off(dev.prev())
dev.off()
## End(Not run)
```

`dev2bitmap` *Graphics Device for Bitmap Files via GhostScript*

Description

`bitmap` generates a graphics file. `dev2bitmap` copies the current graphics device to a file in a graphics format.

Usage

```
bitmap(file, type = "png16m", height = 7, width = 7, res = 72,
       units = "in", pointsize, taa = NA, gaa = NA, ...)

dev2bitmap(file, type = "png16m", height = 7, width = 7, res = 72,
           units = "in", pointsize, ...,
           method = c("postscript", "pdf"), taa = NA, gaa = NA)
```

Arguments

`file`	The output file name, with an appropriate extension.
`type`	The type of bitmap. the default is `"png256"`.
`width, height`	Dimensions of the display region.
`res`	Resolution, in dots per inch.
`units`	The units in which `height` and `width` are given. Can be `in` (inches), `px` (pixels), `cm` or `mm`.
`pointsize`	The pointsize to be used for text: defaults to something reasonable given the width and height
`...`	Other parameters passed to `postscript` or `pdf`.
`method`	Should the plot be done by `postscript` or `pdf`?
`taa, gaa`	Number of bits of antialiasing for text and for graphics respectively. Usually 4 (for best effect) or 2. Not supported on all types.

Details

`dev2bitmap` works by copying the current device to a `postscript` or `pdf` device, and post-processing the output file using `ghostscript`. `bitmap` works in the same way using a `postscript` device and post-processing the output as 'printing'.

You will need `ghostscript`: the full path to the executable can be set by the environment variable `R_GSCMD`. (If this is unset the command `"gs"` is used, which will work if it is in your path.)

The types available will depend on the version of `ghostscript`, but are likely to include `"pcxmono"`, `"pcxgray"`, `"pcx16"`, `"pcx256"`, `"pcx24b"`, `"pcxcmyk"`, `"pbm"`, `"pbmraw"`, `"pgm"`, `"pgmraw"`, `"pgnm"`, `"pgnmraw"`, `"pnm"`, `"pnmraw"`, `"ppm"`, `"ppmraw"`, `"pkm"`, `"pkmraw"`, `"tiffcrle"`, `"tiffg3"`, `"tiffg32d"`, `"tiffg4"`, `"tifflzw"`, `"tiffpack"`, `"tiff12nc"`, `"tiff24nc"`, `"psmono"`, `"psgray"`, `"psrgb"`, `"bit"`, `"bitrgb"`, `"bitcmyk"`, `"pngmono"`, `"pnggray"`, `"pngalpha"`, `"png16"`, `"png256"`, `"png16m"`, `"png48"`, `"jpeg"`, `"jpeggray"`, `"pdfwrite"`.

The default type, `"png16m"` supports 24-bit colour and anti-aliasing. Versions of R prior to 2.7.0 defaulted to `"png256"`, which used a palette of 256 colours and could be a more compact representation. Monochrome graphs can used `"pngmono"`, or `"pnggray"` if anti-aliasing is desired.

For formats which contain a single image, a file specification like `Rplots%03d.png` can be used: this is interpreted by GhostScript.

For `dev2bitmap` if just one of `width` and `height` is specified, the other is chosen to preserve aspect ratio of the device being copied. The main reason to prefer `method = "pdf"` over the default would be to allow semi-transparent colours to be used.

For graphics parameters such as `"cra"` that need to work in pixels, the default resolution of 72dpi is always used.

Value

None.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the "R Internals Manual". These devices follow the underlying device, so when viewed at the stated `res`:

- The default device size is 7 inches square.
- Font sizes are in big points.
- The default font family is (for the standard GhostScript setup) URW Nimbus Sans.
- Line widths are as a multiple of 1/96 inch, with no minimum.
- Circle of any radius are allowed.
- Colours are interpreted by the viewing/printing application.

See Also

`savePlot`, which for `windows` and `X11(type = "Cairo")` provides a simple way to record a PNG record of the current plot.

`postscript`, `pdf`, `png` and `jpeg` and on Windows `bmp`.

To display an array of data, see `image`.

devAskNewPage *Prompt before New Page*

Description

This function can be used to control (for the current device) whether the user is prompted before starting a new page of output.

Usage

```
devAskNewPage(ask = NULL)
```

Arguments

ask	NULL or a logical value. If TRUE, the user is prompted before a new page of output is started.

Details

If the current device is the null device, this will open a graphics device.

The default argument just returns the current setting and does not change it.

The default value when a device is opened is taken from the setting of options("device.ask.default").

The precise circumstances when the user will be asked to confirm a new page depend on the graphics subsystem. Obviously this needs to be an interactive session. In addition 'recording' needs to be in operation, so only when the display list is enabled (see dev.control) which it usually is only on a screen device.

Value

The current prompt setting *before* any new setting is applied.

See Also

plot.new, grid.newpage

Devices *List of Graphical Devices*

Description

The following graphics devices are currently available:

- postscript Writes PostScript graphics commands to a file
- pdf Write PDF graphics commands to a file
- pictex Writes LaTeX/PicTeX graphics commands to a file
- xfig Device for XFIG graphics file format
- bitmap bitmap pseudo-device via GhostScript (if available).

The following devices will be functional if R was compiled to use them (they exist but will return with a warning on other systems):

- `X11` The graphics device for the X11 Window system
- `bmp` Windows bitmap device
- `jpeg` JPEG bitmap device
- `png` PNG bitmap device
- `tiff` TIFF bitmap device
- `cairo_pdf`, `cairo_ps` PDF and PostScript devices based on cairo graphics.
- `quartz` The graphics device for the Mac OS X native Quartz 2d graphics system. (This is only functional on Mac OS X, but unlike R < 2.7.0 can be used from the command line and not just from the GUI console.)

Details

If no device is open, using a high-level graphics function will cause a device to be opened. Which device is given by `options("device")` which is initially set as the most appropriate for each platform: a screen device for most interactive use and `pdf` (or the setting of `R_DEFAULT_DEVICE`) otherwise. The exception is interactive use under Unix if no screen device is known to be available, when `pdf()` is used.

See Also

The individual help files for further information on any of the devices listed here; `X11.options`, `quartz.options`, `ps.options` and `pdf.options` for how to customize devices.

`dev.interactive`, `dev.cur`, `dev.print`, `graphics.off`, `image`, `dev2bitmap`.

`capabilities` to see if `X11`, `jpeg` `png` and `quartz` are available.

Examples

```
## Not run:
## open the default screen device on this platform if no device is
## open
if(dev.cur() == 1) dev.new()
## End(Not run)
```

`embedFonts` *Embed Fonts in PostScript and PDF*

Description

Runs Ghostscript to process a PDF or PostScript file and embed all fonts in the file.

Usage

```
embedFonts(file, format, outfile = file, fontpaths = "",
           options = "")
```

Arguments

file	a character string giving the name of the original file.
format	either "pswrite" or "pdfwrite". If not specified, it is guessed from the suffix of file.
outfile	the name of the new file (with fonts embedded).
fontpaths	a character vector giving directories that Ghostscript will search for fonts.
options	a character string containing further options to Ghostscript.

Details

This function is not necessary if you just use the standard default fonts for PostScript and PDF output.

If you use a special font, this function is useful for embedding that font in your PostScript or PDF document so that it can be shared with others without them having to install your special font (provided the font licence allows this).

If the special font is not installed for Ghostscript, you will need to tell Ghostscript where the font is, using something like options="-sFONTPATH=path/to/font".

This function relies on a suitable Ghostscript executable being in your path, or the environment variable R_GSCMD (the same as bitmap) being set as the full path to the Ghostscript executable. This defaults to "gs".

Note that Ghostscript may do font substitution, so the font embedded may differ from that specified in the original file.

Value

The shell command used to invoke Ghostscript is returned invisibly. This may be useful for debugging purposes as you can run the command by hand in a shell to look for problems.

See Also

postscriptFonts, Devices.

Paul Murrell and Brian Ripley (2006) Non-standard fonts in PostScript and PDF graphics. *R News*, 6(2):41–47. http://cran.r-project.org/doc/Rnews/Rnews_2006-2.pdf.

---

extendrange *Extend a Numerical Range by a Small Percentage*

Description

Extends a numerical range by a small percentage, i.e., fraction, *on both sides*.

Usage

```
extendrange(x, r = range(x, na.rm = TRUE), f = 0.05)
```

Arguments

`x`	numeric vector; not used if `r` is specified.
`r`	numeric vector of length 2; defaults to the `range` of `x`.
`f`	number specifying the fraction by which the range should be extended.

Value

A numeric vector of length 2, `r + c(-f,f) * diff(r)`.

See Also

`range`; `pretty` which can be considered a sophisticated extension of `extendrange`.

Examples

```
x <- 1:5
(r <- range(x))          # 1    5
extendrange(x)           # 0.8  5.2
extendrange(x, f= 0.01) # 0.96 5.04
## Use 'r' if you have it already:
stopifnot(identical(extendrange(r=r),
                    extendrange(x)))
```

`getGraphicsEvent` *Wait for a mouse or keyboard event from a graphics window*

Description

This function waits for input from a graphics window in the form of a mouse or keyboard event.

Usage

```
getGraphicsEvent(prompt = "Waiting for input",
                 onMouseDown = NULL, onMouseMove = NULL,
                 onMouseUp = NULL, onKeybd = NULL)
```

Arguments

`prompt`	prompt to be displayed to the user
`onMouseDown`	a function to respond to mouse clicks
`onMouseMove`	a function to respond to mouse movement
`onMouseUp`	a function to respond to mouse button releases
`onKeybd`	a function to respond to key presses

Details

This function allows user input from some graphics devices (currently only the Windows screen display). When called, event handlers may be installed to respond to events involving the mouse or keyboard.

The mouse event handlers should be functions with header `function(buttons, x, y)`. The coordinates `x` and `y` will be passed to mouse event handlers in device independent coordinates (i.e. the lower left corner of the window is `(0,0)`, the upper right is `(1,1)`). The `buttons` argument will be a vector listing the buttons that are pressed at the time of the event, with 0 for left, 1 for middle, and 2 for right.

The keyboard event handler should be a function with header `function(key)`. A single element character vector will be passed to this handler, corresponding to the key press. Shift and other modifier keys will have been processed, so `shift-a` will be passed as `"A"`. The following special keys may also be passed to the handler:

- Control keys, passed as `"Ctrl-A"`, etc.
- Navigation keys, passed as one of `"Left", "Up", "Right", "Down", "PgUp", "PgDn", "End", "Home"`
- Edit keys, passed as one of `"Ins", "Del"`
- Function keys, passed as one of `"F1", "F2", ...`

The event handlers are standard R functions, and will be executed in an environment as though they had been called directly from `getGraphicsEvent`.

Events will be processed until

- one of the event handlers returns a non-`NULL` value which will be returned as the value of `getGraphicsEvent`, or
- the user interrupts the function from the console.

Value

A non-`NULL` value returned from one of the event handlers.

Author(s)

Duncan Murdoch

Examples

```
## Not run:
    mousedown <- function(buttons, x, y) {
        plx <- grconvertX(x, "ndc", "user")
        ply <- grconvertY(y, "ndc", "user")
        cat("Buttons ", paste(buttons, collapse=" "), " at ndc",
            x, y, "user", plx, ply, "\n")
        points(plx, ply, col="red", pch=19, cex=2)
        if (x > 0.85 && y > 0.85) "Done"
        else NULL
    }

    mousemove <- function(buttons, x, y) {
        plx <- grconvertX(x, "ndc", "user")
        ply <- grconvertY(y, "ndc", "user")
        points(plx, ply)
```

```
        NULL
    }

    keybd <- function(key) {
        cat("Key <", key, ">\n", sep = "")
    }

    plot(0:1, 0:1, type='n')
    getGraphicsEvent("Click on upper right to quit",
                     onMouseDown = mousedown,
                     onMouseMove = mousemove,
                     onKeybd = keybd)
## End(Not run)
```

`gray` *Gray Level Specification*

Description

Create a vector of colors from a vector of gray levels.

Usage

```
gray(level)
grey(level)
```

Arguments

`level`	a vector of desired gray levels between `0` and `1`; zero indicates `"black"` and one indicates `"white"`.

Details

The values returned by `gray` can be used with a `col=` specification in graphics functions or in `par`.

`grey` is an alias for `gray`.

Value

A vector of colors of the same length as `level`.

See Also

`rainbow`, `hsv`, `hcl`, `rgb`.

Examples

```
gray(0:8 / 8)
```

`gray.colors` *Gray Color Palette*

Description

Create a vector of `n` gamma-corrected gray colors.

Usage

```
gray.colors(n, start = 0.3, end = 0.9, gamma = 2.2)
grey.colors(n, start = 0.3, end = 0.9, gamma = 2.2)
```

Arguments

`n`	the number of gray colors (≥ 1) to be in the palette.
`start`	starting gray level in the palette (should be between `0` and `1` where zero indicates `"black"` and one indicates `"white"`).
`end`	ending gray level in the palette.
`gamma`	the gamma correction.

Details

The function `gray.colors` chooses a series of `n` gamma-corrected gray levels between `start` and `end`: $(start^\gamma, \ldots, end^\gamma)^{(1/\gamma)}$. The returned palette contains the corresponding gray colors. This palette is used in `barplot.default`.

`grey.colors` is an alias for `gray.colors`.

Value

A vector of `n` gray colors.

See Also

`gray`, `rainbow`, `palette`.

Examples

```
require(graphics)

pie(rep(1,12), col = gray.colors(12))
barplot(1:12, col = gray.colors(12))
```

`hcl` *HCL Color Specification*

Description

Create a vector of colors from vectors specifying hue, chroma and luminance.

Usage

```
hcl(h = 0, c = 35, l = 85, alpha, fixup = TRUE)
```

Arguments

`h`	The hue of the color specified as an angle in the range [0,360]. 0 yields red, 120 yields green 240 yields blue, etc.
`c`	The chroma of the color. The upper bound for chroma depends on hue and luminance.
`l`	A value in the range [0,100] giving the luminance of the colour. For a given combination of hue and chroma, only a subset of this range is possible.
`alpha`	numeric vector of values in the range `[0,1]` for alpha transparency channel (0 means transparent and 1 means opaque).
`fixup`	a logical value which indicates whether the resulting RGB values should be corrected to ensure that a real color results. if `fixup` is `FALSE` RGB components lying outside the range [0,1] will result in an `NA` value.

Details

This function corresponds to polar coordinates in the CIE-LUV color space. Steps of equal size in this space correspond to approximately equal perceptual changes in color. Thus, `hcl` can be thought of as a perceptually based version of `hsv`.

The function is primarily intended as a way of computing colors for filling areas in plots where area corresponds to a numerical value (pie charts, bar charts, mosaic plots, histograms, etc). Choosing colors which have equal chroma and luminance provides a way of minimising the irradiation illusion which would otherwise produce a misleading impression of how large the areas are.

The default values of chroma and luminance make it possible to generate a full range of hues and have a relatively pleasant pastel appearance.

The RGB values produced by this function correspond to the sRGB color space used on most PC computer displays. There are other packages which provide more general color space facilities.

Semi-transparent colors (`0 < alpha < 1`) are supported only on some devices: see `rgb`.

Value

A vector of character strings which can be used as color specifications by R graphics functions.

Note

At present there is no guarantee that the colours rendered by R graphics devices will correspond to their sRGB description. It is planned to adopt sRGB as the standard R color description in future.

Author(s)

Ross Ihaka

References

Ihaka, R. (2003). Colour for Presentation Graphics, Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003), March 20-22, 2003, Technische Universität Wien, Vienna, Austria. http://www.ci.tuwien.ac.at/Conferences/DSC-2003.

See Also

hsv, rgb.

Examples

```
require(graphics)

# The Foley and Van Dam PhD Data.
csd <- matrix(c( 4,2,4,6, 4,3,1,4, 4,7,7,1,
                 0,7,3,2, 4,5,3,2, 5,4,2,2,
                 3,1,3,0, 4,4,6,7, 1,10,8,7,
                 1,5,3,2, 1,5,2,1, 4,1,4,3,
                 0,3,0,6, 2,1,5,5), nrow=4)

csphd <- function(colors)
  barplot(csd, col = colors, ylim = c(0,30),
          names = 72:85, xlab = "Year", ylab = "Students",
          legend = c("Winter", "Spring", "Summer", "Fall"),
          main = "Computer Science PhD Graduates", las = 1)

# The Original (Metaphorical) Colors (Ouch!)
csphd(c("blue", "green", "yellow", "orange"))

# A Color Tetrad (Maximal Color Differences)
csphd(hcl(h = c(30, 120, 210, 300)))

# Same, but lighter and less colorful
# Turn of automatic correction to make sure
# that we have defined real colors.
csphd(hcl(h = c(30, 120, 210, 300),
          c = 20, l = 90, fixup = FALSE))

# Analogous Colors
# Good for those with red/green color confusion
csphd(hcl(h = seq(60, 240, by = 60)))

# Metaphorical Colors
csphd(hcl(h = seq(210, 60, length = 4)))

# Cool Colors
csphd(hcl(h = seq(120, 0, length = 4) + 150))

# Warm Colors
csphd(hcl(h = seq(120, 0, length = 4) - 30))

# Single Color
```

```
hist(stats::rnorm(1000), col = hcl(240))
```

`Hershey` *Hershey Vector Fonts in R*

Description

If the `family` graphical parameter (see `par`) has been set to one of the Hershey fonts (see 'Details') Hershey vector fonts are used to render text.

When using the `text` and `contour` functions Hershey fonts may be selected via the `vfont` argument, which is a character vector of length 2 (see 'Details' for valid values). This allows Cyrillic to be selected, which is not available via the font families.

Usage

```
Hershey
```

Details

The Hershey fonts have two advantages:

1. vector fonts describe each character in terms of a set of points; R renders the character by joining up the points with straight lines. This intimate knowledge of the outline of each character means that R can arbitrarily transform the characters, which can mean that the vector fonts look better for rotated text.
2. this implementation was adapted from the GNU libplot library which provides support for non-ASCII and non-English fonts. This means that it is possible, for example, to produce weird plotting symbols and Japanese characters.

Drawback:
You cannot use mathematical expressions (`plotmath`) with Hershey fonts.

The Hershey characters are organised into a set of fonts. A particular font is selected by specifying one of the following font families via `par(family)` and specifying the desired font face (plain, bold, italic, bold-italic) via `par(font)`.

family	faces available
"HersheySerif"	plain, bold, italic, bold-italic
"HersheySans"	plain, bold, italic, bold-italic
"HersheyScript"	plain, bold
"HersheyGothicEnglish"	plain
"HersheyGothicGerman"	plain
"HersheyGothicItalian"	plain
"HersheySymbol"	plain, bold, italic, bold-italic
"HersheySansSymbol"	plain, italic

In the `vfont` specification for the `text` and `contour` functions, the Hershey font is specified by a typeface (e.g., `serif` or `sans serif`) and a fontindex or 'style' (e.g., `plain` or `italic`). The first element of `vfont` specifies the typeface and the second element specifies the fontindex. The first table produced by `demo(Hershey)` shows the character `a` produced by each of the different fonts.

The available `typeface` and `fontindex` values are available as list components of the variable `Hershey`. The allowed pairs for `(typeface, fontindex)` are:

serif	plain
serif	italic
serif	bold
serif	bold italic
serif	cyrillic
serif	oblique cyrillic
serif	EUC
sans serif	plain
sans serif	italic
sans serif	bold
sans serif	bold italic
script	plain
script	italic
script	bold
gothic english	plain
gothic german	plain
gothic italian	plain
serif symbol	plain
serif symbol	italic
serif symbol	bold
serif symbol	bold italic
sans serif symbol	plain
sans serif symbol	italic

and the indices of these are available as `Hershey$allowed`.

Escape sequences: The string to be drawn can include escape sequences, which all begin with a `\`. When R encounters a `\`, rather than drawing the `\`, it treats the subsequent character(s) as a coded description of what to draw.

One useful escape sequence (in the current context) is of the form: `\123`. The three digits following the `\` specify an octal code for a character. For example, the octal code for `p` is 160 so the strings `"p"` and `"\160"` are equivalent. This is useful for producing characters when there is not an appropriate key on your keyboard.

The other useful escape sequences all begin with `\\`. These are described below. Remember that backslashes have to be doubled in R character strings, so they need to be entered with *four* backslashes.

Symbols: an entire string of Greek symbols can be produced by selecting the HersheySymbol or HersheySansSymbol family or the Serif Symbol or Sans Serif Symbol typeface. To allow Greek symbols to be embedded in a string which uses a non-symbol typeface, there are a set of symbol escape sequences of the form `\\ab`. For example, the escape sequence `\\*a` produces a Greek alpha. The second table in `demo(Hershey)` shows all of the symbol escape sequences and the symbols that they produce.

ISO Latin-1: further escape sequences of the form `\\ab` are provided for producing ISO Latin-1 characters. Another option is to use the appropriate octal code. The (non-ASCII) ISO Latin-1 characters are in the range 241...377. For example, `\366` produces the character `o` with an umlaut. The third table in `demo(Hershey)` shows all of the ISO Latin-1 escape sequences.

These characters can be used directly in a Latin-1 locale or on a system with MBCS support. (In the latter, characters not in Latin-1 are replaced by a dot.)

Several characters are missing, c-cedilla has no cedilla and 'sharp s' (`U+00DF`, also known as 'esszett') is rendered as `ss`.

Special Characters: a set of characters are provided which do not fall into any standard font. These can only be accessed by escape sequence. For example, `\\LI` produces the zodiac sign for Libra, and `\\JU` produces the astronomical sign for Jupiter. The fourth table in `demo(Hershey)` shows all of the special character escape sequences.

Cyrillic Characters: cyrillic characters are implemented according to the K018-R encoding, and can be used directly in such a locale using the Serif typeface and Cyrillic (or Oblique Cyrillic) fontindex. Alternatively they can be specified via an octal code in the range 300 to 337 for lower case characters or 340 to 377 for upper case characters. The fifth table in `demo(Hershey)` shows the octal codes for the available Cyrillic characters.

Cyrillic has to be selected via a `("serif", fontindex)` pair rather than via a font family.

Japanese Characters: 83 Hiragana, 86 Katakana, and 603 Kanji characters are implemented according to the EUC-JP (Extended Unix Code) encoding. Each character is identified by a unique hexadecimal code. The Hiragana characters are in the range 0x2421 to 0x2473, Katakana are in the range 0x2521 to 0x2576, and Kanji are (scattered about) in the range 0x3021 to 0x6d55.

When using the Serif typeface and EUC fontindex, these characters can be produced by a *pair* of octal codes. Given the hexadecimal code (e.g., 0x2421), take the first two digits and add 0x80 and do the same to the second two digits (e.g., 0x21 and 0x24 become 0xa4 and 0xa1), then convert both to octal (e.g., 0xa4 and 0xa1 become 244 and 241). For example, the first Hiragana character is produced by `\244\241`.

It is also possible to use the hexadecimal code directly. This works for all non-EUC fonts by specifying an escape sequence of the form `\\#J1234`. For example, the first Hiragana character is produced by `\\#J2421`.

The Kanji characters may be specified in a third way, using the so-called "Nelson Index", by specifying an escape sequence of the form `\\#N1234`. For example, the (obsolete) Kanji for 'one' is produced by `\\#N0001`.

`demo(Japanese)` shows the available Japanese characters.

Raw Hershey Glyphs: all of the characters in the Hershey fonts are stored in a large array. Some characters are not accessible in any of the Hershey fonts. These characters can only be accessed via an escape sequence of the form `\\#H1234`. For example, the fleur-de-lys is produced by `\\#H0746`. The sixth and seventh tables of `demo(Hershey)` shows all of the available raw glyphs.

References

http://www.gnu.org/software/plotutils/plotutils.html

See Also

`demo(Hershey)`, `par`, `text`, `contour`.

`Japanese` for the Japanese characters in the Hershey fonts.

Examples

```
Hershey

## for tables of examples, see demo(Hershey)
```

`hsv` *HSV Color Specification*

Description

Create a vector of colors from vectors specifying hue, saturation and value.

Usage

```
hsv(h = 1, s = 1, v = 1, gamma = 1, alpha)
```

Arguments

`h,s,v`	numeric vectors of values in the range `[0,1]` for 'hue', 'saturation' and 'value' to be combined to form a vector of colors. Values in shorter arguments are recycled.
`gamma`	a gamma-correction exponent, γ
`alpha`	numeric vector of values in the range `[0,1]` for alpha transparency channel (0 means transparent and 1 means opaque).

Details

Semi-transparent colors (`0 < alpha < 1`) are supported only on some devices: see `rgb`.

Value

This function creates a vector of colors corresponding to the given values in HSV space. The values returned by `hsv` can be used with a `col=` specification in graphics functions or in `par`.

Gamma correction

For each color, (r, g, b) in RGB space (with all values in $[0, 1]$), the final color corresponds to $(r^\gamma, g^\gamma, b^\gamma)$.

See Also

`hcl` for a perceptually based version of `hsv()`, `rgb` and `rgb2hsv` for RGB to HSV conversion; `rainbow`, `gray`.

Examples

```
require(graphics)

hsv(.5,.5,.5)

## Look at gamma effect:
n <- 20;  y <- -sin(3*pi*((1:n)-1/2)/n)
op <- par(mfrow=c(3,2),mar=rep(1.5,4))
for(gamma in c(.4, .6, .8, 1, 1.2, 1.5))
  plot(y, axes = FALSE, frame.plot = TRUE,
       xlab = "", ylab = "", pch = 21, cex = 30,
       bg = rainbow(n, start=.85, end=.1, gamma = gamma),
```

```
        main = paste("Red tones;  gamma=",format(gamma)))
par(op)
```

`Japanese` *Japanese characters in R*

Description

The implementation of Hershey vector fonts provides a large number of Japanese characters (Hiragana, Katakana, and Kanji).

Details

Without keyboard support for typing Japanese characters, the only way to produce these characters is to use special escape sequences: see `Hershey`.

For example, the Hiragana character for the sound "ka" is produced by `\\#J242b` and the Katakana character for this sound is produced by `\\#J252b`. The Kanji ideograph for "one" is produced by `\\#J306c` or `\\#N0001`.

The output from `demo(Japanese)` shows tables of the escape sequences for the available Japanese characters.

References

http://www.gnu.org/software/plotutils/plotutils.html

See Also

`demo(Japanese)`, `Hershey`, `text`

Examples

```
require(graphics)

plot(1:9, type="n", axes=FALSE, frame=TRUE, ylab="",
     main= "example(Japanese)", xlab= "using Hershey fonts")
par(cex=3)
Vf <- c("serif", "plain")
text(4, 2, "\\#J2438\\#J2421\\#J2451\\#J2473", vfont = Vf)
text(4, 4, "\\#J2538\\#J2521\\#J2551\\#J2573", vfont = Vf)
text(4, 6, "\\#J467c\\#J4b5c", vfont = Vf)
text(4, 8, "Japan", vfont = Vf)
par(cex=1)
text(8, 2, "Hiragana")
text(8, 4, "Katakana")
text(8, 6, "Kanji")
text(8, 8, "English")
```

`make.rgb` *Create colour spaces*

Description

These functions specify colour spaces for use in `convertColor`.

Usage

```
make.rgb(red, green, blue, name = NULL, white = "D65",
         gamma = 2.2)

colorConverter(toXYZ, fromXYZ, name, white=NULL)
```

Arguments

`red,green,blue`	Chromaticity (xy or xyY) of RGB primaries
`name`	Name for the colour space
`white`	Character string specifying the reference white (see 'Details'.)
`gamma`	Display gamma (nonlinearity). A positive number or the string `"sRGB"`
`fromXYZ`	Function to convert from XYZ tristimulus coordinates to this space
`toXYZ`	Function to convert from this space to XYZ tristimulus coordinates.

Details

An RGB colour space is defined by the chromaticities of the red, green and blue primaries. These are given as vectors of length 2 or 3 in xyY coordinates (the Y component is not used and may be omitted). The chromaticities are defined relative to a reference white, which must be one of the CIE standard illuminants: "A", "B", "C", "D50", "D55", "D60", "E" (usually "D65").

The display gamma is most commonly 2.2, though 1.8 is used for Apple RGB. The sRGB standard specifies a more complicated function that is close to a gamma of 2.2; `gamma="sRGB"` uses this function.

Colour spaces other than RGB can be specified directly by giving conversions to and from XYZ tristimulus coordinates. The functions should take two arguments. The first is a vector giving the coordinates for one colour. The second argument is the reference white. If a specific reference white is included in the definition of the colour space (as for the RGB spaces) this second argument should be ignored and may be `...`.

Value

An object of class `colorConverter`

References

Conversion algorithms from `http://www.brucelindbloom.com`

See Also

`convertColor`

Examples

```
(pal <- make.rgb(red=  c(0.6400,0.3300),
                 green=c(0.2900,0.6000),
                 blue= c(0.1500,0.0600),
                 name = "PAL/SECAM RGB"))

## converter for sRGB in #rrggbb format
hexcolor <- colorConverter(toXYZ = function(hex,...) {
                            rgb <- t(col2rgb(hex))/255
                            colorspaces$sRGB$toXYZ(rgb,...) },
                           fromXYZ = function(xyz,...) {
                              rgb <- colorspaces$sRGB$fromXYZ(xyz,..)
                              rgb <- round(rgb,5)
                              if (min(rgb) < 0 || max(rgb) > 1)
                                   as.character(NA)
                              else
                                   rgb(rgb[1],rgb[2],rgb[3])},
                           white = "D65", name = "#rrggbb")

(cols <- t(col2rgb(palette())))
(luv <- convertColor(cols,from="sRGB", to="Luv", scale.in=255))
(hex <- convertColor(luv, from="Luv",  to=hexcolor, scale.out=NULL))

## must make hex a matrix before using it
(cc <- round(convertColor(as.matrix(hex), from= hexcolor, to= "sRGB",
                          scale.in=NULL, scale.out=255)))
stopifnot(cc == cols)
```

---

n2mfrow *Compute Default mfrow From Number of Plots*

Description

Easy setup for plotting multiple figures (in a rectangular layout) on one page. This computes a sensible default for `par(mfrow)`.

Usage

```
n2mfrow(nr.plots)
```

Arguments

`nr.plots`	integer; the number of plot figures you'll want to draw.

Value

A length two integer vector `nr, nc` giving the number of rows and columns, fulfilling `nr >= nc >= 1` and `nr * nc >= nr.plots`.

Author(s)

Martin Maechler

See Also

par, layout.

Examples

```
require(graphics)

n2mfrow(8) # 3 x 3

n <- 5 ; x <- seq(-2,2, len=51)
## suppose now that 'n' is not known {inside function}
op <- par(mfrow = n2mfrow(n))
for (j in 1:n)
   plot(x, x^j, main = substitute(x^ exp, list(exp = j)), type = "l",
   col = "blue")

sapply(1:10, n2mfrow)
```

---

nclass *Compute the Number of Classes for a Histogram*

Description

Compute the number of classes for a histogram.

Usage

```
nclass.Sturges(x)
nclass.scott(x)
nclass.FD(x)
```

Arguments

x	A data vector.

Details

`nclass.Sturges` uses Sturges' formula, implicitly basing bin sizes on the range of the data.

`nclass.scott` uses Scott's choice for a normal distribution based on the estimate of the standard error, unless that is zero where it returns `1`.

`nclass.FD` uses the Freedman-Diaconis choice based on the inter-quartile range (`IQR`) unless that's zero where it reverts to `mad(x, constant=2)` and when that is 0 as well, returns `1`.

Value

The suggested number of classes.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S-PLUS.* Springer, page 112.

Freedman, D. and Diaconis, P. (1981) On the histogram as a density estimator: L_2 theory. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete* **57**, 453–476.

Scott, D. W. (1979) On optimal and data-based histograms. *Biometrika* **66**, 605–610.

Scott, D. W. (1992) *Multivariate Density Estimation. Theory, Practice, and Visualization*. Wiley.

See Also

`hist` and `truehist` (which use a different default).

Examples

```
set.seed(1)
x <- stats::rnorm(1111)
nclass.Sturges(x)

## Compare them:
NC <- function(x) c(Sturges = nclass.Sturges(x),
      Scott = nclass.scott(x), FD = nclass.FD(x))
NC(x)
onePt <- rep(1, 11)
NC(onePt) # no longer gives NaN
```

---

palette *Set or View the Graphics Palette*

Description

View or manipulate the color palette which is used when a `col=` has a numeric index.

Usage

```
palette(value)
```

Arguments

`value`	an optional character vector.

Details

If `value` has length 1, it is taken to be the name of a built in color palette. If `value` has length greater than 1 it is assumed to contain a description of the colors which are to make up the new palette (either by name or by RGB levels).

If `value` is omitted or has length 0, no change is made the current palette.

Currently, the only built-in palette is `"default"`.

Value

The palette which *was* in effect. This is `invisible` unless the argument is omitted.

See Also

colors for the vector of built-in named colors; hsv, gray, rainbow, terrain.colors,... to construct colors.

colorRamp to interpolate colors, making custom palettes; col2rgb for translating colors to RGB 3-vectors.

Examples

```
require(graphics)

palette()               # obtain the current palette
palette(rainbow(6))     # six color rainbow

(palette(gray(seq(0,.9,len=25)))) # gray scales; print old palette
matplot(outer(1:100,1:30), type='l', lty=1,lwd=2, col=1:30,
        main = "Gray Scales Palette",
        sub = "palette(gray(seq(0,.9,len=25)))")
palette("default")      # reset back to the default
```

Palettes *Color Palettes*

Description

Create a vector of n contiguous colors.

Usage

```
rainbow(n, s = 1, v = 1, start = 0, end = max(1,n - 1)/n,
        gamma = 1, alpha = 1)
heat.colors(n, alpha = 1)
terrain.colors(n, alpha = 1)
topo.colors(n, alpha = 1)
cm.colors(n, alpha = 1)
```

Arguments

n	the number of colors (≥ 1) to be in the palette.
s,v	the 'saturation' and 'value' to be used to complete the HSV color descriptions.
start	the (corrected) hue in [0,1] at which the rainbow begins.
end	the (corrected) hue in [0,1] at which the rainbow ends.
gamma	the gamma correction, see argument gamma in hsv.
alpha	the alpha transparency, a number in [0,1], see argument alpha in hsv.

Details

Conceptually, all of these functions actually use (parts of) a line cut out of the 3-dimensional color space, parametrized by `hsv(h,s,v, gamma)`, where `gamma`$= 1$ for the *foo*`.colors` function, and hence, equispaced hues in RGB space tend to cluster at the red, green and blue primaries.

Some applications such as contouring require a palette of colors which do not wrap around to give a final color close to the starting one.

With `rainbow`, the parameters `start` and `end` can be used to specify particular subranges of hues. The following values can be used when generating such a subrange: red=0, yellow=$\frac{1}{6}$, green=$\frac{2}{6}$, cyan=$\frac{3}{6}$, blue=$\frac{4}{6}$ and magenta=$\frac{5}{6}$.

Value

A character vector, `cv`, of color names. This can be used either to create a user–defined color palette for subsequent graphics by `palette(cv)`, a `col=` specification in graphics functions or in `par`.

See Also

`colors`, `palette`, `hsv`, `hcl`, `rgb`, `gray` and `col2rgb` for translating to RGB numbers.

Examples

```
require(graphics)
# A Color Wheel
pie(rep(1,12), col=rainbow(12))

##------ Some palettes ------------
demo.pal <-
  function(n, border = if (n<32) "light gray" else NA,
           main = paste("color palettes;  n=",n),
           ch.col = c("rainbow(n, start=.7, end=.1)", "heat.colors(n)",
                      "terrain.colors(n)", "topo.colors(n)",
                      "cm.colors(n)"))
{
    nt <- length(ch.col)
    i <- 1:n; j <- n / nt; d <- j/6; dy <- 2*d
    plot(i,i+d, type="n", yaxt="n", ylab="", main=main)
    for (k in 1:nt) {
        rect(i-.5, (k-1)*j+ dy, i+.4, k*j,
             col = eval(parse(text=ch.col[k])), border = border)
        text(2*j,  k * j +dy/4, ch.col[k])
    }
}
n <- if(.Device == "postscript") 64 else 16
     # Since for screen, larger n may give color allocation problem
demo.pal(n)
```

`pdf` *PDF Graphics Device*

Description

`pdf` starts the graphics device driver for producing PDF graphics.

Usage

```
pdf(file = ifelse(onefile, "Rplots.pdf", "Rplot%03d.pdf"),
    width, height, onefile, family, title, fonts, version,
    paper, encoding, bg, fg, pointsize, pagecentre, colormodel,
    useDingbats)
```

Arguments

file	a character string giving the name of the file. For use with onefile=FALSE give a C integer format such as "Rplot%03d.pdf" (the default in that case). (See postscript for further details.)
width, height	the width and height of the graphics region in inches. The default values are 7.
onefile	logical: if true (the default) allow multiple figures in one file. If false, generate a file with name containing the page number for each page. Defaults to TRUE.
family	the font family to be used, see postscript. Defaults to "Helvetica".
title	title string to embed as the /Title field in the file. Defaults to "R Graphics Output".
fonts	a character vector specifying R graphics font family names for fonts which will be included in the PDF file. Defaults to NULL.
version	a string describing the PDF version that will be required to view the output. This is a minimum, and will be increased (with a warning) if necessary. Defaults to "1.4", but see 'Details'.
paper	the target paper size. The choices are "a4", "letter", "legal" (or "us") and "executive" (and these can be capitalized), or "a4r" and "USr" for rotated ('landscape'). The default is "special", which means that the width and height specify the paper size. A further choice is "default"; if this is selected, the papersize is taken from the option "papersize" if that is set and as "a4" if it is unset or empty. Defaults "special".
encoding	the name of an encoding file. See postscript for details. Defaults to "default".
bg	the initial background color to be used. Defaults to "transparent".
fg	the initial foreground color to be used. Defaults to "black".
pointsize	the default point size to be used. Strictly speaking, in bp, that is 1/72 of an inch, but approximately in points. Defaults to 12.
pagecentre	logical: should the device region be centred on the page? – is only relevant for paper != "special". Defaults to true.
colormodel	a character string describing the color model: currently allowed values are "rgb", "gray" and "cmyk". Defaults to "rgb".
useDingbats	logical. Should small circles be rendered *via* the Dingbats font? Defaults to true, which produces smaller and better output – this is provided for those who insist on using broken PDF viewers.

Details

All arguments except file default to values given by pdf.options(). The ultimate defaults are quoted in the arguments section.

pdf() opens the file file and the PDF commands needed to plot any graphics requested are sent to that file.

The file argument is interpreted as a C integer format as used by sprintf, with integer argument the page number. The default gives files 'Rplot001.pdf', ..., 'Rplot999.pdf', 'Rplot1000.pdf', ...

The family argument can be used to specify a PDF-specific font family as the initial/default font for the device.

If a device-independent R graphics font family is specified (e.g., via par(family=) in the graphics package), the PDF device makes use of the PostScript font mappings to convert the R graphics font family to a PDF-specific font family description. (See the documentation for pdfFonts.)

R does *not* embed fonts in the PDF file, so it is only straightforward to use mappings to the font families that can be assumed to be available in any PDF viewer: "Times" (equivalently "serif"), "Helvetica" (equivalently "sans") and "Courier" (equivalently "mono"). Other families may be specified, but it is the user's responsibility to ensure that these fonts are available on the system and third-party software, e.g., Ghostscript, may be required to embed the fonts so that the PDF can be included in other documents (e.g., LaTeX): see embedFonts. The URW-based families described for postscript can be used with viewers set up to use URW fonts, which is usual with those based on xpdf or Ghostscript. Since embedFonts makes use of Ghostscript, it should be able to embed the URW-based families for use with other viewers.

See postscript for details of encodings, as the internal code is shared between the drivers. The native PDF encoding is given in file 'PDFDoc.enc'.

pdf writes uncompressed PDF. It is primarily intended for producing PDF graphics for inclusion in other documents, and PDF-includers such as pdftex are usually able to handle compression.

The PDF produced is fairly simple, with each page being represented as a single stream. The R graphics model does not distinguish graphics objects at the level of the driver interface.

The version argument declares the version of PDF that gets produced. The version must be at least 1.4 for semi-transparent output to be understood, and at least 1.3 if CID fonts are to be used: if these features are used the version number will be increased (with a warning). Specifying a low version number is useful if you want to produce PDF output that can be viewed on older or non-Adobe PDF viewers. (PDF 1.4 requires Acrobat 5 or later.)

Line widths as controlled by par(lwd=) are in multiples of 1/96 inch. Multiples less than 1 are allowed. pch="." with cex = 1 corresponds to a square of side 1/72 inch, which is also the 'pixel' size assumed for graphics parameters such as "cra".

The paper argument sets the /MediaBox entry in the file, which defaults to width by height. If it is set to something other than "special", a device region of the specified size is (by default) centred on the rectangle given by the paper size: if either width or height is less than 0.1 or too large to give a total margin of 0.5 inch, it is reset to the corresponding paper dimension minus 0.5. Thus if you want the default behaviour of postscript use pdf(paper="a4r", width=0, height=0) to centre the device region on a landscape A4 page with 0.25 inch margins.

When the background colour is fully transparent (as is the initial default value), the PDF produced does not paint the background. Almost all PDF viewers will use a white canvas so the visual effect is if the background were white. This will not be the case when printing onto coloured paper, though.

Color models

The default color model is RGB, and model "gray" maps RGB colors to greyscale using perceived luminosity (biased towards green). "cmyk" outputs in CMYK colorspace. Nothing in R specifies the interpretation of the RGB or CMYK color spaces, and the simplest possible conversion to CMYK is used (http://en.wikipedia.org/wiki/CMYK_color_model#Mapping_RGB_to_CMYK).

Conventions

This section describes the implementation of the conventions for graphics devices set out in the "R Internals Manual".

- The default device size is 7 inches square.
- Font sizes are in big points.
- The default font family is Helvetica.
- Line widths are as a multiple of 1/96 inch, with no minimum.
- Circles of any radius are allowed. Unless `useDingbats = FALSE`, opaque circles of less than 10 big points radius are rendered using char 108 in the Dingbats font: all semi-transparent and larger circles using a Bézier curve for each quadrant.
- Colours are interpreted by the viewing/printing application.

Note

Acrobat Reader does not use the fonts specified but rather emulates them from multiple-master fonts. This can be seen in imprecise centering of characters, for example the multiply and divide signs in Helvetica. This can be circumvented by embedding fonts where possible. Most other viewers substitute fonts, e.g. URW fonts for the standard Helvetica and Times fonts, and these too often have different font metrics from the true fonts.

Acrobat Reader 5.x and later can be extended by support for Asian and (so-called) Central European fonts (the latter only for 7.x and later, part of the 'Extended' pack for 8.x and later), and these will be needed for the full use of encodings other than Latin-1. See http://www.adobe.com/products/acrobat/acrrasianfontpack.html for Reader 5.x to 8.x, and Updates on http://www.adobe.com for 9.x.

See Also

pdfFonts, pdf.options, embedFonts, Devices, postscript. cairo_pdf and (on Mac OS X only) quartz for other devices that can produce PDF.

More details of font families and encodings and especially handling text in a non-Latin-1 encoding and embedding fonts can be found in

Paul Murrell and Brian Ripley (2006) Non-standard fonts in PostScript and PDF graphics. *R News*, 6(2):41–47. http://cran.r-project.org/doc/Rnews/Rnews_2006-2.pdf.

Examples

```
## Not run:
## Test function for encodings
TestChars <- function(encoding="ISOLatin1", ...)
{
    pdf(encoding=encoding, ...)
    par(pty="s")
    plot(c(-1,16), c(-1,16), type="n", xlab="", ylab="",
         xaxs="i", yaxs="i")
         title(paste("Centred chars in encoding", encoding))
    grid(17, 17, lty=1)
    for(i in c(32:255)) {
        x <- i %% 16
        y <- i %/% 16
        points(x, y, pch=i)
```

```
    }
    dev.off()
}
## there will be many warnings.
TestChars("ISOLatin2")
## this does not view properly in older viewers.
TestChars("ISOLatin2", family="URWHelvetica")
## works well for viewing in gs-based viewers, and often in xpdf.
## End(Not run)
```

`pdf.options` *Auxiliary Function to Set/View Defaults for Arguments of pdf*

Description

The auxiliary function `pdf.options` can be used to set or view (if called without arguments) the default values for some of the arguments to `pdf`.

`pdf.options` needs to be called before calling `pdf`, and the default values it sets can be overridden by supplying arguments to `pdf`.

Usage

```
pdf.options(..., reset = FALSE)
```

Arguments

`...`	arguments `width`, `height`, `onefile`, `family`, `title`, `fonts`, `paper`, `encoding`, `pointsize`, `bg`, `fg`, `pagecentre`, `useDingbats` and `colormodel` can be supplied.
`reset`	logical: should the defaults be reset to their 'factory-fresh' values?

Details

If both `reset = TRUE` and `...` are supplied the defaults are first reset to the 'factory-fresh' values and then the new values are applied.

Value

A named list of all the defaults. If any arguments are supplied the return values are the old values and the result has the visibility flag turned off.

See Also

`pdf`, `ps.options`.

Examples

```
pdf.options(bg = "pink")
utils::str(pdf.options())
pdf.options(reset = TRUE) # back to factory-fresh
```

`pictex` *A PicTeX Graphics Driver*

Description

This function produces graphics suitable for inclusion in TeX and LaTeX documents.

Usage

```
pictex(file = "Rplots.tex", width = 5, height = 4, debug = FALSE,
       bg = "white", fg = "black")
```

Arguments

`file`	the file where output will appear.
`width`	The width of the plot in inches.
`height`	the height of the plot in inches.
`debug`	should debugging information be printed.
`bg`	the background color for the plot. Ignored.
`fg`	the foreground color for the plot. Ignored.

Details

This driver does not have any font metric information, so the use of `plotmath` is not supported.

Multiple plots will be placed as separate environments in the output file.

Line widths are ignored except when setting the spacing of line textures. `pch="."` corresponds to a square of side 1pt.

This device does not support colour (nor does the PicTeX package), and all colour settings are ignored.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the "R Internals Manual".

- The default device size is 5 inches by 5 inches.
- There is no `pointsize` argument: the default size is interpreted as 10 point.
- The only font family is `cmss10`.
- Line widths are only used when setting the spacing on line textures.
- Circle of any radius are allowed.
- Colour is not supported.

Author(s)

This driver was provided by Valerio Aimale ⟨valerio@svpop.com.dist.unige.it⟩ of the Department of Internal Medicine, University of Genoa, Italy.

References

Knuth, D. E. (1984) *The TeXbook.* Reading, MA: Addison-Wesley.

Lamport, L. (1994) *LATEX: A Document Preparation System.* Reading, MA: Addison-Wesley.

Goossens, M., Mittelbach, F. and Samarin, A. (1994) *The LATEX Companion.* Reading, MA: Addison-Wesley.

See Also

`postscript`, `Devices`.

Examples

```
require(graphics)

pictex()
plot(1:11,(-5:5)^2, type='b', main="Simple Example Plot")
dev.off()
##--------------------
## Not run:
%% LaTeX Example
\documentclass{article}
\usepackage{pictex}
\begin{document}
%...
\begin{figure}[h]
  \centerline{\input{Rplots.tex}}
  \caption{}
\end{figure}
%...
\end{document}

%%-- plain TeX Example --
\input pictex
$$ \input Rplots.tex $$
## End(Not run)
##--------------------
unlink("Rplots.tex")
```

`plotmath` *Mathematical Annotation in R*

Description

If the `text` argument to one of the text-drawing functions (`text`, `mtext`, `axis`, `legend`) in R is an expression, the argument is interpreted as a mathematical expression and the output will be formatted according to TeX-like rules. Expressions can also be used for titles, subtitles and x- and y-axis labels (but not for axis labels on `persp` plots).

In most cases other language objects (names and calls, including formulas) are coerced to expressions and so can also be used.

Details

A mathematical expression must obey the normal rules of syntax for any R expression, but it is interpreted according to very different rules than for normal R expressions.

It is possible to produce many different mathematical symbols, generate sub- or superscripts, produce fractions, etc.

The output from `demo(plotmath)` includes several tables which show the available features. In these tables, the columns of grey text show sample R expressions, and the columns of black text show the resulting output.

The available features are also described in the tables below:

Syntax	Meaning
`x + y`	x plus y
`x - y`	x minus y
`x*y`	juxtapose x and y
`x/y`	x forwardslash y
`x %+-% y`	x plus or minus y
`x %/% y`	x divided by y
`x %*% y`	x times y
`x %.% y`	x cdot y
`x[i]`	x subscript i
`x^2`	x superscript 2
`paste(x, y, z)`	juxtapose x, y, and z
`sqrt(x)`	square root of x
`sqrt(x, y)`	yth root of x
`x == y`	x equals y
`x != y`	x is not equal to y
`x < y`	x is less than y
`x <= y`	x is less than or equal to y
`x > y`	x is greater than y
`x >= y`	x is greater than or equal to y
`x %~~% y`	x is approximately equal to y
`x %=~% y`	x and y are congruent
`x %==% y`	x is defined as y
`x %prop% y`	x is proportional to y
`plain(x)`	draw x in normal font
`bold(x)`	draw x in bold font
`italic(x)`	draw x in italic font
`bolditalic(x)`	draw x in bolditalic font
`symbol(x)`	draw x in symbol font
`list(x, y, z)`	comma-separated list
`...`	ellipsis (height varies)
`cdots`	ellipsis (vertically centred)
`ldots`	ellipsis (at baseline)
`x %subset% y`	x is a proper subset of y
`x %subseteq% y`	x is a subset of y
`x %notsubset% y`	x is not a subset of y
`x %supset% y`	x is a proper superset of y
`x %supseteq% y`	x is a superset of y
`x %in% y`	x is an element of y
`x %notin% y`	x is not an element of y
`hat(x)`	x with a circumflex

`tilde(x)`	x with a tilde
`dot(x)`	x with a dot
`ring(x)`	x with a ring
`bar(xy)`	xy with bar
`widehat(xy)`	xy with a wide circumflex
`widetilde(xy)`	xy with a wide tilde
`x %<->% y`	x double-arrow y
`x %->% y`	x right-arrow y
`x %<-% y`	x left-arrow y
`x %up% y`	x up-arrow y
`x %down% y`	x down-arrow y
`x %<=>% y`	x is equivalent to y
`x %=>% y`	x implies y
`x %<=% y`	y implies x
`x %dblup% y`	x double-up-arrow y
`x %dbldown% y`	x double-down-arrow y
`alpha – omega`	Greek symbols
`Alpha – Omega`	uppercase Greek symbols
`theta1, phi1, sigma1, omega1`	cursive Greek symbols
`Upsilon1`	capital upsilon with hook
`aleph`	first letter of Hebrew alphabet
`infinity`	infinity symbol
`partialdiff`	partial differential symbol
`nabla`	nabla, gradient symbol
`32*degree`	32 degrees
`60*minute`	60 minutes of angle
`30*second`	30 seconds of angle
`displaystyle(x)`	draw x in normal size (extra spacing)
`textstyle(x)`	draw x in normal size
`scriptstyle(x)`	draw x in small size
`scriptscriptstyle(x)`	draw x in very small size
`underline(x)`	draw x underlined
`x ~~ y`	put extra space between x and y
`x + phantom(0) + y`	leave gap for "0", but don't draw it
`x + over(1, phantom(0))`	leave vertical gap for "0" (don't draw)
`frac(x, y)`	x over y
`over(x, y)`	x over y
`atop(x, y)`	x over y (no horizontal bar)
`sum(x[i], i==1, n)`	sum x[i] for i equals 1 to n
`prod(plain(P)(X==x), x)`	product of P(X=x) for all values of x
`integral(f(x)*dx, a, b)`	definite integral of f(x) wrt x
`union(A[i], i==1, n)`	union of A[i] for i equals 1 to n
`intersect(A[i], i==1, n)`	intersection of A[i]
`lim(f(x), x %->% 0)`	limit of f(x) as x tends to 0
`min(g(x), x > 0)`	minimum of g(x) for x greater than 0
`inf(S)`	infimum of S
`sup(S)`	supremum of S
`x^y + z`	normal operator precedence
`x^(y + z)`	visible grouping of operands
`x^{y + z}`	invisible grouping of operands
`group("(",list(a, b),"]")`	specify left and right delimiters
`bgroup("(",atop(x,y),")")`	use scalable delimiters

`group(lceil, x, rceil)`	special delimiters

The symbol font uses Adobe Symbol encoding so, for example, a lower case mu can be obtained either by the special symbol `mu` or by `symbol("m")`. This provides access to symbols that have no special symbol name, for example, the universal, or forall, symbol is `symbol("\042")`.

Note to TeX users: TeX's `\Upsilon` is `Upsilon1`, TeX's `\varepsilon` is close to `epsilon`, and there is no equivalent of TeX's `\epsilon`. TeX's `\varpi` is close to `omega1`. `vartheta`, `varphi` and `varsigma` are allowed as synonyms for `theta1`, `phi1` and `sigma1`.

`sigma1` is also known as `stigma`, its Unicode name.

Control characters (e.g. `\n`) are not interpreted in character strings in plotmath, unlike normal plotting.

The fonts used are taken from the current font family, and so can be set by `par(family=)` in base graphics, and `gpar(fontfamily=)` in package **grid**.

Other symbols

On many OSes and some graphics devices many other symbols are available as part of the standard text font, and all of the symbols in the Adobe Symbol encoding are in principle available *via* changing the font face or (see 'Details') plotmath: see the examples section of `points` for a function to display them. ('In principle' because some of the glyphs are missing from some implementations of the symbol font.) Unfortunately, `postscript` and `pdf` have support for little more than European and CJK characters and the Adobe Symbol encoding (and in a few fonts, also Cyrillic characters).

In a UTF-8 locale any Unicode character can be entered, perhaps as a `\uxxxx` or `\Uxxxxxxxx` escape sequence, but the issue is whether the graphics device is apply to display the character. The widest range of characters is likely to be available in the `X11` device using cairo: see its help page for how installing additional fonts can help.

In non-UTF-8 locales there is normally no support for symbols not in the languages for which the current encoding was intended.

References

Murrell, P. and Ihaka, R. (2000) An approach to providing mathematical annotation in plots. *Journal of Computational and Graphical Statistics*, **9**, 582–599.

The symbol codes can be found in octal in the Adobe reference manuals, e.g. for Postscript `http://www.adobe.com/products/postscript/pdfs/PLRM.pdf` or PDF `http://www.adobe.com/devnet/acrobat/pdfs/pdf_reference_1-7.pdf` and in decimal, octal and hex at `http://www.stat.auckland.ac.nz/~paul/R/CM/AdobeSym.html`.

See Also

`demo(plotmath)`, `axis`, `mtext`, `text`, `title`, `substitute` `quote`, `bquote`

Examples

```
require(graphics)

x <- seq(-4, 4, len = 101)
y <- cbind(sin(x), cos(x))
matplot(x, y, type = "l", xaxt = "n",
        main = expression(paste(plain(sin) * phi, "  and  ",
```

```
                                  plain(cos) * phi)),
        ylab = expression("sin" * phi, "cos" * phi), # only 1st is taken
        xlab = expression(paste("Phase Angle ", phi)),
        col.main = "blue")
axis(1, at = c(-pi, -pi/2, 0, pi/2, pi),
     labels = expression(-pi, -pi/2, 0, pi/2, pi))

## How to combine "math" and numeric variables :
plot(1:10, type="n", xlab="", ylab="", main = "plot math & numbers")
theta <- 1.23 ; mtext(bquote(hat(theta) == .(theta)))
for(i in 2:9)
    text(i,i+1, substitute(list(xi,eta) == group("(",list(x,y),")"),
                           list(x=i, y=i+1)))
## note that both of these use calls rather than expressions.

plot(1:10, 1:10)
text(4, 9, expression(hat(beta) == (X^t * X)^{-1} * X^t * y))
text(4, 8.4, "expression(hat(beta) == (X^t * X)^{-1} * X^t * y)",
     cex = .8)
text(4, 7, expression(bar(x) == sum(frac(x[i], n), i==1, n)))
text(4, 6.4, "expression(bar(x) == sum(frac(x[i], n), i==1, n))",
     cex = .8)
text(8, 5, expression(paste(frac(1, sigma*sqrt(2*pi)), " ",
                            plain(e)^{frac(-(x-mu)^2, 2*sigma^2)})),
     cex = 1.2)

## some other useful symbols
plot.new(); plot.window(c(0,4), c(15,1))
text(1, 1, "universal", adj=0); text(2.5, 1,  "\\042")
text(3, 1, expression(symbol("\042")))
text(1, 2, "existential", adj=0); text(2.5, 2,  "\\044")
text(3, 2, expression(symbol("\044")))
text(1, 3, "suchthat", adj=0); text(2.5, 3,  "\\047")
text(3, 3, expression(symbol("\047")))
text(1, 4, "therefore", adj=0); text(2.5, 4,  "\\134")
text(3, 4, expression(symbol("\134")))
text(1, 5, "perpendicular", adj=0); text(2.5, 5,  "\\136")
text(3, 5, expression(symbol("\136")))
text(1, 6, "circlemultiply", adj=0); text(2.5, 6,  "\\304")
text(3, 6, expression(symbol("\304")))
text(1, 7, "circleplus", adj=0); text(2.5, 7,  "\\305")
text(3, 7, expression(symbol("\305")))
text(1, 8, "emptyset", adj=0); text(2.5, 8,  "\\306")
text(3, 8, expression(symbol("\306")))
text(1, 9, "angle", adj=0); text(2.5, 9,  "\\320")
text(3, 9, expression(symbol("\320")))
text(1, 10, "leftangle", adj=0); text(2.5, 10,  "\\341")
text(3, 10, expression(symbol("\341")))
text(1, 11, "rightangle", adj=0); text(2.5, 11,  "\\361")
text(3, 11, expression(symbol("\361")))
```

Description

Graphics devices for JPEG, PNG or TIFF format bitmap files.

Usage

```
bmp(filename = "Rplot%03d.bmp",
    width = 480, height = 480, units = "px",
    pointsize = 12, bg = "white", res = NA, ...,
    type = c("cairo", "Xlib", "quartz"), antialias)

jpeg(filename = "Rplot%03d.jpeg",
     width = 480, height = 480, units = "px",
     pointsize = 12, quality = 75, bg = "white", res = NA, ...,
     type = c("cairo", "Xlib", "quartz"), antialias)

png(filename = "Rplot%03d.png",
    width = 480, height = 480, units = "px",
    pointsize = 12, bg = "white",  res = NA, ...,
    type = c("cairo", "Xlib", "cairo1", "quartz"), antialias)

tiff(filename = "Rplot%03d.tiff",
     width = 480, height = 480, units = "px", pointsize = 12,
     compression = c("none", "rle", "lzw", "jpeg", "zip"),
     bg = "white", res = NA,  ...,
     type = c("cairo", "Xlib", "quartz"), antialias)
```

Arguments

`filename`	the name of the output file. The page number is substituted if a C integer format is included in the character string, as in the default. (The result must be less than `PATH_MAX` characters long, and may be truncated if not. See `postscript` for further details.) Tilde expansion is performed where supported by the platform.
`width`	the width of the device.
`height`	the height of the device.
`units`	The units in which `height` and `width` are given. Can be `px` (pixels, the default), `in` (inches), `cm` or `mm`.
`pointsize`	the default pointsize of plotted text, interpreted as big points (1/72 inch) at `res` dpi.
`bg`	the initial background colour: can be overridden by setting par("bg").
`quality`	the 'quality' of the JPEG image, as a percentage. Smaller values will give more compression but also more degradation of the image.
`compression`	the type of compression to be used.
`res`	The nominal resolution in dpi which will be recorded in the bitmap file, if a positive integer. Also used for `units` other than the default, and to convert points to pixels.
`...`	for `type = "Xlib"` only, additional arguments to the underlying `X11` device, such as `gamma` and `fonts`.
`type`	character string, one of `"Xlib"` (the only type prior to R 2.7.0) or `"quartz"` (some Mac OS X builds) or `"cairo"` or `"cairo1"`. The latter two will

only be available if the system was compiled with support for cairo – otherwise "Xlib" will be used. The default is set by getOption("bitmapType") – the 'out of the box' default is "quartz" or "cairo" where available, otherwise "Xlib".

antialias for type = "cairo", giving the type of anti-aliasing (if any) to be used. See X11. The default is set by X11.options.

Details

Plots in PNG and JPEG format can easily be converted to many other bitmap formats, and both can be displayed in modern web browsers. The PNG format is lossless and is best for line diagrams and blocks of colour. The JPEG format is lossy, but may be useful for image plots, for example. BMP is a standard format on Windows. TIFF is a meta-format: the default format written by tiff is lossless and stores RGB (and alpha where appropriate) values uncompressed—such files are widely accepted, which is their main virtue over PNG.

png supports transparent backgrounds: use bg = "transparent". Not all PNG viewers render files with transparency correctly. When transparency is in use in the type = "Xlib" variant a very light grey is used as the background and so appear as transparent if used in the plot. This allows opaque white to be used, as in the example. The type = "cairo" and type = "cairo1" variants allows semi-transparent colours, including on a transparent or semi-transparent background.

tiff(type = "cairo") supports semi-transparent colours, including on a transparent or semi-transparent background.

R can be compiled without support for each of these devices: this will be reported if you attempt to use them on a system where they are not supported. For type = "Xlib" they may not be usable unless the X11 display is available to the owner of the R process. type = "cairo" requires cairo 1.2 or later. type = "quartz" uses the quartz device and so is only available where that is (on some Mac OS X) builds. It is at present experimental.

By default no resolution is recorded in the file. Viewers will often assume a nominal resolution of 72dpi when none is recorded. As resolutions in PNG files are recorded in pixels/metre, the reported dpi value will be changed slightly.

For graphics parameters that make use of dimensions in inches (including font sizes in points) the resolution used is res (or 72dpi if unset).

png will use a palette if there are less than 256 colours on the page, and record a 24-bit RGB file otherwise (or a 32-bit RGBA file if type = "cairo" and non-opaque colours are used).

png(type = "cairo1") uses cairo directly to output the file – the resolution is not recorded and the PNG format used is chosen by cairo (and undocumented – it is currently 32-bit RGBA, that is with an alpha channel whether used or not). It is included for use on platforms with cairo 1.0 (where type = "cairo" is not available).

Value

A plot device is opened: nothing is returned to the R interpreter.

Warnings

Note that by default the width and height are in pixels not inches. A warning will be issued if both are less than 20.

If you plot more than one page on one of these devices and do not include something like %d for the sequence number in file, the file will contain the last page plotted.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the "R Internals Manual".

- The default device size is in pixels.
- Font sizes are in big points interpreted at `res` dpi.
- The default font family is Helvetica.
- Line widths in 1/96 inch, minimum one pixel for `type = "Xlib"`, 0.01 for `type = "cairo"`.
- For `type = "Xlib"` circle radii are in pixels with minimum one.
- Colours are interpreted by the viewing application.

For `type = "quartz"` see the help for `quartz`.

Note

For `type = "Xlib"` these devices are based on the `X11` device. The colour model used will be that set up by `X11.options` at the time the first Xlib-based devices was opened (or the first after all such devices have been closed).

Author(s)

Guido Masarotto and Brian Ripley

See Also

`Devices`, `dev.print`

`capabilities` to see if these devices are supported by this build of R, and if `type = "cairo"` is supported.

`bitmap` provides an alternative way to generate PNG and JPEG plots that does not depend on accessing the X11 display but does depend on having GhostScript installed. (Devices `GDD` in CRAN package **GDD** and `CairoJPEG` / `CairoPNG` in CRAN package **Cairo** are further alternatives using several other additional pieces of software.)

Examples

```
## these examples will work only if the devices are available
## and either cairo or an X11 display is available.

## copy current plot to a (large) PNG file
## Not run: dev.print(png, file="myplot.png", width=1024, height=768)

png(file="myplot.png", bg="transparent")
plot(1:10)
rect(1, 5, 3, 7, col="white")
dev.off()

## will make myplot1.jpeg and myplot2.jpeg
jpeg(file="myplot%d.jpeg")
example(rect)
dev.off()
```

`postscript` *PostScript Graphics*

Description

`postscript` starts the graphics device driver for producing PostScript graphics.

Usage

```
postscript(file = ifelse(onefile, "Rplots.ps", "Rplot%03d.ps"),
           onefile, family, title, fonts, encoding, bg, fg,
           width, height, horizontal, pointsize,
           paper, pagecentre, print.it, command, colormodel)
```

Arguments

`file`	a character string giving the name of the file. If it is `""`, the output is piped to the command given by the argument `command`. If it is of the form `"|cmd"`, the output is piped to the command given by 'cmd'. For use with `onefile = FALSE` give a `printf` format such as `"Rplot%03d.ps"` (the default in that case). The string should not otherwise contain a `%`: if it is really necessary, use `%%` in the string for `%` in the file name. A single integer format matching the regular expression `"%[#0 +=-]*[0-9.]*[diouxX]"` is allowed.
`onefile`	logical: if true (the default) allow multiple figures in one file. If false, generate a file name containing the page number for each page and use an EPSF header and no `DocumentMedia` comment. Defaults to the `TRUE`.
`family`	the initial font family to be used, normally as a character string. See the section 'Families'. Defaults to `"Helvetica"`.
`title`	title string to embed as the `Title` comment in the file. Defaults to `"R Graphics Output"`.
`fonts`	a character vector specifying additional R graphics font family names for font families whose declarations will be included in the PostScript file and are available for use with the device. See 'Families' below. Defaults to `NULL`.
`encoding`	the name of an encoding file. Defaults to `"default"`. The latter is interpreted as '"ISOLatin1.enc"' unless the locale is recognized as corresponding to a language using ISO 8859-5,7,13,15 or KOI8-R,U. The file is looked forin the 'enc' directory of package **grDevices** if the path does not contain a path separator. An extension `".enc"` can be omitted.
`bg`	the initial background color to be used. If `"transparent"` (or any other non-opaque colour), no background is painted. Defaults to `"transparent"`.
`fg`	the initial foreground color to be used. Defaults to `"black"`.
`width, height`	the width and height of the graphics region in inches. Default to `t0`. If `paper != "special"` and `width` or `height` is less than `0.1` or too large to give a total margin of 0.5 inch, it is reset to the corresponding paper dimension minus 0.5.

horizontal	the orientation of the printed image, a logical. Defaults to true, that is landscape orientation on paper sizes with width less than height.
pointsize	the default point size to be used. Strictly speaking, in bp, that is 1/72 of an inch, but approximately in points. Defaults to 12.
paper	the size of paper in the printer. The choices are "a4", "letter" (or "us"), "legal" and "executive" (and these can be capitalized). Also, "special" can be used, when arguments width and height specify the paper size. A further choice is "default" (the default): If this is selected, the papersize is taken from the option "papersize" if that is set and to "a4" if it is unset or empty.
pagecentre	logical: should the device region be centred on the page? Defaults to true.
print.it	logical: should the file be printed when the device is closed? (This only applies if file is a real file name.) Defaults to false.
command	the command to be used for 'printing'. Defaults to "default", the value of option "printcmd". The length limit is 2*PATH_MAX, typically 8096 bytes.
colormodel	a character string describing the color model: currently allowed values as "rgb", "rgb-nogray", "gray" and "cmyk". Defaults to "rgb".

Details

All arguments except file default to values given by ps.options(). The ultimate defaults are quoted in the arguments section.

postscript opens the file file and the PostScript commands needed to plot any graphics requested are written to that file. This file can then be printed on a suitable device to obtain hard copy.

The file argument is interpreted as a C integer format as used by sprintf, with integer argument the page number. The default gives files 'Rplot001.ps', ..., 'Rplot999.ps', 'Rplot1000.ps', ...

The postscript produced for a single R plot is EPS (*Encapsulated PostScript*) compatible, and can be included into other documents, e.g., into LaTeX, using \includegraphics{<filename>}. For use in this way you will probably want to use setEPS() to set the defaults as horizontal = FALSE, onefile = FALSE, paper = "special". Note that the bounding box is for the *device* region: if you find the white space around the plot region excessive, reduce the margins of the figure region via par(mar=).

Most of the PostScript prologue used is taken from the R character vector .ps.prolog. This is marked in the output, and can be changed by changing that vector. (This is only advisable for PostScript experts: the standard version is in namespace:grDevices.)

A PostScript device has a default family, which can be set by the user via family. If other font families are to be used when drawing to the PostScript device, these must be declared when the device is created via fonts; the font family names for this argument are R graphics font family names (see the documentation for postscriptFonts).

Line widths as controlled by par(lwd=) are in multiples of 1/96 inch: multiples less than 1 are allowed. pch="." with cex = 1 corresponds to a square of side 1/72 inch, which is also the 'pixel' size assumed for graphics parameters such as "cra".

When the background colour is fully transparent (as is the initial default value), the PostScript produced does not paint the background. Almost all PostScript viewers will use a white canvas so the visual effect is if the background were white. This will not be the case when printing onto coloured paper, though.

Families

Font families are collections of fonts covering the five font faces, (conventionally plain, bold, italic, bold-italic and symbol) selected by the graphics parameter `par(font=)` or the grid parameter `gpar(fontface=)`. Font families can be specified either as an an initial/default font family for the device via the `family` argument or after the device is opened by the graphics parameter `par(family=)` or the grid parameter `gpar(fontfamily=)`. Families which will be used in addition to the initial family must be specified in the `fonts` argument when the device is opened.

Font families are declared via a call to `postscriptFonts`.

The argument `family` specifies the initial/default font family to be used. In normal use it is one of `"AvantGarde"`, `"Bookman"`, `"Courier"`, `"Helvetica"`, `"Helvetica-Narrow"`, `"NewCenturySchoolbook"`, `"Palatino"` or `"Times"`, and refers to the standard Adobe PostScript fonts families of those names which are included (or cloned) in all common PostScript devices.

Many PostScript emulators (including those based on `ghostscript`) use the URW equivalents of these fonts, which are `"URWGothic"`, `"URWBookman"`, `"NimbusMon"`, `"NimbusSan"`, `"NimbusSanCond"`, `"CenturySch"`, `"URWPalladio"` and `"NimbusRom"` respectively. If your PostScript device is using URW fonts, you will obtain access to more characters and more appropriate metrics by using these names. To make these easier to remember, `"URWHelvetica" == "NimbusSan"` and `"URWTimes" == "NimbusRom"` are also supported.

Another type of family makes use of CID-keyed fonts for East Asian languages – see `postscriptFonts`.

The `family` argument is normally a character string naming a font family, but family objects generated by `Type1Font` and `CIDFont` are also accepted. For compatibility with earlier versions of R, the initial family can also be specified as a vector of four or five afm files.

Note that R does not embed the font(s) used in the PostScript output: see `embedFonts` for a utility to help do so.

Encodings

Encodings describe which glyphs are used to display the character codes (in the range 0–255). Most commonly R uses ISOLatin1 encoding, and the examples for `text` are in that encoding. However, the encoding used on machines running R may well be different, and by using the `encoding` argument the glyphs can be matched to encoding in use. This suffices for European and Cyrillic languages, but not for CJK languages. For the latter, composite CID fonts are used. These fonts are useful for other languages: for example they may contain Greek glyphs. (The rest of this section applies only when CID fonts are not used.)

None of this will matter if only ASCII characters (codes 32–126) are used as all the encodings (except `"TeXtext"`) agree over that range. Some encodings are supersets of ISOLatin1, too. However, if accented and special characters do not come out as you expect, you may need to change the encoding. Some other encodings are supplied with R: `"WinAnsi.enc"` and `"MacRoman.enc"` correspond to the encodings normally used on Windows and Classic MacOS (at least by Adobe), and `"PDFDoc.enc"` is the first 256 characters of the Unicode encoding, the standard for PDF. There are also encodings `"ISOLatin2.enc"`, `"CP1250.enc"`, `"ISOLatin7.enc"` (ISO 8859-13), `"CP1257.enc"`, and `"ISOLatin9.enc"` (ISO 8859-15), `"Cyrillic.enc"` (ISO 8859-5), `"KOI8-R.enc"`, `"KOI8-U.enc"`, `"CP1251.enc"`, `"Greek.enc"` (ISO 8859-7) and `"CP1253.enc"`. Note that many glyphs in these encodings are not in the fonts corresponding to the standard families. (The Adobe ones for all but Courier, Helvetica and Times cover little more than Latin-1, whereas the URW ones also cover Latin-2, Latin-7, Latin-9 and Cyrillic but no Greek. The Adobe exceptions cover the Latin character sets, but not the Euro.)

If you specify the encoding, it is your responsibility to ensure that the PostScript font contains the glyphs used. One issue here is the Euro symbol which is in the WinAnsi and MacRoman encodings but may well not be in the PostScript fonts. (It is in the URW variants; it is not in the supplied Adobe Font Metric files.)

There is an exception. Character 45 (`"-"`) is always set as minus (its value in Adobe ISOLatin1) even though it is hyphen in the other encodings. Hyphen is available as character 173 (octal 0255) in all the Latin encodings, Cyrillic and Greek. (This can be entered as `"\uad"` in a UTF-8 locale.) There are some discrepancies in accounts of glyphs 39 and 96: the supplied encodings (except CP1250 and CP1251) treat these as 'quoteright' and 'quoteleft' (rather than 'quotesingle'/'acute' and 'grave' respectively), as they are in the Adobe documentation.

TeX fonts

TeX has traditionally made use of fonts such as Computer Modern which are encoded rather differently, in a 7-bit encoding. This encoding can be specified by `encoding = "TeXtext.enc"`, taking care that the ASCII characters `< > \ _ { }` are not available in those fonts.

There are supplied families `"ComputerModern"` and `"ComputerModernItalic"` which use this encoding, and which are only supported for `postscript` (and not `pdf`). They are intended to use with the Type 1 versions of the TeX CM fonts. It will normally be possible to include such output in TeX or LaTeX provided it is processed with `dvips -Ppfb -j0` or the equivalent on your system. (`-j0` turns off font subsetting.) When `family = "ComputerModern"` is used, the italic/bold-italic fonts used are slanted fonts (`cmsl10` and `cmbxsl10`). To use text italic fonts instead, set `family = "ComputerModernItalic"`.

These families use the TeX math italic and symbol fonts for a comprehensive but incomplete coverage of the glyphs covered by the Adobe symbol font in other families. This is achieved by special-casing the postscript code generated from the supplied 'CM_symbol_10.afm'.

Color models

The default color model is RGB, with pure gray colors expressed as greyscales. Color model `"rgb-nogray"` uses only RGB, model `"cmyk"` only CMYK, and model `"gray"` only greyscales (and selecting any other colour is an error). Nothing in R specifies the interpretation of the RGB or CMYK color spaces, and the simplest possible conversion to CMYK is used (http://en.wikipedia.org/wiki/CMYK_color_model#Mapping_RGB_to_CMYK).

Printing

A postscript plot can be printed via `postscript` in two ways.

1. Setting `print.it = TRUE` causes the command given in argument `command` to be called with argument `"file"` when the device is closed. Note that the plot file is not deleted unless `command` arranges to delete it.
2. `file=""` or `file="|cmd"` can be used to print using a pipe on systems that support 'popen'. Failure to open the command will probably be reported to the terminal but not to 'popen', in which case close the device by `dev.off` immediately.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the "R Internals Manual".

- The default device size is 7 inches square.

- Font sizes are in big points.
- The default font family is Helvetica.
- Line widths are as a multiple of 1/96 inch, with no minimum.
- Circle of any radius are allowed.
- Colours are interpreted by the viewing/printing application.

Author(s)

Support for Computer Modern fonts is based on a contribution by Brian D'Urso ⟨durso@hussle.harvard.edu⟩.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`postscriptFonts`, `Devices`, `check.options` which is called from both `ps.options` and `postscript`. `cairo_ps` for another device that can produce PostScript.

More details of font families and encodings and especially handling text in a non-Latin-1 encoding and embedding fonts can be found in

Paul Murrell and Brian Ripley (2006) Non-standard fonts in PostScript and PDF graphics. *R News*, 6(2):41–47. http://cran.r-project.org/doc/Rnews/Rnews_2006-2.pdf.

Examples

```
require(graphics)
## Not run:
# open the file "foo.ps" for graphics output
postscript("foo.ps")
# produce the desired graph(s)
dev.off()               # turn off the postscript device
postscript("|lp -dlw")
# produce the desired graph(s)
dev.off()               # plot will appear on printer

# for URW PostScript devices
postscript("foo.ps", family = "NimbusSan")

## for inclusion in Computer Modern TeX documents, perhaps
postscript("cm_test.eps", width = 4.0, height = 3.0,
           horizontal = FALSE, onefile = FALSE, paper = "special",
           family = "ComputerModern", encoding = "TeXtext.enc")
## The resultant postscript file can be used by dvips -Ppfb -j0.

## To test out encodings, you can use
TestChars <- function(encoding="ISOLatin1", family="URWHelvetica")
{
    postscript(encoding=encoding, family=family)
    par(pty="s")
    plot(c(-1,16), c(-1,16), type="n", xlab="", ylab="",
         xaxs="i", yaxs="i")
    title(paste("Centred chars in encoding", encoding))
```

```
    grid(17, 17, lty=1)
    for(i in c(32:255)) {
        x <- i %% 16
        y <- i %/% 16
        points(x, y, pch=i)
    }
    dev.off()
}
## there will be many warnings.  We use URW to get a complete enough
## set of font metrics.
TestChars()
TestChars("ISOLatin2")
TestChars("WinAnsi")
## End(Not run)
```

`postscriptFonts` *PostScript and PDF Font Families*

Description

These functions handle the translation of a R graphics font family name to a PostScript or PDF font description, used by the `postscript` or `pdf` graphics devices.

Usage

```
postscriptFonts(...)
pdfFonts(...)
```

Arguments

`...`	either character strings naming mappings to display, or named arguments specifying mappings to add or change.

Details

If these functions are called with no argument they list all the existing mappings, whereas if they are called with named arguments they add (or change) mappings.

A PostScript or PDF device is created with a default font family (see the documentation for `postscript`), but it is also possible to specify a font family when drawing to the device (for example, see the documentation for `"family"` in `par` and for `"fontfamily"` in `gpar` in the **grid** package).

The font family sent to the device is a simple string name, which must be mapped to a set of PostScript fonts. Separate lists of mappings for `postscript` and `pdf` devices are maintained for the current R session and can be added to by the user.

The `postscriptFonts` and `pdfFonts` functions can be used to list existing mappings and to define new mappings. The `Type1Font` and `CIDFont` functions can be used to create new mappings, when the `xxxFonts` function is used to add them to the database. See the examples.

Default mappings are provided for three device-independent family names: `"sans"` for a sans-serif font (to `"Helvetica"`), `"serif"` for a serif font (to `"Times"`) and `"mono"` for a monospaced font (to `"Courier"`).

Mappings for a number of standard Adobe fonts (and URW equivalents) are also provided: "AvantGarde", "Bookman", "Courier", "Helvetica", "Helvetica-Narrow", "NewCenturySchoolbook", "Palatino" and "Times"; "URWGothic", "URWBookman", "NimbusMon", "NimbusSan" (synonym "URWHelvetica"), "NimbusSanCond", "CenturySch", "URWPalladio" and "NimbusRom" (synonym "URWTimes").

There are also mappings for "ComputerModern" and "ComputerModernItalic".

Finally, there are some default mappings for East Asian locales described in a separate section.

The specification of font metrics and encodings is described in the help for the postscript function.

The fonts are not embedded in the resulting PostScript or PDF file, so software including the PostScript or PDF plot file should either embed the font outlines (usually from '.pfb' or '.pfa' files) or use DSC comments to instruct the print spooler or including application to do so (see also embedFonts).

A font family has both an R-level name, the argument name used when postscriptFonts was called, and an internal name, the family component. These two names are the same for all the pre-defined font families.

Once a font family is in use it cannot be changed. 'In use' means that it has been specified *via* a family or fonts argument to an invocation of the same graphics device already in the R session. (For these purposes xfig counts the same as postscript but only uses some of the predefined mappings.)

Value

A list of one or more font mappings.

East Asian fonts

There are some default mappings for East Asian locales: "Japan1", "Japan1HeiMin", "Japan1GothicBBB", and "Japan1Ryumin" for Japanese; "Korea1" and "Korea1deb" for Korean; "GB1" (Simplified Chinese) for mainland China and Singapore; "CNS1" (Traditional Chinese) for Hong Kong and Taiwan.

These refer to the following fonts

Japan1 (PS)	HeiseiKakuGo-W5
	Linotype Japanese printer font
Japan1 (PDF)	KozMinPro-Regular-Acro
	from Adobe Reader 7.0 Japanese Font Pack
Japan1HeiMin (PS)	HeiseiMin-W3
	Linotype Japanese printer font
Japan1HeiMin (PDF)	HeiseiMin-W3-Acro
	from Adobe Reader 7.0 Japanese Font Pack
Japan1GothicBBB	GothicBBB-Medium
	Japanese-market PostScript printer font
Japan1Ryumin	Ryumin-Light
	Japanese-market PostScript printer font
Korea1 (PS)	Baekmuk-Batang
	TrueType font found on some Linux systems
Korea1 (PDF)	HYSMyeongJoStd-Medium-Acro
	from Adobe Reader 7.0 Korean Font Pack
Korea1deb (PS)	Batang-Regular

	another name for Baekmuk-Batang
Korea1deb (PDF)	`HYGothic-Medium-Acro`
	from Adobe Reader 4.0 Korean Font Pack
GB1 (PS)	`BousungEG-Light-GB`
	TrueType font found on some Linux systems
GB1 (PDF)	`STSong-Light-Acro`
	from Adobe Reader 7.0 Simplified Chinese Font Pack
CNS1 (PS)	`MOESung-Regular`
	Ken Lunde's CJKV resources
CNS1 (PDF)	`MSungStd-Light-Acro`
	from Adobe Reader 7.0 Traditional Chinese Font Pack

`Baekmuk-Batang` can be found at `ftp://ftp.mizi.com/pub/baekmuk/`. `BousungEG-Light-GB` can be found at `ftp://ftp.gnu.org/pub/non-gnu/chinese-fonts-truetype/`. Ken Lunde's CJKV resources are at `ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/adobe/samples/`. These will need to be installed or otherwise made available to the postscript/PDF interpreter such as ghostscript (and not all interpreters can handle TrueType fonts).

You may well find that your postscript/PDF interpreters has been set up to provide aliases for many of these fonts. For example, ghostscript on Windows can optionally be installed to map common CJK fonts names to Windows TrueType fonts. (You may want to add the `-Acro` versions as well.)

Adding a mapping for a CID-keyed font is for gurus only.

Author(s)

Support for Computer Modern fonts is based on a contribution by Brian D'Urso ⟨durso@hussle.harvard.edu⟩.

See Also

`postscript` and `pdf`; `Type1Font` and `CIDFont` for specifying new font mappings.

Examples

```
postscriptFonts()
## This duplicates "ComputerModernItalic".
CMitalic <- Type1Font("ComputerModern2",
                      c("CM_regular_10.afm", "CM_boldx_10.afm",
                        "cmti10.afm", "cmbxti10.afm",
                         "CM_symbol_10.afm"),
                      encoding = "TeXtext.enc")
postscriptFonts(CMitalic = CMitalic)

## A CID font for Japanese using a different CMap and
## corresponding cmapEncoding.
`Jp_UCS-2` <- CIDFont("TestUCS2",
                  c("Adobe-Japan1-UniJIS-UCS2-H.afm",
                    "Adobe-Japan1-UniJIS-UCS2-H.afm",
                    "Adobe-Japan1-UniJIS-UCS2-H.afm",
                    "Adobe-Japan1-UniJIS-UCS2-H.afm"),
                  "UniJIS-UCS2-H", "UCS-2")
pdfFonts(`Jp_UCS-2` = `Jp_UCS-2`)
names(pdfFonts())
```

`ps.options` *Auxiliary Function to Set/View Defaults for Arguments of postscript*

Description

The auxiliary function `ps.options` can be used to set or view (if called without arguments) the default values for some of the arguments to `postscript`.

`ps.options` needs to be called before calling `postscript`, and the default values it sets can be overridden by supplying arguments to `postscript`.

Usage

```
ps.options(..., reset = FALSE, override.check = FALSE)

setEPS(...)
setPS(...)
```

Arguments

`...`	arguments `onefile`, `family`, `title`, `fonts`, `encoding`, `bg`, `fg`, `width`, `height`, `horizontal`, `pointsize`, `paper`, `pagecentre`, `print.it`, `command` and `colormodel` can be supplied. `onefile`, `horizontal` and `paper` are *ignored* for `setEPS` and `setPS`.
`reset`	logical: should the defaults be reset to their 'factory-fresh' values?
`override.check`	logical argument passed to `check.options`. See the Examples.

Details

If both `reset = TRUE` and `...` are supplied the defaults are first reset to the 'factory-fresh' values and then the new values are applied.

For backwards compatibility argument `append` is accepted but ignored with a warning.

`setEPS` and `setPS` are wrappers to set defaults appropriate for figures for inclusion in documents (the default size is 7 inches square unless `width` or `height` is supplied) and for spooling to a PostScript printer respectively. For historical reasons the latter is the ultimate default.

Value

A named list of all the previous defaults. If `...` or `reset = TRUE` is supplied the result has the visibility flag turned off.

See Also

`postscript`, `pdf.options`

Examples

```
ps.options(bg = "pink")
utils::str(ps.options())

### ---- error checking of arguments: ----
ps.options(width=0:12, onefile=0, bg=pi)
# override the check for 'width', but not 'bg':
ps.options(width=0:12, bg=pi, override.check = c(TRUE,FALSE))
utils::str(ps.options())
ps.options(reset = TRUE) # back to factory-fresh
```

---

quartz *MacOS X Quartz Device*

Description

`quartz` starts a graphics device driver for the Mac OS X System. It supports plotting both to the screen (the default) and to various graphics file formats.

Usage

```
quartz(title, width, height, pointsize, family, fontsmooth, antialias,
       type, file = NULL, bg, canvas, dpi)

quartz.options(..., reset = FALSE)
```

Arguments

`title`	title for the Quartz window (applies to on-screen output only), default `"Quartz %d"`. A C-style format for an integer will be substituted by the device number (see the `file` argument to `postscript` for further details).
`width`	the width of the plotting area in inches. Default `7`.
`height`	the height of the plotting area in inches. Default `7`.
`pointsize`	the default pointsize to be used. Default `12`.
`family`	this is the family name of the font that will be used by the device. Default `"Helvetica"`.
`fontsmooth`	logical specifying if fonts should be smoothed. Default `TRUE`. Currently unused.
`antialias`	whether to use antialiasing. Default `TRUE`.
`type`	the type of output to use. See 'Details' for more information. Default `"native"`.
`file`	an optional target for the graphics device. The default, `NULL`, selects a default name where one is needed. See 'Details' for more information.
`bg`	the initial background colour to use for the device. Default `"transparent"`. An opaque colour such as `"white"` will normally be required on off-screen types that support transparency such as `"png"` and `"tiff"`.
`canvas`	canvas colour to use for an on-screen device. Default `"white"`, and will be forced to be an opaque colour.

dpi	resolution of the output. The default (NA_real_) for an on-screen display defaults to the resolution of the main screen, and to 72 dpi otherwise. See 'Details'.
...	Any of the arguments to quartz except file.
reset	logical: should the defaults be reset to their defaults?

Details

The defaults for all but one of the arguments of quartz are set by quartz.options: the 'Arguments' section gives the 'factory-fresh' defaults.

The Quartz graphics device supports a variety of output types. On-screen output types are "" or "native" (picks the best possible on-screen output), "Cocoa" (Mac OS X 10.4 and later) and "Carbon" (not currently implemented – potentially Mac OS X 10.3 and earlier). Off-screen output types produce output files and utilize the file argument. type = "pdf" gives PDF output. The following bitmap formats may be supported (on OS X 10.4 and later): "png", "jpeg", "jpg", "jpeg2000", "tif", "tiff", "gif", "psd" (Adobe Photoshop), "bmp" (Windows bitmap), "sgi" and "pict". (The availability of some formats is OS-version-dependent.)

To reproduce the default of older Quartz devices on-screen, set dpi = 72 (for a permanent solution set quartz.options(dpi = 72)).

The file argument is used for off-screen drawing. The actual file is only created when the device is closed (e.g. using dev.off()). For the bitmap devices, the page number is substituted if a C integer format is included in the character string, e.g. Rplot%03d.png. (The result must be less than PATH_MAX characters long, and may be truncated if not. See postscript for further details.) If a file argument is not supplied, the default is Rplots.pdf or Rplot%03d.*type*.

If a device-independent R graphics font family is specified (e.g., via par(family=) in the graphics package), the Quartz device makes use of the Quartz font database (see quartzFonts) to convert the R graphics font family to a Quartz-specific font family description.

On-screen devices are launched with a semi-transparent canvas. Once a new plot is created, the canvas is first painted with the canvas colour and then the current background colour (which can be transparent or semi-transparent). Off-screen devices have no canvas colour, and so start with a transparent background where possible (e.g. type="png" and type="tiff") – otherwise it appears that a solid white canvas is assumed in the Quartz code.

title can be used for on-screen output. It must be a single character string with an optional integer printf-style format that will be substituted by the device number. It is also optionally used (without a format) to give a title to a PDF file.

Calling quartz() sets .Device to "quartz" for on-scren devices and to "quartz_off_screen" otherwise.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the "R Internals Manual".

- The default device size is 7 inches square.
- Font sizes are in big points.
- The default font family is Arial.
- Line widths are a multiple of 1/96 inch with no minimum set by R.
- Circle radii are real-valued with no minimum set by R.
- Colour interpretation is by the viewer, including the screen display.

See Also

quartzFonts, Devices.

png for way to access the bitmap types of this device via R's standard bitmap devices.

Examples

```
## Not run:
## put something this is your .Rprofile to customize the defaults
setHook(packageEvent("grDevices", "onLoad"),
        function(...) grDevices::quartz.options(width=8, height=6,
                                                pointsize=10))
## End(Not run)
```

---

quartzFonts *quartz Fonts*

Description

These functions handle the translation of a device-independent R graphics font family name to a quartz font description.

Usage

```
quartzFont(family)

quartzFonts(...)
```

Arguments

family	a character vector containing the four PostScript font names for plain, bold, italic, and bolditalic versions of a font family.
...	either character strings naming mappings to display, or new (named) mappings to define.

Details

A quartz device is created with a default font (see the documentation for quartz), but it is also possible to specify a font family when drawing to the device (for example, see the documentation for gpar in the grid package).

The font family sent to the device is a simple string name, which must be mapped to something more specific to quartz fonts. A list of mappings is maintained and can be modified by the user.

The quartzFonts function can be used to list existing mappings and to define new mappings. The quartzFont function can be used to create a new mapping.

Default mappings are provided for three device-independent font family names: "sans" for a sans-serif font, "serif" for a serif font and "mono" for a monospaced font.

See Also

quartz

Examples

```
quartzFonts()
quartzFonts("mono")
```

---

recordGraphics *Record graphics operations*

Description

Records arbitrary code on the graphics engine display list. Useful for encapsulating calculations with graphical output that depends on the calculations. Intended *only* for expert use.

Usage

```
recordGraphics(expr, list, env)
```

Arguments

expr	object of mode expression or call or an unevaluated expression.
list	a list defining the environment in which expr is to be evaluated.
env	An environment specifying where R looks for objects not found in envir.

Details

The code in expr is evaluated in an environment constructed from list, with env as the parent of that environment.

All three arguments are saved on the graphics engine display list so that on a device resize or copying between devices, the original evaluation environment can be recreated and the code can be re-evaluated to reproduce the graphical output.

Value

The value from evaluating expr.

Warning

This function is not intended for general use. Incorrect or improper use of this function could lead to unintended and/or undesirable results.

An example of acceptable use is querying the current state of a graphics device or graphics system setting and then calling a graphics function.

An example of improper use would be calling the assign function to performing assignments in the global environment.

See Also

eval

Examples

```
require(graphics)

plot(1:10)
# This rectangle remains 1inch wide when the device is resized
recordGraphics(
  {
    rect(4, 2,
         4 + diff(par("usr")[1:2])/par("pin")[1], 3)
  },
  list(),
  getNamespace("graphics"))
```

---

recordPlot *Record and Replay Plots*

Description

Functions to save the current plot in an R variable, and to replay it.

Usage

```
recordPlot()
replayPlot(x)
```

Arguments

x	A saved plot.

Details

These functions record and replay the displaylist of the current graphics device. The returned object is of class "recordedplot", and replayPlot acts as a print method for that class.

Value

recordPlot returns an object of class "recordedplot".

replayPlot has no return value.

Warning

The format of recorded plots may change between R versions. Recorded plots should **not** be used as a permanent storage format for R plots.

R will always attempt to replay a recorded plot, but if the plot was recorded with a different R version then bad things may happen.

`rgb` *RGB Color Specification*

Description

This function creates colors corresponding to the given intensities (between 0 and `max`) of the red, green and blue primaries.

An alpha transparency value can also be specified (0 means fully transparent and `max` means opaque). If `alpha` is not specified, an opaque colour is generated.

The names argument may be used to provide names for the colors.

The values returned by these functions can be used with a `col=` specification in graphics functions or in `par`.

Usage

```
rgb(red, green, blue, alpha, names = NULL, maxColorValue = 1)
```

Arguments

`red, blue, green, alpha`
: numeric vectors with values in $[0, M]$ where M is `maxColorValue`. When this is `255`, the `red`, `blue`, `green`, and `alpha` values are coerced to integers in `0:255` and the result is computed most efficiently.

`names`
: character. The names for the resulting vector.

`maxColorValue`
: number giving the maximum of the color values range, see above.

Details

The colors may be specified by passing a matrix or dataframe as argument `red`, and leaving `blue` and `green` missing. In this case the first three columns of `red` are taken to be the `red`, `green` and `blue` values.

Semi-transparent colors (`0 < alpha < 1`) are supported only on some devices: at the time of writing on the `pdf`, `windows`, `quartz` and `X11(type="cairo")` devices and associated bitmap devices (`jpeg`, `png`, `bmp`, `tiff` and `bitmap`). They are supported by several third-party devices such as those in packages **Cairo**, **cairoDevice** and **JavaGD**. Only some of these devices support semi-transparent backgrounds.

Most other graphics devices plot semi-transparent colors as fully transparent, usually with a warning when first encountered.

Value

A character vector with elements of 7 or 9 characters, `"#"` followed by the red, blue, green and optionally alpha values in hexadecimal (after rescaling to `0 ... 255`).

See Also

`col2rgb`for translating R colors to RGB vectors; `rainbow`, `hsv`, `hcl`, `gray`.

Examples

```
rgb(0,1,0)

rgb((0:15)/15, green=0, blue=0, names=paste("red",0:15,sep="."))

rgb(0, 0:12, 0, max = 255)# integer input

ramp <- colorRamp(c("red", "white"))
rgb( ramp(seq(0, 1, length = 5)), max = 255)
```

---

rgb2hsv *RGB to HSV Conversion*

Description

`rgb2hsv` transforms colors from RGB space (red/green/blue) into HSV space (hue/saturation/value).

Usage

```
rgb2hsv(r, g = NULL, b = NULL, gamma = 1, maxColorValue = 255)
```

Arguments

`r`	vector of 'red' values in $[0, M]$, (M =`maxColorValue`) or 3-row rgb matrix.
`g`	vector of 'green' values, or `NULL` when `r` is a matrix.
`b`	vector of 'blue' values, or `NULL` when `r` is a matrix.
`gamma`	a gamma-correction (supposedly applied to the r,g,b values previously), see `hsv(...., gamma)`.
`maxColorValue`	number giving the maximum of the RGB color values range. The default `255` corresponds to the typical `0:255` RGB coding as in `col2rgb()`.

Details

Value (brightness) gives the amount of light in the color.
Hue describes the dominant wavelength.
Saturation is the amount of Hue mixed into the color.

Value

A matrix with a column for each color. The three rows of the matrix indicate hue, saturation and value and are named `"h"`, `"s"`, and `"v"` accordingly.

Author(s)

R interface by Wolfram Fischer ⟨wolfram@fischer-zim.ch⟩;
C code mainly by Nicholas Lewin-Koh ⟨nikko@hailmail.net⟩.

See Also

`hsv`, `col2rgb`, `rgb`.

Examples

```
## These (saturated, bright ones) only differ by hue
(rc <- col2rgb(c("red", "yellow","green","cyan", "blue", "magenta")))
(hc <- rgb2hsv(rc))
6 * hc["h",] # the hues are equispaced


(rgb3 <- floor(256 * matrix(stats::runif(3*12), 3,12)))
(hsv3 <- rgb2hsv(rgb3))
## Consistency :
stopifnot(rgb3 == col2rgb(hsv(h=hsv3[1,], s=hsv3[2,], v=hsv3[3,])),
          all.equal(hsv3, rgb2hsv(rgb3/255, maxColorValue = 1)))

## A (simplified) pure R version -- originally by Wolfram Fischer --
## showing the exact algorithm:
rgb2hsvR <- function(rgb, gamma = 1, maxColorValue = 255)
{
    if(!is.numeric(rgb)) stop("rgb matrix must be numeric")
    d <- dim(rgb)
    if(d[1] != 3) stop("rgb matrix must have 3 rows")
    n <- d[2]
    if(n == 0) return(cbind(c(h=1,s=1,v=1))[,0])
    rgb <- rgb/maxColorValue
    if(gamma != 1) rgb <- rgb ^ (1/gamma)

    ## get the max and min
    v <- apply( rgb, 2, max)
    s <- apply( rgb, 2, min)
    D <- v - s # range

    ## set hue to zero for undefined values (gray has no hue)
    h <- numeric(n)
    notgray <- ( s != v )

    ## blue hue
    idx <- (v == rgb[3,] & notgray )
    if (any (idx))
        h[idx] <- 2/3 + 1/6 * (rgb[1,idx] - rgb[2,idx]) / D[idx]
    ## green hue
    idx <- (v == rgb[2,] & notgray )
    if (any (idx))
        h[idx] <- 1/3 + 1/6 * (rgb[3,idx] - rgb[1,idx]) / D[idx]
    ## red hue
    idx <- (v == rgb[1,] & notgray )
    if (any (idx))
        h[idx] <-       1/6 * (rgb[2,idx] - rgb[3,idx]) / D[idx]

    ## correct for negative red
    idx <- (h < 0)
    h[idx] <- 1+h[idx]

    ## set the saturation
    s[! notgray] <- 0;
    s[notgray] <- 1 - s[notgray] / v[notgray]

    rbind( h=h, s=s, v=v )
```

```
}

## confirm the equivalence:
all.equal(rgb2hsv (rgb3),
          rgb2hsvR(rgb3), tol=1e-14) # TRUE
```

`savePlot` *Save Cairo X11 Plot to File*

Description

Save the current page of a cairo `X11()` device to a file.

Usage

```
savePlot(filename = paste("Rplot", type, sep="."),
         type = c("png", "jpeg", "tiff", "bmp"),
         device = dev.cur())
```

Arguments

`filename`	filename to save to.
`type`	file type: only `"png"` will be accepted for cairo version 1.0.
`device`	the device to save from.

Details

Only `X11` devices of types `"cairo"` and `"nbcairo"` are supported.

This works by copying the image surface to a file. For PNG will always be a 24-bit per pixel PNG 'DirectClass' file, for JPEG the quality is 75% and for TIFF there is no compression.

At present the plot is saved after rendering onto the canvas (default opaque white), so for the default `bg = "transparent"` the effective background colour is the canvas colour.

Value

Invisible `NULL`.

See Also

`X11`, `dev.copy`, `dev.print`

trans3d *3D to 2D Transformation for Perspective Plots*

Description

Projection of 3-dimensional to 2-dimensional points using a 4x4 viewing transformation matrix. Mainly for adding to perspective plots such as `persp`.

Usage

```
trans3d(x,y,z, pmat)
```

Arguments

`x, y, z`	numeric vectors of equal length, specifying points in 3D space.
`pmat`	a 4×4 *viewing transformation matrix*, suitable for projecting the 3D coordinates (x, y, z) into the 2D plane using homogeneous 4D coordinates (x, y, z, t); such matrices are returned by `persp()`.

Value

a list with two components

`x,y`	the projected 2d coordinates of the 3d input `(x,y,z)`.

See Also

`persp`

Examples

```
## See  help(persp) {after attaching the 'graphics' package}
##      -----------
```

Type1Font *Type 1 and CID Fonts*

Description

These functions are used to define the translation of a R graphics font family name to a Type 1 or CID font descriptions, used by both the `postscript` and `pdf` graphics devices.

Usage

```
Type1Font(family, metrics, encoding = "default")

CIDFont(family, cmap, cmapEncoding, pdfresource = "")
```

Arguments

family	a character string giving the name to be used internally for a Type 1 or CID-keyed font family. This needs to uniquely identify each family, so if you modify a family which is in use (see postscriptFonts) you need to change the family name.
metrics	a character vector of four or five strings giving paths to the afm (Adobe Font Metric) files for the font.
cmap	the name of a CMap file for a CID-keyed font.
encoding	for Type1Font, the name of an encoding file. Defaults to "default", which maps on this platform to "ISOLatin1.enc". Otherwise, a file name in the 'enc' directory of the **grDevices** package, which is used if the path does not contain a path separator. An extension ".enc" can be omitted.
cmapEncoding	The name of a character encoding to be used with the named CMap file: strings will be translated to this encoding when written to the file.
pdfresource	A chunk of PDF code; only required for using a CID-keyed font on pdf; users should not be expected to provide this.

Details

For Type1Fonts, if four '.afm' files are supplied the fifth is taken to be "Symbol.afm". Relative paths are taken relative to the directory '*R_HOME*/library/grDevices/afm'. The fifth (symbol) font must be in AdobeSym encoding. However, the glyphs in the first four fonts are referenced by name and any encoding given within the '.afm' files is not used.

Glyphs in CID-keyed fonts are accessed by ID (number) and not by name. The CMap file maps encoded strings (usually in a MBCS) to IDs, so cmap and cmapEncoding specifications must match. There are no real bold or italic versions of CID fonts (bold/italic were very rarely used in traditional CJK topography), and for the pdf device all four font faces will be identical. However, for the postscript device, bold and italic (and bold italic) are emulated.

CID-keyed fonts are intended only for use for the glyphs of CJK languages, which are all monospaced and are all treated as filling the same bounding box. (Thus plotmath will work with such characters, but the spacing will be less carefully controlled than with Western glyphs.) The CID-keyed fonts do contain other characters, including a Latin alphabet: non-CJK glyphs are regarded as monospaced with half the width of CJK glyphs. This is often the case, but sometimes Latin glyphs designed for proportional spacing are used (and may look odd). We strongly recommend that CID-keyed fonts are **only** used for CJK glyphs.

Value

A list of class "Type1Font" or "CIDFont".

See Also

postscript, pdf, postscriptFonts, and pdfFonts.

Examples

```
## This duplicates "ComputerModernItalic".
CMitalic <- Type1Font("ComputerModern2",
                      c("CM_regular_10.afm", "CM_boldx_10.afm",
                        "cmti10.afm", "cmbxti10.afm",
                        "CM_symbol_10.afm"),
```

```
                        encoding = "TeXtext.enc")

## Not run:
## This could be used by
postscript(family = CMitalic)
## or
postscriptFonts(CMitalic = CMitalic)  # once in a session
postscript(family = "CMitalic", encoding = "TeXtext.enc")
## End(Not run)
```

`x11` *X Window System Graphics*

Description

`X11` starts a graphics device driver for the X Window System (version 11). This can only be done on machines/accounts that have access to an X server.

`x11` is recognized as a synonym for `X11`.

Usage

```
X11(display = "", width, height, pointsize, gamma, bg, canvas,
    fonts, xpos, ypos, title, type, antialias)

X11.options(..., reset = FALSE)
```

Arguments

`display`	the display on which the graphics window will appear. The default is to use the value in the user's environment variable `DISPLAY`. This is ignored (with a warning) if an X11 device is already open on another display.
`width, height`	the width and height of the plotting window, in inches. If `NA`, taken from the resources and if not specified there defaults to `7` inches. See also 'Resources'.
`pointsize`	the default pointsize to be used. Defaults to `12`.
`gamma`	the gamma correction factor. This value is used to help ensure that the colours perceived are linearly related to RGB values (see `hsv`). By default 1 (default correction).
`bg`	colour, the initial background colour. Default `"transparent"`.
`canvas`	colour. The colour of the canvas, which is visible only when the background colour is transparent. Should be a solid colour (and any alpha value will be ignored). Default `"white"`.
`fonts`	X11 font description strings into which weight, slant and size will be substituted. There are two, the first for fonts 1 to 4 and the second for font 5, the symbol font. See section 'Fonts'.
`xpos, ypos`	integer: initial position of the top left corner of the window, in pixels. Negative values are from the opposite corner, e.g. `xpos=-100` says the top right corner should be 100 pixels from the right edge of the screen. If `NA` (the default), successive devices are cascaded in 20 pixel steps from the top left. See also 'Resources'.

title	character string, up to 100 bytes. With the default, "", a suitable title is created internally. A C-style format for an integer will be substituted by the device number (see the file argument to postscript for further details). How non-ASCII titles are handled is implementation-dependent.
type	character string, one of "Xlib" (the only type prior to R 2.7.0) or "cairo" or "nbcairo". The latter two will only be available if the system was compiled with support for cairo. Default "cairo" where available, otherwise "Xlib".
antialias	for cairo types, the type of anti-aliasing (if any) to be used. One of c("default", "none", "gray", "subpixel").
reset	logical: should the defaults be reset to their defaults?
...	Any of the arguments to X11, plus colortype and maxcubesize (see section 'Colour Rendering').

Details

The defaults for all of the arguments of X11 are set by X11.options: the 'Arguments' section gives the 'factory-fresh' defaults.

The initial size and position are only hints, and may not be acted on by the window manager. Also, some systems (especially laptops) are set up to appear to have a screen of a different size to the physical screen.

Option type selects between two separate devices: R can be built with support for neither, type = "Xlib" or both. Where both are available, types "cairo" and "nbcairo" offer

- antialiasing of text and lines.
- translucent colours.
- scalable text, including to sizes like 4.5 pt.
- full support for UTF-8, so on systems with suitable fonts you can plot in many languages on a single figure (and this will work even in non-UTF-8 locales). The output should be locale-independent.

type = "nbcairo" is the same device as type="cairo" without buffering: which is faster will depend on the X11 connection. Both will be slower than type = "Xlib", especially on a slow X11 connection as all the rendering is done on the machine running R rather than in the X server.

All devices which use an X11 server (including the type = "Xlib" versions of bitmap devices such as png) share internal structures, which means that they must use the same display and visual. If you want to change display, first close all such devices.

X11 Fonts

This section applies only to type = "Xlib".

An initial/default font family for the device can be specified via the fonts argument, but if a device-independent R graphics font family is specified (e.g., via par(family=) in the graphics package), the X11 device makes use of the X11 font database (see X11Fonts) to convert the R graphics font family to an X11-specific font family description.

X11 chooses fonts by matching to a pattern, and it is quite possible that it will choose a font in the wrong encoding or which does not contain glyphs for your language (particularly common in iso10646-1 fonts).

The fonts argument is a two-element character vector, and the first element will be crucial in successfully using non-Western-European fonts. Settings that have proved useful include

"-*-mincho-%s-%s-*-*-%d-*-*-*-*-*-*-*" for CJK languages and "-cronyx-helvetica-%s-%s-*-*-%d-*-*-*-*-*-*-*" for Russian.

For UTF-8 locales, the XLC_LOCALE databases provide mappings between character encodings, and you may need to add an entry for your locale (e.g. Fedora Core 3 lacked one for ru_RU.utf8).

Cairo Fonts

The cairo-based device works directly with font family names such as "Helvetica" which should be selected by par or gpar. There are mappings for the three device-independent font families, "sans" for a sans-serif font (to "Helvetica"), "serif" for a serif font (to "Times") and "mono" for a monospaced font (to "Courier").

The font selection is handled by Pango (usually) or cairo (on Mac OS X and perhaps elsewhere). Both make use of fontconfig (http://wwww.fontconfig.org) to select fonts and so the results depend on the fonts installed on the system running R – setting the environmnent variable FC_DEBUG to 1 allows some tracing of the selection process.

This works best when high-quality scalable fonts are installed, usually in Type 1 or TrueType formats: see the "R Installation and Administration Manual" for advice on how to obtain and install such fonts.

Resources

The standard X11 resource geometry can be used to specify the window position and/or size, but will be overridden by values specified as arguments or non-NA defaults set in X11.options. The class looked for is R_x11. Note that the resource specifies the width and height in pixels and not in inches. See for example http://web.mit.edu/answers/xwindows/xwindows_resources.html and perhaps 'man X' (or http://www.xfree86.org/current/X.7.html). An example line in '~/.Xresources' might be

```
R_x11*geometry: 900x900-0+0
```

which specifies a 900 x 900 pixel window at the top right of the screen.

Colour Rendering

X11 supports several 'visual' types, and nowadays almost all systems support 'truecolor' which X11 will use by default. This uses a direct specification of any RGB colour up to the depth supported (usually 8 bits per colour). Other visuals make use of a palette to support fewer colours, only grays or even only black/white. The palette is shared between all X11 clients, so it can be necessary to limit the number of colours used by R.

Cairo-based devices currently support only 'truecolor' visuals. (Cairo 1.6 will support other visuals.)

The default for type="Xlib" is to use the best possible colour model for the visual of the X11 server. This can be overridden by the colortype argument of X11.options. **Note:** All X11 and type = "Xlib" bmp, jpeg, png and tiff devices share a colortype which is set when the first device to be opened. To change the colortype you need to close *all* open such devices, and then use X11.options(colortype=).

The colortype types are tried in the order "true", "pseudo", "gray" and "mono" (black or white only). The values "pseudo" and "pseudo.cube" provide colour strategies for a pseudocolor visual. The first strategy provides on-demand colour allocation which produces exact colours until the colour resources of the display are exhausted (when plotting will fail). The second allocates (if possible) a standard colour cube, and requested colours are approximated by the closest value in the cube.

With `colortype` equal to `"pseudo.cube"` or `"gray"` successively smaller palettes are tried until one is completely allocated. If allocation of the smallest attempt fails the device will revert to `"mono"`. For `"gray"` the search starts at 256 grays for a display with depth greater than 8, otherwise with half the available colours. For `"pseudo.cube"` the maximum cube size is set by `X11.options(maxcolorsize=)` and defaults to 256. With that setting the largest cube tried is 4 levels each for RGB, using 64 colours in the palette.

Anti-aliasing

Anti-aliasing is only supported for cairo-based devices, and applies to graphics and to fonts. It is generally preferable for lines and text, but can lead to undesirable effects for fills, e.g. for `image` plots, and so is never used for fills.

`antialias = "default"` is in principle platform-dependent, but seems most often equivalent to `antialias = "gray"`.

Conventions

This section describes the implementation of the conventions for graphics devices set out in the "R Internals Manual".

- The default device size is 7 inches square.
- Font sizes are in big points.
- The default font family is Helvetica.
- Line widths in 1/96 inch, minimum one pixel for `type = "Xlib"`, 0.01 otherwise.
- For `type = "Xlib"` circle radii are in pixels with minimum one.
- Colours are interpreted by the X11 server, normally in a fair approximation to sRGB.

See Also

`Devices`, `X11Fonts`, `savePlot`.

Examples

```
## Not run:
## put something this is your .Rprofile to customize the defaults
setHook(packageEvent("grDevices", "onLoad"),
        function(...) grDevices::X11.options(width=8, height=6, xpos=0,
                                             pointsize=10))
## End(Not run)
```

X11Fonts *X11 Fonts*

Description

These functions handle the translation of a device-independent R graphics font family name to an X11 font description.

Usage

```
X11Font(font)

X11Fonts(...)
```

Arguments

font	a character string containing an X11 font description.
...	either character strings naming mappings to display, or new (named) mappings to define.

Details

These functions apply only to an X11 device with type = "Xlib" – X11(type = "Cairo" uses a different mechanism to select fonts.

Such a device is created with a default font (see the documentation for X11), but it is also possible to specify a font family when drawing to the device (for example, see the documentation for "family" in par and for "fontfamily" in gpar in the **grid** package).

The font family sent to the device is a simple string name, which must be mapped to something more specific to X11 fonts. A list of mappings is maintained and can be modified by the user.

The X11Fonts function can be used to list existing mappings and to define new mappings. The X11Font function can be used to create a new mapping.

Default mappings are provided for three device-independent font family names: "sans" for a sans-serif font, "serif" for a serif font and "mono" for a monospaced font.

See Also

X11

Examples

```
X11Fonts()
X11Fonts("mono")
utopia <- X11Font("-*-utopia-*-*-*-*-*-*-*-*-*-*-*-*")
X11Fonts(utopia=utopia)
```

---

xfig *XFig Graphics Device*

Description

xfig starts the graphics device driver for producing XFig (version 3.2) graphics.

The auxiliary function ps.options can be used to set and view (if called without arguments) default values for the arguments to xfig and postscript.

Usage

```
xfig(file = ifelse(onefile, "Rplots.fig", "Rplot%03d.fig"),
     onefile = FALSE, encoding = "none",
     paper = "default", horizontal = TRUE,
     width = 0, height = 0, family = "Helvetica",
     pointsize = 12, bg = "transparent", fg = "black",
     pagecentre = TRUE)
```

Arguments

file	a character string giving the name of the file. For use with onefile = FALSE give a C integer format such as "Rplot%03d.fig" (the default in that case). (See postscript for further details.)
onefile	logical: if true allow multiple figures in one file. If false, assume only one page per file and generate a file number containing the page number.
encoding	The encoding in which to write text strings. The default is not to re-encode. This can be any encoding recognized by iconv: in a Western UTF-8 locale you probably want to select an 8-bit encoding such as latin1, and in an East Asian locale an EUC encoding. If re-encoding fails, the text strings will be written in the current encoding with a warning.
paper	the size of paper region. The choices are "A4", "Letter" and "Legal" (and these can be lowercase). A further choice is "default", which is the default. If this is selected, the papersize is taken from the option "papersize" if that is set to a non-empty value, otherwise "A4".
horizontal	the orientation of the printed image, a logical. Defaults to true, that is landscape orientation.
width, height	the width and height of the graphics region in inches. The default is to use the entire page less a 0.5 inch overall margin in each direction. (See postscript for further details.)
family	the font family to be used. This must be one of "AvantGarde", "Bookman", "Courier", "Helvetica" (the default), "Helvetica-Narrow", "NewCenturySchoolbook", "Palatino" or "Times". Any other value is replaced by "Helvetica", with a warning.
pointsize	the default point size to be used.
bg	the initial background color to be used.
fg	the initial foreground color to be used.
pagecentre	logical: should the device region be centred on the page?

Details

Although xfig can produce multiple plots in one file, the XFig format does not say how to separate or view them. So onefile = FALSE is the default.

The file argument is interpreted as a C integer format as used by sprintf, with integer argument the page number. The default gives files 'Rplot001.fig', ..., 'Rplot999.fig', 'Rplot1000.fig', ...

Line widths as controlled by par(lwd=) are in multiples of 5/6*1/72 inch. Multiples less than 1 are allowed. pch="." with cex = 1 corresponds to a square of side 1/72 inch.

Windows users can make use of WinFIG (http://www.schmidt-web-berlin.de/WinFIG.htm, shareware).

Conventions

This section describes the implementation of the conventions for graphics devices set out in the "R Internals Manual".

- The default device size is the paper size with a 0.25 inch border on all sides.
- Font sizes are in big points.
- The default font family is Helvetica.
- Line widths are integers, multiples of 5/432 inch.
- Circle radii are multiples of 1/1200 inch.
- Colours are interpreted by the viewing/printing application.

Note

Only some line textures (`0 <= lty < 4`) are used. Eventually this may be partially remedied, but the XFig file format does not allow as general line textures as the R model. Unimplemented line textures are displayed as *dash-double-dotted*.

There is a limit of 512 colours (plus white and black) per file.

See Also

`Devices`, `postscript`, `ps.options`.

`xy.coords` *Extracting Plotting Structures*

Description

`xy.coords` is used by many functions to obtain x and y coordinates for plotting. The use of this common mechanism across all relevant R functions produces a measure of consistency.

Usage

```
xy.coords(x, y = NULL, xlab = NULL, ylab = NULL, log = NULL,
          recycle = FALSE)
```

Arguments

`x, y`	the x and y coordinates of a set of points. Alternatively, a single argument `x` can be provided.
`xlab,ylab`	names for the x and y variables to be extracted.
`log`	character, `"x"`, `"y"` or both, as for `plot`. Sets negative values to `NA` and gives a warning.
`recycle`	logical; if `TRUE`, recycle (`rep`) the shorter of `x` or `y` if their lengths differ.

Details

An attempt is made to interpret the arguments `x` and `y` in a way suitable for bivariate plotting (or other bivariate procedures).

If `y` is `NULL` and `x` is a

formula: of the form `yvar ~ xvar`. `xvar` and `yvar` are used as x and y variables.

list: containing components `x` and `y`, these are used to define plotting coordinates.

time series: the x values are taken to be `time(x)` and the y values to be the time series.

matrix or `data.frame` with two or more columns: the first is assumed to contain the x values and the second the y values. *Note* that is also true if `x` has columns named `"x"` and `"y"`; these names will be irrelevant here.

In any other case, the `x` argument is coerced to a vector and returned as **y** component where the resulting `x` is just the index vector `1:n`. In this case, the resulting `xlab` component is set to `"Index"`.

If `x` (after transformation as above) inherits from class `"POSIXt"` it is coerced to class `"POSIXct"`.

Value

A list with the components

`x`	numeric (i.e., `"double"`) vector of abscissa values.
`y`	numeric vector of the same length as `x`.
`xlab`	`character(1)` or `NULL`, the 'label' of `x`.
`ylab`	`character(1)` or `NULL`, the 'label' of `y`.

See Also

`plot.default`, `lines`, `points` and `lowess` are examples of functions which use this mechanism.

Examples

```
xy.coords(stats::fft(c(1:10)), NULL)

with(cars, xy.coords(dist ~ speed, NULL)$xlab ) # = "speed"

xy.coords(1:3, 1:2, recycle=TRUE)
xy.coords(-2:10,NULL, log="y")
##> warning: 3 y values <=0 omitted ..
```

xyTable *Multiplicities of (x,y) Points, e.g., for a Sunflower Plot*

Description

Given (x,y) points, determine their multiplicity – checking for equality only up to some (crude kind of) noise. Note that this is special kind of 2D binning.

Usage

```
xyTable(x, y = NULL, digits)
```

Arguments

x,y	numeric vectors of the same length; alternatively other (x,y) argument combinations as allowed by xy.coords(x,y).
digits	integer specifying the significant digits to be used for determining equality of coordinates. These are compared after rounding them via signif(*,digits).

Value

A list with three components of same length,

x	x coordinates, rounded and sorted.
y	y coordinates, rounded (and sorted within x).
number	multiplicities (positive integers); i.e., number[i] is the multiplicity of (x[i],y[i]).

See Also

sunflowerplot which typically uses xyTable(); signif.

Examples

```
xyTable(iris[,3:4], digits = 6)

## Discretized uncorrelated Gaussian:

require(stats)
xy <- data.frame(x = round(sort(rnorm(100))), y = rnorm(100))
xyTable(xy, digits = 1)
```

`xyz.coords` *Extracting Plotting Structures*

Description

Utility for obtaining consistent x, y and z coordinates and labels for three dimensional (3D) plots.

Usage

```
xyz.coords(x, y = NULL, z = NULL,
           xlab = NULL, ylab = NULL, zlab = NULL,
           log = NULL, recycle = FALSE)
```

Arguments

`x, y, z`	the x, y and z coordinates of a set of points. Both `y` and `z` can be left at `NULL`. In this case, an attempt is made to interpret `x` in a way suitable for plotting. If the argument is a formula `zvar ~ xvar + yvar`, `xvar`, `yvar` and `zvar` are used as x, y and z variables; if the argument is a list containing components `x`, `y` and `z`, these are assumed to define plotting coordinates; if the argument is a matrix or `data.frame` with three or more columns, the first is assumed to contain the x values, the 2nd the y ones, and the 3rd the z ones – independently of any column names that `x` may have. Alternatively two arguments `x` and `y` can be provided (leaving `z = NULL`). One may be real, the other complex; in any other case, the arguments are coerced to vectors and the values plotted against their indices.
`xlab, ylab, zlab`	names for the x, y and z variables to be extracted.
`log`	character, `"x"`, `"y"`, `"z"` or combinations. Sets negative values to `NA` and gives a warning.
`recycle`	logical; if `TRUE`, recycle (`rep`) the shorter ones of `x`, `y` or `z` if their lengths differ.

Value

A list with the components

`x`	numeric (i.e., `double`) vector of abscissa values.
`y`	numeric vector of the same length as `x`.
`z`	numeric vector of the same length as `x`.
`xlab`	`character(1)` or `NULL`, the axis label of `x`.
`ylab`	`character(1)` or `NULL`, the axis label of `y`.
`zlab`	`character(1)` or `NULL`, the axis label of `z`.

Author(s)

Uwe Ligges and Martin Maechler

See Also

`xy.coords` for 2D.

Examples

```
xyz.coords(data.frame(10*1:9, -4), y = NULL, z = NULL)

xyz.coords(1:6, stats::fft(1:6), z = NULL, xlab = "X", ylab = "Y")

y <- 2 * (x2 <- 10 + (x1 <- 1:10))
xyz.coords(y ~ x1 + x2, y = NULL, z = NULL)

xyz.coords(data.frame(x = -1:9, y = 2:12, z = 3:13), y = NULL, z = NULL,
           log = "xy")
##> Warning message: 2 x values <= 0 omitted ...
```

Chapter 4

The `graphics` package

`graphics-package` *The R Graphics Package*

Description

R functions for base graphics

Details

This package contains functions for base graphics. Base graphics are traditional S graphics, as opposed to the newer grid graphics.

For a complete list of functions with individual help pages, use `library(help="graphics")`.

Author(s)

R Development Core Team and contributors worldwide

Maintainer: R Core Team ⟨R-core@r-project.org⟩

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

`abline` *Add Straight Lines to a Plot*

Description

This function adds one or more straight lines through the current plot.

Usage

```
abline(a = NULL, b = NULL, h = NULL, v = NULL, reg = NULL,
       coef = NULL, untf = FALSE, ...)
```

Arguments

a, b	the intercept and slope, single values.
untf	logical asking whether to *untransform*. See 'Details'.
h	the y-value(s) for horizontal line(s).
v	the x-value(s) for vertical line(s).
coef	a vector of length two giving the intercept and slope.
reg	an object with a coef method. See 'Details'.
...	graphical parameters such as col, lty and lwd (possibly as vectors: see 'Details') and the line characteristics lend, ljoin and lmitre.

Details

Typical usages are

```
abline(a, b, untf = FALSE, ...)
abline(h=, untf = FALSE, ...)
abline(v=, untf = FALSE, ...)
abline(coef=, untf = FALSE, ...)
abline(reg=, untf = FALSE, ...)
```

The first form specifies the line in intercept/slope form (alternatively a can be specified on its own and is taken to contain the slope and intercept in vector form).

The h= and v= forms draw horizontal and vertical lines at the specified coordinates.

The coef form specifies the line by a vector containing the slope and intercept.

reg is a regression object with a coef method. If this returns a vector of length 1 then the value is taken to be the slope of a line through the origin, otherwise, the first 2 values are taken to be the intercept and slope.

If untf is true, and one or both axes are log-transformed, then a curve is drawn corresponding to a line in original coordinates, otherwise a line is drawn in the transformed coordinate system. The h and v parameters always refer to original coordinates.

The graphical parameters col, lty and lwd can be specified; see par for details. For the h= and v= usages they can be vectors of length greater than one, recycled as necessary.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

lines and segments for connected and arbitrary lines given by their *endpoints*. par.

Examples

```
## Setup up coordinate system (with x==y aspect ratio):
plot(c(-2,3), c(-1,5), type = "n", xlab="x", ylab="y", asp = 1)
## the x- and y-axis, and an integer grid
abline(h=0, v=0, col = "gray60")
text(1,0, "abline( h = 0 )", col = "gray60", adj = c(0, -.1))
```

```
abline(h = -1:5, v = -2:3, col = "lightgray", lty=3)
abline(a=1, b=2, col = 2)
text(1,3, "abline( 1, 2 )", col=2, adj=c(-.1,-.1))

## Simple Regression Lines:
require(stats)
sale5 <- c(6, 4, 9, 7, 6, 12, 8, 10, 9, 13)
plot(sale5)
abline(lsfit(1:10,sale5))
abline(lsfit(1:10,sale5, intercept = FALSE), col= 4) # less fitting

z <- lm(dist ~ speed, data = cars)
plot(cars)
abline(z) # equivalent to abline(reg = z) or
abline(coef = coef(z))

## trivial intercept model
abline(mC <- lm(dist ~ 1, data = cars)) ## the same as
abline(a = coef(mC), b = 0, col = "blue")
```

`arrows` *Add Arrows to a Plot*

Description

Draw arrows between pairs of points.

Usage

```
arrows(x0, y0, x1, y1, length = 0.25, angle = 30, code = 2,
       col = par("fg"), lty = par("lty"), lwd = par("lwd"),
       ...)
```

Arguments

`x0, y0`	coordinates of points **from** which to draw.
`x1, y1`	coordinates of points **to** which to draw.
`length`	length of the edges of the arrow head (in inches).
`angle`	angle from the shaft of the arrow to the edge of the arrow head.
`code`	integer code, determining *kind* of arrows to be drawn.
`col, lty, lwd`	graphical parameters, possible vectors. `NA` values in `col` cause the arrow to be omitted.
`...`	graphical parameters such as `xpd` and the line characteristics `lend`, `ljoin` and `lmitre`: see `par`.

Details

For each `i`, an arrow is drawn between the point `(x0[i], y0[i])` and the point `(x1[i],y1[i])`. The coordinate vectors will be recycled to the length of the longest.

If `code=1` an arrowhead is drawn at `(x0[i],y0[i])` and if `code=2` an arrowhead is drawn at `(x1[i],y1[i])`. If `code=3` a head is drawn at both ends of the arrow. Unless `length = 0`, when no head is drawn.

The graphical parameters `col`, `lty` and `lwd` can be vectors of length greater than one and will be recycled if necessary.

The direction of a zero-length arrow is indeterminate, and hence so is the direction of the arrowheads. To allow for rounding error, arrowheads are omitted (with a warning) on any arrow of length less than 1/1000 inch.

Note

The first four arguments in the comparable S function are named `x1,y1,x2,y2`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`segments` to draw segments.

Examples

```
x <- stats::runif(12); y <- stats::rnorm(12)
i <- order(x,y); x <- x[i]; y <- y[i]
plot(x,y, main="arrows(.) and segments(.)")
## draw arrows from point to point :
s <- seq(length(x)-1)# one shorter than data
arrows(x[s], y[s], x[s+1], y[s+1], col= 1:3)
s <- s[-length(s)]
segments(x[s], y[s], x[s+2], y[s+2], col= 'pink')
```

---

assocplot *Association Plots*

Description

Produce a Cohen-Friendly association plot indicating deviations from independence of rows and columns in a 2-dimensional contingency table.

Usage

```
assocplot(x, col = c("black", "red"), space = 0.3,
          main = NULL, xlab = NULL, ylab = NULL)
```

Arguments

x	a two-dimensional contingency table in matrix form.
col	a character vector of length two giving the colors used for drawing positive and negative Pearson residuals, respectively.
space	the amount of space (as a fraction of the average rectangle width and height) left between each rectangle.
main	overall title for the plot.
xlab	a label for the x axis. Defaults to the name (if any) of the row dimension in x.
ylab	a label for the y axis. Defaults to the name (if any) of the column dimension in x.

Details

For a two-way contingency table, the signed contribution to Pearson's χ^2 for cell i, j is $d_{ij} = (f_{ij} - e_{ij})/\sqrt{e_{ij}}$, where f_{ij} and e_{ij} are the observed and expected counts corresponding to the cell. In the Cohen-Friendly association plot, each cell is represented by a rectangle that has (signed) height proportional to d_{ij} and width proportional to $\sqrt{e_{ij}}$, so that the area of the box is proportional to the difference in observed and expected frequencies. The rectangles in each row are positioned relative to a baseline indicating independence ($d_{ij} = 0$). If the observed frequency of a cell is greater than the expected one, the box rises above the baseline and is shaded in the color specified by the first element of col, which defaults to black; otherwise, the box falls below the baseline and is shaded in the color specified by the second element of col, which defaults to red.

A more flexible and extensible implementation of association plots written in the grid graphics system is provided in the function assoc in the contributed package **vcd** (Meyer, Zeileis and Hornik, 2005).

References

Cohen, A. (1980), On the graphical display of the significant components in a two-way contingency table. *Communications in Statistics—Theory and Methods*, **A9**, 1025–1041.

Friendly, M. (1992), Graphical methods for categorical data. *SAS User Group International Conference Proceedings*, **17**, 190–200. http://www.math.yorku.ca/SCS/sugi/sugi17-paper.html

Meyer, D., Zeileis, A., and Hornik, K. (2005) The strucplot framework: Visualizing multi-way contingency tables with vcd. *Report 22*, Department of Statistics and Mathematics, Wirtschaftsuniversität Wien, Research Report Series. http://epub.wu-wien.ac.at/dyn/openURL?id=oai:epub.wu-wien.ac.at:epub-wu-01_8a1

See Also

mosaicplot, chisq.test.

Examples

```
## Aggregate over sex:
x <- margin.table(HairEyeColor, c(1, 2))
x
assocplot(x, main = "Relation between hair and eye color")
```

Axis *Generic function to add an Axis to a Plot*

Description

Generic function to add a suitable axis to the current plot.

Usage

```
Axis(x = NULL, at = NULL, ..., side, labels = NULL)
```

Arguments

x	an object which indicates the range over which an axis should be drawn
at	the points at which tick-marks are to be drawn.
side	an integer specifying which side of the plot the axis is to be drawn on. The axis is placed as follows: 1=below, 2=left, 3=above and 4=right.
labels	this can either be a logical value specifying whether (numerical) annotations are to be made at the tickmarks, or a character or expression vector of labels to be placed at the tickpoints. If this is specified as a character or expression vector, at should be supplied and they should be the same length.
...	Arguments to be passed to methods and perhaps then to axis.

Details

This is a generic function. It works in a slightly non-standard way: if x is supplied and non-NULL it dispatches on x, otherwise if at is supplied and non-NULL it dispatches on at, and the default action is to call axis, omitting argument x.

The idea is that for plots for which either or both of the axes are numerical but with a special interpretation, the standard plotting functions (including boxplot, contour, coplot, filled.contour, pairs, plot.default, rug and stripchart) will set up user coordinates and Axis will be called to label them appropriately.

There are "Date", "POSIXct" and "POSIXlt" methods which can pass an argument format onto the appropriate axis method (see axis.POSIXct).

Value

The numeric locations on the axis scale at which tick marks were drawn when the plot was first drawn (see 'Details').

This function is usually invoked for its side effect, which is to add an axis to an already existing plot.

See Also

axis.

`axis` *Add an Axis to a Plot*

Description

Adds an axis to the current plot, allowing the specification of the side, position, labels, and other options.

Usage

```
axis(side, at = NULL, labels = TRUE, tick = TRUE, line = NA,
     pos = NA, outer = FALSE, font = NA, lty = "solid",
     lwd = 1, lwd.ticks = lwd, col = NULL, col.ticks = NULL,
     hadj = NA, padj = NA, ...)
```

Arguments

`side`	an integer specifying which side of the plot the axis is to be drawn on. The axis is placed as follows: 1=below, 2=left, 3=above and 4=right.
`at`	the points at which tick-marks are to be drawn. Non-finite (infinite, `NaN` or `NA`) values are omitted. By default (when `NULL`) tickmark locations are computed, see 'Details' below.
`labels`	this can either be a logical value specifying whether (numerical) annotations are to be made at the tickmarks, or a character or expression vector of labels to be placed at the tickpoints. (Other objects are coerced by `as.graphicsAnnot`.) If this is not logical, `at` should also be supplied and of the same length. If `labels` is of length zero after coercion, it has the same effect as supplying `TRUE`.
`tick`	a logical value specifying whether tickmarks and an axis line should be drawn
`line`	the number of lines into the margin at which the axis line will be drawn, if not `NA`.
`pos`	the coordinate at which the axis line is to be drawn: if not `NA` this overrides the value of `line`.
`outer`	a logical value indicating whether the axis should be drawn in the outer plot margin, rather than the standard plot margin.
`font`	font for text. Defaults to `par("font")`.
`lty`	line type for both the axis line and the tick marks.
`lwd, lwd.ticks`	line widths for the axis line and the tick marks. Zero or negative values will suppress the line or ticks.
`col, col.ticks`	colors for the axis line and the tick marks respectively. `col = NULL` means to use `par("fg")`, possibly specified inline, and `col=NULL` means to use whatever color `col` resolved to.
`hadj`	adjustment (see `par("adj")`) for all labels *parallel* ('horizontal') to the reading direction. If this is not a finite value, the default is used (centring for strings parallel to the axis, justification of the end nearest the axis otherwise).

padj — adjustment for each tick label *perpendicular* to the reading direction. For labels parallel to the axes, padj=0 means right or top alignment, and padj=1 means left or bottom alignment. This can be a vector given a value for each string, and will be recycled as necessary.

If padj is not a finite value (the default), the value of par("las") determines the adjustment. For strings plotted perpendicular to the axis the default is to centre the string.

... — other graphical parameters may also be passed as arguments to this function, particularly, cex.axis, col.axis and font.axis for axis annotation, mgp and xaxp or yaxp for positioning, tck or tcl for tick mark length and direction, las for vertical/horizontal label orientation, or fg instead of col, and xpd for clipping. See par on these.

Parameters xaxt (sides 1 and 3) and yaxt (sides 2 and 4) control if the axis is plotted at all.

Note that lab will partial match to argument labels unless the latter is also supplied. (Since the default axes have already been set up by plot.window, lab will not be acted on by axis.)

Details

The axis line is drawn from the lowest to the highest value of at, but will be clipped at the plot region. By default, only ticks which are drawn from points within the plot region (up to a tolerance for rounding error) are plotted, but the ticks and their labels may well extend outside the plot region. Use xpd=TRUE or xpd=NA to allow axes to extend further.

When at = NULL, pretty tick mark locations are computed internally (the same way axTicks(side) would) from par("xaxp") or "yaxp" and par("xlog") (or "ylog"). Note that these locations may change if an on-screen plot is resized (for example, if the plot argument asp (see plot.window) is set.)

If labels is not specified, the numeric values supplied or calculated for at are converted to character strings as if they were a numeric vector printed by print.default(digits=7).

The code tries hard not to draw overlapping tick labels, and so will omit labels where they would abut or overlap previously drawn labels. This can result in, for example, every other tick being labelled. (The ticks are drawn left to right or bottom to top, and space at least the size of an 'm' is left between labels.)

If either line or pos is set, they (rather than par("mgp")[3]) determine the position of the axis line and tick marks, and the tick labels are placed par("mgp")[2] further lines into (or towards for pos) the margin.

Several of the graphics parameters affect the way axes are drawn. The vertical (for sides 1 and 3) positions of the axis and the tick labels are controlled by mgp[2:3] and mex, the size and direction of the ticks is controlled by tck and tcl and the appearance of the tick labels by cex.axis, col.axis and font.axis with orientation controlled by las (but not srt, unlike S which uses srt if at is supplied and las if it is not). Note that adj is not supported and labels are always centered. See par for details.

Value

The numeric locations on the axis scale at which tick marks were drawn when the plot was first drawn (see 'Details').

This function is usually invoked for its side effect, which is to add an axis to an already existing plot.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`Axis` for a generic interface.

`axTicks` returns the axis tick locations corresponding to `at=NULL`; `pretty` is more flexible for computing pretty tick coordinates and does *not* depend on (nor adapt to) the coordinate system in use.

Several graphics parameters affecting the appearance are documented in `par`.

Examples

```
require(stats) # for rnorm
plot(1:4, rnorm(4), axes = FALSE)
axis(1, 1:4, LETTERS[1:4])
axis(2)
box() #- to make it look "as usual"

plot(1:7, rnorm(7), main = "axis() examples",
     type = "s", xaxt = "n", frame = FALSE, col = "red")
axis(1, 1:7, LETTERS[1:7], col.axis = "blue")
# unusual options:
axis(4, col = "violet", col.axis="dark violet", lwd = 2)
axis(3, col = "gold", lty = 2, lwd = 0.5)

# one way to have a custom x axis
plot(1:10, xaxt = "n")
axis(1, xaxp=c(2, 9, 7))
```

`axis.POSIXct` *Date and Date-time Plotting Functions*

Description

Functions to plot objects of classes `"POSIXlt"`, `"POSIXct"` and `"Date"` representing calendar dates and times.

Usage

```
axis.POSIXct(side, x, at, format, labels = TRUE, ...)
axis.Date(side, x, at, format, labels = TRUE, ...)

## S3 method for class 'POSIXct':
plot(x, y, xlab = "", ...)
## S3 method for class 'POSIXlt':
plot(x, y, xlab = "", ...)
## S3 method for class 'Date':
plot(x, y, xlab = "", ...)
```

Arguments

x, at	A date-time or date object.
y	numeric values to be plotted against x.
xlab	a character string giving the label for the x axis.
side	See axis.
format	See strptime.
labels	Either a logical value specifying whether annotations are to be made at the tick-marks, or a vector of character strings to be placed at the tickpoints.
...	Further arguments to be passed from or to other methods, typically graphical parameters or arguments of plot.default. For the plot methods, also format.

Details

The functions plot against an x-axis of date-times. axis.POSIXct and axis.Date work quite hard to choose suitable time units (years, months, days, hours, minutes or seconds) and a sensible output format, but this can be overridden by supplying a format specification.

If at is supplied it specifies the locations of the ticks and labels whereas if x is specified a suitable grid of labels is chosen. Printing of tick labels can be suppressed by using labels = FALSE.

Value

The locations on the axis scale at which tick marks were drawn.

See Also

DateTimeClasses, Dates for details of the classes.

Examples

```
with(beaver1, {
time <- strptime(paste(1990, day, time %/% 100, time %% 100),
                 "%Y %j %H %M")
plot(time, temp, type="l") # axis at 4-hour intervals.
# now label every hour on the time axis
plot(time, temp, type="l", xaxt="n")
r <- as.POSIXct(round(range(time), "hours"))
axis.POSIXct(1, at=seq(r[1], r[2], by="hour"), format="%H")
})

plot(.leap.seconds, 1:23, type="n", yaxt="n",
     xlab="leap seconds", ylab="", bty="n")
rug(.leap.seconds)
## or as dates
lps <- as.Date(.leap.seconds)
plot(lps, 1:23, type = "n", yaxt = "n", xlab = "leap seconds",
     ylab = "", bty = "n")
rug(lps)

## 100 random dates in a 10-week period
random.dates <- as.Date("2001/1/1") + 70*sort(stats::runif(100))
plot(random.dates, 1:100)
# or for a better axis labelling
```

```
plot(random.dates, 1:100, xaxt="n")
axis.Date(1, at=seq(as.Date("2001/1/1"), max(random.dates)+6, "weeks"))
axis.Date(1, at=seq(as.Date("2001/1/1"), max(random.dates)+6, "days"),
     labels = FALSE, tcl = -0.2)
```

`axTicks` *Compute Axis Tickmark Locations*

Description

Compute pretty tickmark locations, the same way as R does internally. This is only non-trivial when **log** coordinates are active. By default, gives the `at` values which `axis(side)` would use.

Usage

```
axTicks(side, axp = NULL, usr = NULL, log = NULL)
```

Arguments

`side`	integer in 1:4, as for `axis`.
`axp`	numeric vector of length three, defaulting to `par("xaxp")` or `par("yaxp")` depending on the `side` argument.
`usr`	numeric vector of length four, defaulting to `par("usr")` giving horizontal ('x') and vertical ('y') user coordinate limits.
`log`	logical indicating if log coordinates are active; defaults to `par("xlog")` or `par("ylog")`.

Details

The `axp`, `usr`, and `log` arguments must be consistent as their default values (the `par(..)` results) are. If you specify all three (as non-NULL), the graphics environment is not used at all. Note that the meaning of `axp` alters very much when `log` is `TRUE`, see the documentation on `par(xaxp=.)`.

`axTicks()` can be regarded as an R implementation of the C function `CreateAtVector()` in '..../src/main/plot.c' which is called by `axis(side,*)` when no argument `at` is specified.

Value

numeric vector of coordinate values at which axis tickmarks can be drawn. By default, when only the first argument is specified, these values should be identical to those that `axis(side)` would use or has used.

See Also

`axis`, `par`. `pretty` uses the same algorithm (but independently of the graphics environment) and has more options. However it is not available for `log = TRUE`.

Examples

```
 plot(1:7, 10*21:27)
 axTicks(1)
 axTicks(2)
 stopifnot(identical(axTicks(1), axTicks(3)),
           identical(axTicks(2), axTicks(4)))

## Show how axTicks() and axis() correspond :
op <- par(mfrow = c(3,1))
for(x in 9999*c(1,2,8)) {
    plot(x,9, log = "x")
    cat(formatC(par("xaxp"), width=5),";", T <- axTicks(1),"\n")
    rug(T, col="red")
}
par(op)
```

---

barplot *Bar Plots*

Description

Creates a bar plot with vertical or horizontal bars.

Usage

```
barplot(height, ...)

## Default S3 method:
barplot(height, width = 1, space = NULL,
        names.arg = NULL, legend.text = NULL, beside = FALSE,
        horiz = FALSE, density = NULL, angle = 45,
        col = NULL, border = par("fg"),
        main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
        xlim = NULL, ylim = NULL, xpd = TRUE, log = "",
        axes = TRUE, axisnames = TRUE,
        cex.axis = par("cex.axis"), cex.names = par("cex.axis"),
        inside = TRUE, plot = TRUE, axis.lty = 0, offset = 0,
        add = FALSE, ...)
```

Arguments

`height`	either a vector or matrix of values describing the bars which make up the plot. If `height` is a vector, the plot consists of a sequence of rectangular bars with heights given by the values in the vector. If `height` is a matrix and `beside` is `FALSE` then each bar of the plot corresponds to a column of `height`, with the values in the column giving the heights of stacked sub-bars making up the bar. If `height` is a matrix and `beside` is `TRUE`, then the values in each column are juxtaposed rather than stacked.
`width`	optional vector of bar widths. Re-cycled to length the number of bars drawn. Specifying a single value will have no visible effect unless `xlim` is specified.

space	the amount of space (as a fraction of the average bar width) left before each bar. May be given as a single number or one number per bar. If height is a matrix and beside is TRUE, space may be specified by two numbers, where the first is the space between bars in the same group, and the second the space between the groups. If not given explicitly, it defaults to c(0,1) if height is a matrix and beside is TRUE, and to 0.2 otherwise.
names.arg	a vector of names to be plotted below each bar or group of bars. If this argument is omitted, then the names are taken from the names attribute of height if this is a vector, or the column names if it is a matrix.
legend.text	a vector of text used to construct a legend for the plot, or a logical indicating whether a legend should be included. This is only useful when height is a matrix. In that case given legend labels should correspond to the rows of height; if legend.text is true, the row names of height will be used as labels if they are non-null.
beside	a logical value. If FALSE, the columns of height are portrayed as stacked bars, and if TRUE the columns are portrayed as juxtaposed bars.
horiz	a logical value. If FALSE, the bars are drawn vertically with the first bar to the left. If TRUE, the bars are drawn horizontally with the first at the bottom.
density	a vector giving the density of shading lines, in lines per inch, for the bars or bar components. The default value of NULL means that no shading lines are drawn. Non-positive values of density also inhibit the drawing of shading lines.
angle	the slope of shading lines, given as an angle in degrees (counter-clockwise), for the bars or bar components.
col	a vector of colors for the bars or bar components. By default, grey is used if height is a vector, and a gamma-corrected grey palette if height is a matrix.
border	the color to be used for the border of the bars. Use border = NA to omit borders. If there are shading lines, border = TRUE means use the same colour for the border as for the shading lines.
main,sub	overall and sub title for the plot.
xlab	a label for the x axis.
ylab	a label for the y axis.
xlim	limits for the x axis.
ylim	limits for the y axis.
xpd	logical. Should bars be allowed to go outside region?
log	string specifying if axis scales should be logarithmic; see plot.default.
axes	logical. If TRUE, a vertical (or horizontal, if horiz is true) axis is drawn.
axisnames	logical. If TRUE, and if there are names.arg (see above), the other axis is drawn (with lty=0) and labeled.
cex.axis	expansion factor for numeric axis labels.
cex.names	expansion factor for axis names (bar labels).
inside	logical. If TRUE, the lines which divide adjacent (non-stacked!) bars will be drawn. Only applies when space = 0 (which it partly is when beside = TRUE).
plot	logical. If FALSE, nothing is plotted.
axis.lty	the graphics parameter lty applied to the axis and tick marks of the categorical (default horizontal) axis. Note that by default the axis is suppressed.

offset	a vector indicating how much the bars should be shifted relative to the x axis.
add	logical specifying if bars should be added to an already existing plot; defaults to FALSE.
...	arguments to be passed to/from other methods. For the default method these can include further arguments (such as axes, asp and main) and graphical parameters (see par) which are passed to plot.window(), title() and axis.

Details

This is a generic function, it currently only has a default method. A formula interface may be added eventually.

Value

A numeric vector (or matrix, when beside = TRUE), say mp, giving the coordinates of *all* the bar midpoints drawn, useful for adding to the graph.

If beside is true, use colMeans(mp) for the midpoints of each *group* of bars, see example.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

plot(..., type="h"), dotchart, hist.

Examples

```
require(grDevices) # for colours
tN <- table(Ni <- stats::rpois(100, lambda=5))
r <- barplot(tN, col=rainbow(20))
#- type = "h" plotting *is* 'bar'plot
lines(r, tN, type='h', col='red', lwd=2)

barplot(tN, space = 1.5, axisnames=FALSE,
        sub = "barplot(..., space= 1.5, axisnames = FALSE)")

barplot(VADeaths, plot = FALSE)
barplot(VADeaths, plot = FALSE, beside = TRUE)

mp <- barplot(VADeaths) # default
tot <- colMeans(VADeaths)
text(mp, tot + 3, format(tot), xpd = TRUE, col = "blue")
barplot(VADeaths, beside = TRUE,
        col = c("lightblue", "mistyrose", "lightcyan",
                "lavender", "cornsilk"),
        legend = rownames(VADeaths), ylim = c(0, 100))
title(main = "Death Rates in Virginia", font.main = 4)

hh <- t(VADeaths)[, 5:1]
mybarcol <- "gray20"
```

```
mp <- barplot(hh, beside = TRUE,
        col = c("lightblue", "mistyrose",
                "lightcyan", "lavender"),
        legend = colnames(VADeaths), ylim= c(0,100),
        main = "Death Rates in Virginia", font.main = 4,
        sub = "Faked upper 2*sigma error bars", col.sub = mybarcol,
        cex.names = 1.5)
segments(mp, hh, mp, hh + 2*sqrt(1000*hh/100), col = mybarcol, lwd = 1.5)
stopifnot(dim(mp) == dim(hh))# corresponding matrices
mtext(side = 1, at = colMeans(mp), line = -2,
      text = paste("Mean", formatC(colMeans(hh))), col = "red")

# Bar shading example
barplot(VADeaths, angle = 15+10*1:5, density = 20, col = "black",
        legend = rownames(VADeaths))
title(main = list("Death Rates in Virginia", font = 4))

# border :
barplot(VADeaths, border = "dark blue")

# log scales (not much sense here):
barplot(tN, col=heat.colors(12), log = "y")
barplot(tN, col=gray.colors(20), log = "xy")
```

`box` *Draw a Box around a Plot*

Description

This function draws a box around the current plot in the given color and linetype. The `bty` parameter determines the type of box drawn. See `par` for details.

Usage

```
box(which = "plot", lty = "solid", ...)
```

Arguments

`which`	character, one of `"plot"`, `"figure"`, `"inner"` and `"outer"`.
`lty`	line type of the box.
`...`	further graphical parameters, such as `bty`, `col`, or `lwd`, see `par`. Note that `xpd` is not accepted as clipping is always to the device region.

Details

The choice of colour is complicated. If `col` was supplied and is not `NA`, it is used. Otherwise, if `fg` was supplied and is not `NA`, it is used. The final default is `par("col")`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

rect for drawing of arbitrary rectangles.

Examples

```
plot(1:7, abs(stats::rnorm(7)), type = 'h', axes = FALSE)
axis(1, at = 1:7, labels = letters[1:7])
box(lty = '1373', col = 'red')
```

---

boxplot *Box Plots*

Description

Produce box-and-whisker plot(s) of the given (grouped) values.

Usage

```
boxplot(x, ...)

## S3 method for class 'formula':
boxplot(formula, data = NULL, ..., subset, na.action = NULL)

## Default S3 method:
boxplot(x, ..., range = 1.5, width = NULL, varwidth = FALSE,
        notch = FALSE, outline = TRUE, names, plot = TRUE,
        border = par("fg"), col = NULL, log = "",
        pars = list(boxwex = 0.8, staplewex = 0.5, outwex = 0.5),
        horizontal = FALSE, add = FALSE, at = NULL)
```

Arguments

formula	a formula, such as y ~ grp, where y is a numeric vector of data values to be split into groups according to the grouping variable grp (usually a factor).
data	a data.frame (or list) from which the variables in formula should be taken.
subset	an optional vector specifying a subset of observations to be used for plotting.
na.action	a function which indicates what should happen when the data contain NAs. The default is to ignore missing values in either the response or the group.
x	for specifying data from which the boxplots are to be produced. Either a numeric vector, or a single list containing such vectors. Additional unnamed arguments specify further data as separate vectors (each corresponding to a component boxplot). NAs are allowed in the data.
...	For the formula method, named arguments to be passed to the default method. For the default method, unnamed arguments are additional data vectors (unless x is a list when they are ignored), and named arguments are arguments and graphical parameters to be passed to bxp in addition to the ones given by argument pars (and override those in pars).

`range`	this determines how far the plot whiskers extend out from the box. If `range` is positive, the whiskers extend to the most extreme data point which is no more than `range` times the interquartile range from the box. A value of zero causes the whiskers to extend to the data extremes.
`width`	a vector giving the relative widths of the boxes making up the plot.
`varwidth`	if `varwidth` is `TRUE`, the boxes are drawn with widths proportional to the square-roots of the number of observations in the groups.
`notch`	if `notch` is `TRUE`, a notch is drawn in each side of the boxes. If the notches of two plots do not overlap this is 'strong evidence' that the two medians differ (Chambers *et al.*, 1983, p. 62). See `boxplot.stats` for the calculations used.
`outline`	if `outline` is not true, the outliers are not drawn (as points whereas S+ uses lines).
`names`	group labels which will be printed under each boxplot. Can be a character vector or an expression (see plotmath).
`boxwex`	a scale factor to be applied to all boxes. When there are only a few groups, the appearance of the plot can be improved by making the boxes narrower.
`staplewex`	staple line width expansion, proportional to box width.
`outwex`	outlier line width expansion, proportional to box width.
`plot`	if `TRUE` (the default) then a boxplot is produced. If not, the summaries which the boxplots are based on are returned.
`border`	an optional vector of colors for the outlines of the boxplots. The values in `border` are recycled if the length of `border` is less than the number of plots.
`col`	if `col` is non-null it is assumed to contain colors to be used to colour the bodies of the box plots. By default they are in the background colour.
`log`	character indicating if x or y or both coordinates should be plotted in log scale.
`pars`	a list of (potentially many) more graphical parameters, e.g., `boxwex` or `outpch`; these are passed to `bxp` (if `plot` is true); for details, see there.
`horizontal`	logical indicating if the boxplots should be horizontal; default `FALSE` means vertical boxes.
`add`	logical, if true *add* boxplot to current plot.
`at`	numeric vector giving the locations where the boxplots should be drawn, particularly when `add = TRUE`; defaults to `1:n` where `n` is the number of boxes.

Details

The generic function `boxplot` currently has a default method (`boxplot.default`) and a formula interface (`boxplot.formula`).

If multiple groups are supplied either as multiple arguments or via a formula, parallel boxplots will be plotted, in the order of the arguments or the order of the levels of the factor (see `factor`).

Missing values are ignored when forming boxplots.

Value

List with the following components:

`stats`	a matrix, each column contains the extreme of the lower whisker, the lower hinge, the median, the upper hinge and the extreme of the upper whisker for one group/plot. If all the inputs have the same class attribute, so will this component.

n	a vector with the number of observations in each group.
conf	a matrix where each column contains the lower and upper extremes of the notch.
out	the values of any data points which lie beyond the extremes of the whiskers.
group	a vector of the same length as out whose elements indicate to which group the outlier belongs.
names	a vector of names for the groups.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See also boxplot.stats.

See Also

boxplot.stats which does the computation, bxp for the plotting and more examples; and stripchart for an alternative (with small data sets).

Examples

```
## boxplot on a formula:
boxplot(count ~ spray, data = InsectSprays, col = "lightgray")
# *add* notches (somewhat funny here):
boxplot(count ~ spray, data = InsectSprays,
        notch = TRUE, add = TRUE, col = "blue")

boxplot(decrease ~ treatment, data = OrchardSprays,
        log = "y", col = "bisque")

rb <- boxplot(decrease ~ treatment, data = OrchardSprays, col="bisque")
title("Comparing boxplot()s and non-robust mean +/- SD")

mn.t <- tapply(OrchardSprays$decrease, OrchardSprays$treatment, mean)
sd.t <- tapply(OrchardSprays$decrease, OrchardSprays$treatment, sd)
xi <- 0.3 + seq(rb$n)
points(xi, mn.t, col = "orange", pch = 18)
arrows(xi, mn.t - sd.t, xi, mn.t + sd.t,
       code = 3, col = "pink", angle = 75, length = .1)

## boxplot on a matrix:
mat <- cbind(Uni05 = (1:100)/21, Norm = rnorm(100),
             `5T` = rt(100, df = 5), Gam2 = rgamma(100, shape = 2))
boxplot(as.data.frame(mat),
        main = "boxplot(as.data.frame(mat), main = ...)")
par(las=1)# all axis labels horizontal
boxplot(as.data.frame(mat), main = "boxplot(*, horizontal = TRUE)",
        horizontal = TRUE)

## Using 'at = ' and adding boxplots -- example idea by Roger Bivand :

boxplot(len ~ dose, data = ToothGrowth,
```

```
          boxwex = 0.25, at = 1:3 - 0.2,
          subset = supp == "VC", col = "yellow",
          main = "Guinea Pigs' Tooth Growth",
          xlab = "Vitamin C dose mg",
          ylab = "tooth length",
          xlim = c(0.5, 3.5), ylim = c(0, 35), yaxs = "i")
boxplot(len ~ dose, data = ToothGrowth, add = TRUE,
          boxwex = 0.25, at = 1:3 + 0.2,
          subset = supp == "OJ", col = "orange")
legend(2, 9, c("Ascorbic acid", "Orange juice"),
       fill = c("yellow", "orange"))

## more examples in  help(bxp)
```

`bxp` *Draw Box Plots from Summaries*

Description

`bxp` draws box plots based on the given summaries in `z`. It is usually called from within `boxplot`, but can be invoked directly.

Usage

```
bxp(z, notch = FALSE, width = NULL, varwidth = FALSE,
    outline = TRUE, notch.frac = 0.5, log = "",
    border = par("fg"), pars = NULL, frame.plot = axes,
    horizontal = FALSE, add = FALSE, at = NULL, show.names = NULL,
    ...)
```

Arguments

`z`	a list containing data summaries to be used in constructing the plots. These are usually the result of a call to `boxplot`, but can be generated in any fashion.
`notch`	if `notch` is `TRUE`, a notch is drawn in each side of the boxes. If the notches of two plots do not overlap then the medians are significantly different at the 5 percent level.
`width`	a vector giving the relative widths of the boxes making up the plot.
`varwidth`	if `varwidth` is `TRUE`, the boxes are drawn with widths proportional to the square-roots of the number of observations in the groups.
`outline`	if `outline` is not true, the outliers are not drawn.
`notch.frac`	numeric in (0,1). When `notch=TRUE`, the fraction of the box width that the notches should use.
`border`	character or numeric (vector), the color of the box borders. Is recycled for multiple boxes. Is used as default for the `boxcol`, `medcol`, `whiskcol`, `staplecol`, and `outcol` options (see below).
`log`	character, indicating if any axis should be drawn in logarithmic scale, as in `plot.default`.
`frame.plot`	logical, indicating if a 'frame' (`box`) should be drawn; defaults to `TRUE`, unless `axes = FALSE` is specified.

`horizontal` logical indicating if the boxplots should be horizontal; default `FALSE` means vertical boxes.

`add` logical, if true *add* boxplot to current plot.

`at` numeric vector giving the locations where the boxplots should be drawn, particularly when `add = TRUE`; defaults to `1:n` where `n` is the number of boxes.

`show.names` Set to `TRUE` or `FALSE` to override the defaults on whether an x-axis label is printed for each group.

`pars, ...` graphical parameters (etc) can be passed as arguments to this function, either as a list (`pars`) or normally(`...`), see the following. (Those in `...` take precedence over those in `pars`.)

Currently, `yaxs` and `ylim` are used 'along the boxplot', i.e., vertically, when `horizontal` is false, and `xlim` horizontally. `xaxt`, `yaxt`, `las`, `cex.axis`, and `col.axis` are passed to `axis`, and `main`, `cex.main`, `col.main`, `sub`, `cex.sub`, `col.sub`, `xlab`, `ylab`, `cex.lab`, and `col.lab` are passed to `title`.

In addition, `axes` is accepted (see `plot.window`), with default `TRUE`.

The following arguments (or `pars` components) allow further customization of the boxplot graphics. Their defaults are typically determined from the non-prefixed version (e.g., `boxlty` from `lty`), either from the specified argument or `pars` component or the corresponding `par` one.

boxwex: a scale factor to be applied to all boxes. When there are only a few groups, the appearance of the plot can be improved by making the boxes narrower. The default depends on `at` and typically is 0.8.

staplewex, outwex: staple and outlier line width expansion, proportional to box width; both default to 0.5.

boxlty, boxlwd, boxcol, boxfill: box outline type, width, color, and fill color (which currently defaults to `col` and will in future default to `par("bg")`).

medlty, medlwd, medpch, medcex, medcol, medbg: median line type, line width, point character, point size expansion, color, and background color. The default `medpch= NA` suppresses the point, and `medlty="blank"` does so for the line. Note that`medlwd` defaults to 3× the default `lwd`.

whisklty, whisklwd, whiskcol: whisker line type (default: `"dashed"`), width, and color.

staplelty, staplelwd, staplecol: staple (= end of whisker) line type, width, and color.

outlty, outlwd, outpch, outcex, outcol, outbg: outlier line type, line width, point character, point size expansion, color, and background color. The default `outlty= "blank"` suppresses the lines and `outpch=NA` suppresses points.

Value

An invisible vector, actually identical to the `at` argument, with the coordinates ("x" if horizontal is false, "y" otherwise) of box centers, useful for adding to the plot.

Note

if `add = FALSE`, the default is `xlim = c(0.5, n +0.5)`. It will usually be a good idea to specify the latter if the "x" axis has a log scale or `at` is specified or `width` is far from uniform.

Author(s)

The R Core development team and Arni Magnusson ⟨arnima@u.washington.edu⟩ who has provided most changes for the box*, med*, whisk*, staple*, and out* arguments.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
require(stats)
set.seed(753)
(bx.p <- boxplot(split(rt(100, 4), gl(5,20))))
op <- par(mfrow= c(2,2))
bxp(bx.p, xaxt = "n")
bxp(bx.p, notch = TRUE, axes = FALSE, pch = 4, boxfill=1:5)
bxp(bx.p, notch = TRUE, boxfill= "lightblue", frame= FALSE,
    outl= FALSE, main = "bxp(*, frame= FALSE, outl= FALSE)")
bxp(bx.p, notch = TRUE, boxfill= "lightblue", border= 2:6,
    ylim = c(-4,4), pch = 22, bg = "green", log = "x",
    main = "... log='x', ylim=*")
par(op)
op <- par(mfrow= c(1,2))

## single group -- no label
boxplot (weight ~ group, data = PlantGrowth, subset = group=="ctrl")
## with label
bx <- boxplot(weight ~ group, data = PlantGrowth,
              subset = group=="ctrl", plot = FALSE)
bxp(bx,show.names=TRUE)
par(op)

z <- split(rnorm(1000), rpois(1000,2.2))
boxplot(z, whisklty=3, main="boxplot(z, whisklty = 3)")

## Colour support similar to plot.default:
op <- par(mfrow=1:2, bg="light gray", fg="midnight blue")
boxplot(z,   col.axis="skyblue3", main="boxplot(*, col.axis=..,main=..)")
plot(z[[1]], col.axis="skyblue3", main=   "plot(*, col.axis=..,main=..)")
mtext("par(bg=\"light gray\", fg=\"midnight blue\")",
      outer = TRUE, line = -1.2)
par(op)

## Mimic S-Plus:
splus <- list(boxwex=0.4, staplewex=1, outwex=1, boxfill="grey40",
              medlwd=3, medcol="white", whisklty=3, outlty=1, outpch=NA)
boxplot(z, pars=splus)
## Recycled and "sweeping" parameters
op <- par(mfrow=c(1,2))
 boxplot(z, border=1:5, lty = 3, medlty = 1, medlwd = 2.5)
 boxplot(z, boxfill=1:3, pch=1:5, lwd = 1.5, medcol="white")
par(op)
## too many possibilities
boxplot(z, boxfill= "light gray", outpch = 21:25, outlty = 2,
        bg = "pink", lwd = 2,
```

```
          medcol = "dark blue", medcex = 2, medpch = 20)
```

`cdplot` *Conditional Density Plots*

Description

Computes and plots conditional densities describing how the conditional distribution of a categorical variable `y` changes over a numerical variable `x`.

Usage

```
cdplot(x, ...)

## Default S3 method:
cdplot(x, y,
  plot = TRUE, tol.ylab = 0.05, ylevels = NULL,
  bw = "nrd0", n = 512, from = NULL, to = NULL,
  col = NULL, border = 1, main = "", xlab = NULL, ylab = NULL,
  yaxlabels = NULL, xlim = NULL, ylim = c(0, 1), ...)

## S3 method for class 'formula':
cdplot(formula, data = list(),
  plot = TRUE, tol.ylab = 0.05, ylevels = NULL,
  bw = "nrd0", n = 512, from = NULL, to = NULL,
  col = NULL, border = 1, main = "", xlab = NULL, ylab = NULL,
  yaxlabels = NULL, xlim = NULL, ylim = c(0, 1), ...,
  subset = NULL)
```

Arguments

`x`	an object, the default method expects either a single numerical variable.
`y`	a `"factor"` interpreted to be the dependent variable
`formula`	a `"formula"` of type `y ~ x` with a single dependent `"factor"` and a single numerical explanatory variable.
`data`	an optional data frame.
`plot`	logical. Should the computed conditional densities be plotted?
`tol.ylab`	convenience tolerance parameter for y-axis annotation. If the distance between two labels drops under this threshold, they are plotted equidistantly.
`ylevels`	a character or numeric vector specifying in which order the levels of the dependent variable should be plotted.
`bw, n, from, to, ...`	arguments passed to `density`
`col`	a vector of fill colors of the same length as `levels(y)`. The default is to call `gray.colors`.
`border`	border color of shaded polygons.
`main, xlab, ylab`	character strings for annotation

`yaxlabels`	character vector for annotation of y axis, defaults to `levels(y)`.
`xlim, ylim`	the range of x and y values with sensible defaults.
`subset`	an optional vector specifying a subset of observations to be used for plotting.

Details

`cdplot` computes the conditional densities of `x` given the levels of `y` weighted by the marginal distribution of `y`. The densities are derived cumulatively over the levels of `y`.

This visualization technique is similar to spinograms (see `spineplot`) and plots $P(y|x)$ against x. The conditional probabilities are not derived by discretization (as in the spinogram), but using a smoothing approach via `density`.

Note, that the estimates of the conditional densities are more reliable for high-density regions of x. Conversely, the are less reliable in regions with only few x observations.

Value

The conditional density functions (cumulative over the levels of `y`) are returned invisibly.

Author(s)

Achim Zeileis ⟨Achim.Zeileis@R-project.org⟩

References

Hofmann, H., Theus, M. (2005), *Interactive graphics for visualizing conditional distributions*, Unpublished Manuscript.

See Also

`spineplot`, `density`

Examples

```
## NASA space shuttle o-ring failures
fail <- factor(c(2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1,
                 1, 2, 1, 1, 1, 1, 1),
               levels = 1:2, labels = c("no", "yes"))
temperature <- c(53, 57, 58, 63, 66, 67, 67, 67, 68, 69, 70, 70,
                 70, 70, 72, 73, 75, 75, 76, 76, 78, 79, 81)

## CD plot
cdplot(fail ~ temperature)
cdplot(fail ~ temperature, bw = 2)
cdplot(fail ~ temperature, bw = "SJ")

## compare with spinogram
(spineplot(fail ~ temperature, breaks = 3))

## highlighting for failures
cdplot(fail ~ temperature, ylevels = 2:1)

## scatter plot with conditional density
cdens <- cdplot(fail ~ temperature, plot = FALSE)
plot(I(as.numeric(fail) - 1) ~ jitter(temperature, factor = 2),
```

```
     xlab = "Temperature", ylab = "Conditional failure probability")
lines(53:81, 1 - cdens[[1]](53:81), col = 2)
```

`clip` *Set Clipping Region*

Description

Set clipping region in user coordinates

Usage

```
clip(x1, x2, y1, y2)
```

Arguments

`x1, x2, y1, y2`	user coordinates of clipping rectange

Details

How the clipping rectangle is set depends on the setting of `par("xpd")`: this function changes the current setting until the next high-level plotting command resets it.

Exactly when the clipping region will be reset can be hard to predict. `plot.new` always resets it. Functions such as `lines` and `text` only reset it if `par("xpd")` has been changed. However, functions such as `box`, `mtext`, `title` and `plot.dendrogram` can manipulate the `xpd` setting.

See Also

`par`

Examples

```
x <- rnorm(1000)
hist(x, xlim=c(-4,4))
usr <- par("usr")
clip(usr[1], -2, usr[3], usr[4])
hist(x, col = 'red', add = TRUE)
clip(2, usr[2], usr[3], usr[4])
hist(x, col = 'blue', add = TRUE)
do.call("clip", as.list(usr))  # reset to plot region
```

`contour` *Display Contours*

Description

Create a contour plot, or add contour lines to an existing plot.

Usage

```
contour(x, ...)

## Default S3 method:
contour(x = seq(0, 1, length.out = nrow(z)),
        y = seq(0, 1, length.out = ncol(z)),
        z,
        nlevels = 10, levels = pretty(zlim, nlevels),
        labels = NULL,
        xlim = range(x, finite = TRUE),
        ylim = range(y, finite = TRUE),
        zlim = range(z, finite = TRUE),
        labcex = 0.6, drawlabels = TRUE, method = "flattest",
        vfont, axes = TRUE, frame.plot = axes,
        col = par("fg"), lty = par("lty"), lwd = par("lwd"),
        add = FALSE, ...)
```

Arguments

`x,y`	locations of grid lines at which the values in `z` are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If `x` is a `list`, its components `x$x` and `x$y` are used for `x` and `y`, respectively. If the list has component `z` this is used for `z`.
`z`	a matrix containing the values to be plotted (`NA`s are allowed). Note that `x` can be used instead of `z` for convenience.
`nlevels`	number of contour levels desired **iff** `levels` is not supplied.
`levels`	numeric vector of levels at which to draw contour lines.
`labels`	a vector giving the labels for the contour lines. If `NULL` then the levels are used as labels, otherwise this is coerced by `as.character`.
`labcex`	`cex` for contour labelling. This is an absolute size, not a multiple of `par("cex")`.
`drawlabels`	logical. Contours are labelled if `TRUE`.
`method`	character string specifying where the labels will be located. Possible values are `"simple"`, `"edge"` and `"flattest"` (the default). See the 'Details' section.
`vfont`	if `NULL`, the current font family and face are used for the contour labels. If a character vector of length 2 then Hershey vector fonts are used for the contour labels. The first element of the vector selects a typeface and the second element selects a fontindex (see `text` for more information). The default is `NULL` on graphics devices with high-quality rotation of text and `c("sans serif", "plain")` otherwise.

`xlim, ylim, zlim`
: x-, y- and z-limits for the plot.

`axes, frame.plot`
: logical indicating whether axes or a box should be drawn, see `plot.default`.

`col`
: color for the lines drawn.

`lty`
: line type for the lines drawn.

`lwd`
: line width for the lines drawn.

`add`
: logical. If `TRUE`, add to a current plot.

`...`
: additional arguments to `plot.window`, `title`, `Axis` and `box`, typically graphical parameters such as `cex.axis`.

Details

`contour` is a generic function with only a default method in base R.

The methods for positioning the labels on contours are `"simple"` (draw at the edge of the plot, overlaying the contour line), `"edge"` (draw at the edge of the plot, embedded in the contour line, with no labels overlapping) and `"flattest"` (draw on the flattest section of the contour, embedded in the contour line, with no labels overlapping). The second and third may not draw a label on every contour line.

For information about vector fonts, see the help for `text` and `Hershey`.

Notice that `contour` interprets the `z` matrix as a table of `f(x[i], y[j])` values, so that the x axis corresponds to row number and the y axis to column number, with column 1 at the bottom, i.e. a 90 degree clockwise rotation of the conventional textual layout.

Alternatively, use `contourplot` from the **lattice** package where the `formula` notation allows to use vectors `x,y,z` of the same length.

There is limited control over the axes and frame as arguments `col`, `lwd` and `lty` refer to the contour lines (rather than being general graphical parameters). For more control, add contours to a plot, or add axes and frame to a contour plot.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`options("max.countour.segments")` for the maximal complexity of a single contour line.

`contourLines`, `filled.contour` for color-filled contours, `contourplot` (and `levelplot`) from package **lattice**. Further, `image` and the graphics demo which can be invoked as `demo(graphics)`.

Examples

```
require(grDevices) # for colours
x <- -6:16
op <- par(mfrow = c(2, 2))
contour(outer(x, x), method = "edge", vfont = c("sans serif", "plain"))
z <- outer(x, sqrt(abs(x)), FUN = "/")
image(x, x, z)
```

```
contour(x, x, z, col = "pink", add = TRUE, method = "edge",
        vfont = c("sans serif", "plain"))
contour(x, x, z, ylim = c(1, 6), method = "simple", labcex = 1)
contour(x, x, z, ylim = c(-6, 6), nlev = 20, lty = 2, method = "simple")
par(op)

## Persian Rug Art:
x <- y <- seq(-4*pi, 4*pi, len = 27)
r <- sqrt(outer(x^2, y^2, "+"))
opar <- par(mfrow = c(2, 2), mar = rep(0, 4))
for(f in pi^(0:3))
  contour(cos(r^2)*exp(-r/f),
          drawlabels = FALSE, axes = FALSE, frame = TRUE)

rx <- range(x <- 10*1:nrow(volcano))
ry <- range(y <- 10*1:ncol(volcano))
ry <- ry + c(-1,1) * (diff(rx) - diff(ry))/2
tcol <- terrain.colors(12)
par(opar); opar <- par(pty = "s", bg = "lightcyan")
plot(x = 0, y = 0,type = "n", xlim = rx, ylim = ry, xlab = "", ylab = "")
u <- par("usr")
rect(u[1], u[3], u[2], u[4], col = tcol[8], border = "red")
contour(x, y, volcano, col = tcol[2], lty = "solid", add = TRUE,
        vfont = c("sans serif", "plain"))
title("A Topographic Map of Maunga Whau", font = 4)
abline(h = 200*0:4, v = 200*0:4, col = "lightgray", lty = 2, lwd = 0.1)

## contourLines produces the same contour lines as contour
line.list <- contourLines(x, y, volcano)
plot(x = 0, y = 0,type = "n", xlim = rx, ylim = ry, xlab = "", ylab = "")
u <- par("usr")
rect(u[1], u[3], u[2], u[4], col = tcol[8], border = "red")
contour(x, y, volcano, col = tcol[2], lty = "solid", add = TRUE,
            vfont = c("sans serif", "plain"))
templines <- function(clines) {
  lines(clines[[2]], clines[[3]])
}
invisible(lapply(line.list, templines))
par(opar)
```

`convertXY` *Convert between Graphics Coordinate Systems*

Description

Convert between graphics coordinate systems.

Usage

```
grconvertX(x, from = "user", to = "user")
grconvertY(y, from = "user", to = "user")
```

Arguments

`x, y`	numeric vector of coordinates.
`from, to`	character strings giving the coordinate systems to convert between.

Details

The coordinate systems are

`"user"` user coordinates.

`"inches"` inches.

`"device"` the device coordinate system.

`"ndc"` normalized device coordinates.

`"nfc"` normalized figure coordinates.

`"npc"` normalized plot coordinates.

`"nic"` normalized inner region coordinates. (The 'inner region' is that inside the outer margins.)

(These names can be partially matched.) For the 'normalized' coordinate systems the lower left has value 0 and the top right value 1.

Device coordinates are those in which the device works: they are usually in pixels where that makes sense and in big points (1/72 inch) otherwise (e.g. `pdf` and `postscript`).

Value

A numeric vector of the same length as the input.

Examples

```
op <- par(omd=c(0.1, 0.9, 0.1, 0.9), mfrow = c(1, 2))
plot(1:4)
for(tp in c("in", "dev", "ndc", "nfc", "npc", "nic"))
    print(grconvertX(c(1.0, 4.0), "user", tp))
par(op)
```

`coplot` *Conditioning Plots*

Description

This function produces two variants of the **co**nditioning plots discussed in the reference below.

Usage

```
coplot(formula, data, given.values, panel = points, rows, columns,
       show.given = TRUE, col = par("fg"), pch = par("pch"),
       bar.bg = c(num = gray(0.8), fac = gray(0.95)),
       xlab = c(x.name, paste("Given :", a.name)),
       ylab = c(y.name, paste("Given :", b.name)),
       subscripts = FALSE,
       axlabels = function(f) abbreviate(levels(f)),
       number = 6, overlap = 0.5, xlim, ylim, ...)
co.intervals(x, number = 6, overlap = 0.5)
```

Arguments

formula	a formula describing the form of conditioning plot. A formula of the form `y ~ x | a` indicates that plots of `y` versus `x` should be produced conditional on the variable `a`. A formula of the form `y ~ x| a * b` indicates that plots of `y` versus `x` should be produced conditional on the two variables `a` and `b`. All three or four variables may be either numeric or factors. When `x` or `y` are factors, the result is almost as if `as.numeric()` was applied, whereas for factor `a` or `b`, the conditioning (and its graphics if `show.given` is true) are adapted.
data	a data frame containing values for any variables in the formula. By default the environment where `coplot` was called from is used.
given.values	a value or list of two values which determine how the conditioning on `a` and `b` is to take place. When there is no `b` (i.e., conditioning only on `a`), usually this is a matrix with two columns each row of which gives an interval, to be conditioned on, but is can also be a single vector of numbers or a set of factor levels (if the variable being conditioned on is a factor). In this case (no `b`), the result of `co.intervals` can be used directly as `given.values` argument.
panel	a `function(x, y, col, pch, ...)` which gives the action to be carried out in each panel of the display. The default is `points`.
rows	the panels of the plot are laid out in a `rows` by `columns` array. `rows` gives the number of rows in the array.
columns	the number of columns in the panel layout array.
show.given	logical (possibly of length 2 for 2 conditioning variables): should conditioning plots be shown for the corresponding conditioning variables (default `TRUE`)
col	a vector of colors to be used to plot the points. If too short, the values are recycled.
pch	a vector of plotting symbols or characters. If too short, the values are recycled.
bar.bg	a named vector with components `"num"` and `"fac"` giving the background colors for the (shingle) bars, for **num**eric and **fac**tor conditioning variables respectively.
xlab	character; labels to use for the x axis and the first conditioning variable. If only one label is given, it is used for the x axis and the default label is used for the conditioning variable.
ylab	character; labels to use for the y axis and any second conditioning variable.
subscripts	logical: if true the panel function is given an additional (third) argument `subscripts` giving the subscripts of the data passed to that panel.
axlabels	function for creating axis (tick) labels when x or y are factors.
number	integer; the number of conditioning intervals, for a and b, possibly of length 2. It is only used if the corresponding conditioning variable is not a `factor`.
overlap	numeric < 1; the fraction of overlap of the conditioning variables, possibly of length 2 for x and y direction. When overlap < 0, there will be *gaps* between the data slices.
xlim	the range for the x axis.
ylim	the range for the y axis.
...	additional arguments to the panel function.
x	a numeric vector.

Details

In the case of a single conditioning variable `a`, when both `rows` and `columns` are unspecified, a 'close to square' layout is chosen with `columns >= rows`.

In the case of multiple `rows`, the *order* of the panel plots is from the bottom and from the left (corresponding to increasing `a`, typically).

A panel function should not attempt to start a new plot, but just plot within a given coordinate system: thus `plot` and `boxplot` are not panel functions.

The rendering of arguments `xlab` and `ylab` is not controlled by `par` arguments `cex.lab` and `font.lab` even though they are plotted by `mtext` rather than `title`.

Value

`co.intervals(., number, .)` returns a (`number` $\times$ 2) `matrix`, say `ci`, where `ci[k,]` is the `range` of `x` values for the `k`-th interval.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Cleveland, W. S. (1993) *Visualizing Data.* New Jersey: Summit Press.

See Also

`pairs`, `panel.smooth`, `points`.

Examples

```
## Tonga Trench Earthquakes
coplot(lat ~ long | depth, data = quakes)
given.depth <- co.intervals(quakes$depth, number=4, overlap=.1)
coplot(lat ~ long | depth, data = quakes, given.v=given.depth, rows=1)

## Conditioning on 2 variables:
ll.dm <- lat ~ long | depth * mag
coplot(ll.dm, data = quakes)
coplot(ll.dm, data = quakes, number=c(4,7), show.given=c(TRUE,FALSE))
coplot(ll.dm, data = quakes, number=c(3,7),
       overlap=c(-.5,.1)) # negative overlap DROPS values

## given two factors
Index <- seq(length=nrow(warpbreaks)) # to get nicer default labels
coplot(breaks ~ Index | wool * tension, data = warpbreaks,
       show.given = 0:1)
coplot(breaks ~ Index | wool * tension, data = warpbreaks,
       col = "red", bg = "pink", pch = 21,
       bar.bg = c(fac = "light blue"))

## Example with empty panels:
with(data.frame(state.x77), {
coplot(Life.Exp ~ Income | Illiteracy * state.region, number = 3,
       panel = function(x, y, ...) panel.smooth(x, y, span = .8, ...))
## y ~ factor -- not really sensible, but 'show off':
coplot(Life.Exp ~ state.region | Income * state.division,
       panel = panel.smooth)
```

```
})
```

`curve` *Draw Function Plots*

Description

Draws a curve corresponding to the given function or, for `curve()` also an expression (in `x`) over the interval `[from,to]`.

Usage

```
curve(expr, from = NULL, to = NULL, n = 101, add = FALSE,
      type = "l", ylab = NULL, log = NULL, xlim = NULL, ...)

## S3 method for class 'function':
plot(x, y = 0, to = 1, from = y, xlim = NULL, ...)
```

Arguments

`expr`	an expression written as a function of `x`, or alternatively the name of a function which will be plotted.
`x`	a 'vectorizing' numeric R function.
`from,to`	the range over which the function will be plotted.
`n`	integer; the number of x values at which to evaluate.
`add`	logical; if `TRUE` add to already existing plot.
`xlim`	numeric of length 2; if specified, it serves as default for `c(from, to)`.
`type`	plot type: see `plot.default`.
`y`	alias for `from` for compatibility with `plot()`
`ylab, log, ...`	labels and graphical parameters can also be specified as arguments. `plot.function` passes all these to `curve`.

Details

The evaluation of `expr` is at `n` points equally spaced over the range `[from, to]`, possibly adapted to log scale. The points determined in this way are then joined with straight lines. `x(t)` or `expr` (with `x` inside) must return a numeric of the same length as the argument `t` or `x`.

For `curve()`, if either of `from` or `to` is `NULL`, it defaults to the corresponding element of `xlim`, and `xlim` defaults to the x-limits of the current plot. For `plot(<function>, ..)`, the defaults for $(from, to)$ are $(0, 1)$.

`log` is taken from the current plot only when `add` is true, and otherwise defaults to `""` indicating linear scales on both axes.

This used to be a quick hack which now seems to serve a useful purpose, but can give bad results for functions which are not smooth.

For expensive-to-compute `expr`essions, you should use smarter tools.

See Also

splinefun for spline interpolation, lines.

Examples

```
plot(qnorm)
plot(qlogis, main = "The Inverse Logit : qlogis()")
abline(h=0, v=0:2/2, lty=3, col="gray")

curve(sin, -2*pi, 2*pi)
curve(tan, main = "curve(tan)  --> same x-scale as previous plot")

op <- par(mfrow=c(2,2))
curve(x^3-3*x, -2, 2)
curve(x^2-2, add = TRUE, col = "violet")

## simple and sophisticated, quite similar:
plot(cos, -pi,  3*pi)
plot(cos, xlim = c(-pi,3*pi), n = 1001, col = "blue", add=TRUE)

chippy <- function(x) sin(cos(x)*exp(-x/2))
curve(chippy, -8, 7, n=2001)
plot (chippy, -8, -5)

for(ll in c("","x","y","xy"))
   curve(log(1+x), 1,100, log=ll, sub=paste("log= '",ll,"'",sep=""))
par(op)
```

---

dotchart *Cleveland Dot Plots*

Description

Draw a Cleveland dot plot.

Usage

```
dotchart(x, labels = NULL, groups = NULL, gdata = NULL,
         cex = par("cex"), pch = 21, gpch = 21, bg = par("bg"),
         color = par("fg"), gcolor = par("fg"), lcolor = "gray",
         xlim = range(x[is.finite(x)]),
         main = NULL, xlab = NULL, ylab = NULL, ...)
```

Arguments

x	either a vector or matrix of numeric values (NAs are allowed). If x is a matrix the overall plot consists of juxtaposed dotplots for each row.
labels	a vector of labels for each point. For vectors the default is to use names(x) and for matrices the row labels dimnames(x)[[1]].
groups	an optional factor indicating how the elements of x are grouped. If x is a matrix, groups will default to the columns of x.

gdata	data values for the groups. This is typically a summary such as the median or mean of each group.
cex	the character size to be used. Setting cex to a value smaller than one can be a useful way of avoiding label overlap. Unlike many other graphics functions, this sets the actual size, not a multiple of par("cex").
pch	the plotting character or symbol to be used.
gpch	the plotting character or symbol to be used for group values.
bg	the background color of plotting characters or symbols to be used; use par(bg= *) to set the background color of the whole plot.
color	the color(s) to be used for points and labels.
gcolor	the single color to be used for group labels and values.
lcolor	the color(s) to be used for the horizontal lines.
xlim	horizontal range for the plot, see plot.window, e.g.
main	overall title for the plot, see title.
xlab, ylab	axis annotations as in title.
...	graphical parameters can also be specified as arguments.

Value

This function is invoked for its side effect, which is to produce two variants of dotplots as described in Cleveland (1985).

Dot plots are a reasonable substitute for bar plots.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1985) *The Elements of Graphing Data.* Monterey, CA: Wadsworth.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

Examples

```
dotchart(VADeaths, main = "Death Rates in Virginia - 1940")
op <- par(xaxs="i")# 0 -- 100%
dotchart(t(VADeaths), xlim = c(0,100),
         main = "Death Rates in Virginia - 1940")
par(op)
```

---

filled.contour *Level (Contour) Plots*

Description

This function produces a contour plot with the areas between the contours filled in solid color (Cleveland calls this a level plot). A key showing how the colors map to z values is shown to the right of the plot.

Usage

```
filled.contour(x = seq(0, 1, length.out = nrow(z)),
               y = seq(0, 1, length.out = ncol(z)),
               z,
               xlim = range(x, finite=TRUE),
               ylim = range(y, finite=TRUE),
               zlim = range(z, finite=TRUE),
               levels = pretty(zlim, nlevels), nlevels = 20,
               color.palette = cm.colors,
               col = color.palette(length(levels) - 1),
               plot.title, plot.axes, key.title, key.axes,
               asp = NA, xaxs = "i", yaxs = "i", las = 1,
               axes = TRUE, frame.plot = axes, ...)
```

Arguments

x, y	locations of grid lines at which the values in z are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If x is a list, its components x$x and x$y are used for x and y, respectively. If the list has component z this is used for z.
z	a matrix containing the values to be plotted (NAs are allowed). Note that x can be used instead of z for convenience.
xlim	x limits for the plot.
ylim	y limits for the plot.
zlim	z limits for the plot.
levels	a set of levels which are used to partition the range of z. Must be **strictly** increasing (and finite). Areas with z values between consecutive levels are painted with the same color.
nlevels	if levels is not specified, the range of z, values is divided into approximately this many levels.
color.palette	a color palette function to be used to assign colors in the plot.
col	an explicit set of colors to be used in the plot. This argument overrides any palette function specification.
plot.title	statements which add titles to the main plot.
plot.axes	statements which draw axes (and a box) on the main plot. This overrides the default axes.
key.title	statements which add titles for the plot key.
key.axes	statements which draw axes on the plot key. This overrides the default axis.
asp	the y/x aspect ratio, see plot.window.
xaxs	the x axis style. The default is to use internal labeling.
yaxs	the y axis style. The default is to use internal labeling.
las	the style of labeling to be used. The default is to use horizontal labeling.
axes, frame.plot	logicals indicating if axes and a box should be drawn, as in plot.default.
...	additional graphical parameters, currently only passed to title().

Note

This function currently uses the `layout` function and so is restricted to a full page display. As an alternative consider the `levelplot` and `contourplot` functions from the **lattice** package which work in multipanel displays.

The output produced by `filled.contour` is actually a combination of two plots; one is the filled contour and one is the legend. Two separate coordinate systems are set up for these two plots, but they are only used internally - once the function has returned these coordinate systems are lost. If you want to annotate the main contour plot, for example to add points, you can specify graphics commands in the `plot.axes` argument. An example is given below.

Author(s)

Ross Ihaka.

References

Cleveland, W. S. (1993) *Visualizing Data.* Summit, New Jersey: Hobart.

See Also

`contour`, `image`, `palette`; `contourplot` from package **lattice**.

Examples

```
require(grDevices) # for colours
filled.contour(volcano, color = terrain.colors, asp = 1)# simple

x <- 10*1:nrow(volcano)
y <- 10*1:ncol(volcano)
filled.contour(x, y, volcano, color = terrain.colors,
    plot.title = title(main = "The Topography of Maunga Whau",
    xlab = "Meters North", ylab = "Meters West"),
    plot.axes = { axis(1, seq(100, 800, by = 100))
                  axis(2, seq(100, 600, by = 100)) },
    key.title = title(main="Height\n(meters)"),
    key.axes = axis(4, seq(90, 190, by = 10)))# maybe also asp=1
mtext(paste("filled.contour(.) from", R.version.string),
      side = 1, line = 4, adj = 1, cex = .66)

# Annotating a filled contour plot
a <- expand.grid(1:20, 1:20)
b <- matrix(a[,1] + a[,2], 20)
filled.contour(x = 1:20, y = 1:20, z = b,
               plot.axes={ axis(1); axis(2); points(10,10) })

## Persian Rug Art:
x <- y <- seq(-4*pi, 4*pi, len = 27)
r <- sqrt(outer(x^2, y^2, "+"))
filled.contour(cos(r^2)*exp(-r/(2*pi)), axes = FALSE)
## rather, the key *should* be labeled:
filled.contour(cos(r^2)*exp(-r/(2*pi)), frame.plot = FALSE,
               plot.axes = {})
```

`fourfoldplot` *Fourfold Plots*

Description

Creates a fourfold display of a 2 by 2 by k contingency table on the current graphics device, allowing for the visual inspection of the association between two dichotomous variables in one or several populations (strata).

Usage

```
fourfoldplot(x, color = c("#99CCFF", "#6699CC"),
             conf.level = 0.95,
             std = c("margins", "ind.max", "all.max"),
             margin = c(1, 2), space = 0.2, main = NULL,
             mfrow = NULL, mfcol = NULL)
```

Arguments

`x`	a 2 by 2 by k contingency table in array form, or as a 2 by 2 matrix if k is 1.
`color`	a vector of length 2 specifying the colors to use for the smaller and larger diagonals of each 2 by 2 table.
`conf.level`	confidence level used for the confidence rings on the odds ratios. Must be a single nonnegative number less than 1; if set to 0, confidence rings are suppressed.
`std`	a character string specifying how to standardize the table. Must be one of `"margins"`, `"ind.max"`, or `"all.max"`, and can be abbreviated by the initial letter. If set to `"margins"`, each 2 by 2 table is standardized to equate the margins specified by `margin` while preserving the odds ratio. If `"ind.max"` or `"all.max"`, the tables are either individually or simultaneously standardized to a maximal cell frequency of 1.
`margin`	a numeric vector with the margins to equate. Must be one of `1`, `2`, or `c(1, 2)` (the default), which corresponds to standardizing the row, column, or both margins in each 2 by 2 table. Only used if `std` equals `"margins"`.
`space`	the amount of space (as a fraction of the maximal radius of the quarter circles) used for the row and column lebals.
`main`	character string for the fourfold title.
`mfrow`	a numeric vector of the form `c(nr, nc)`, indicating that the displays for the 2 by 2 tables should be arranged in an `nr` by `nc` layout, filled by rows.
`mfcol`	a numeric vector of the form `c(nr, nc)`, indicating that the displays for the 2 by 2 tables should be arranged in an `nr` by `nc` layout, filled by columns.

Details

The fourfold display is designed for the display of 2 by 2 by k tables.

Following suitable standardization, the cell frequencies f_{ij} of each 2 by 2 table are shown as a quarter circle whose radius is proportional to $\sqrt{f_{ij}}$ so that its area is proportional to the cell frequency. An association (odds ratio different from 1) between the binary row and column variables is indicated by the tendency of diagonally opposite cells in one direction to differ in size from those

in the other direction; color is used to show this direction. Confidence rings for the odds ratio allow a visual test of the null of no association; the rings for adjacent quadrants overlap if and only if the observed counts are consistent with the null hypothesis.

Typically, the number k corresponds to the number of levels of a stratifying variable, and it is of interest to see whether the association is homogeneous across strata. The fourfold display visualizes the pattern of association. Note that the confidence rings for the individual odds ratios are not adjusted for multiple testing.

References

Friendly, M. (1994). A fourfold display for 2 by 2 by k tables. Technical Report 217, York University, Psychology Department. http://www.math.yorku.ca/SCS/Papers/4fold/4fold.ps.gz

See Also

`mosaicplot`

Examples

```
## Use the Berkeley admission data as in Friendly (1995).
x <- aperm(UCBAdmissions, c(2, 1, 3))
dimnames(x)[[2]] <- c("Yes", "No")
names(dimnames(x)) <- c("Sex", "Admit?", "Department")
stats::ftable(x)

## Fourfold display of data aggregated over departments, with
## frequencies standardized to equate the margins for admission
## and sex.
## Figure 1 in Friendly (1994).
fourfoldplot(margin.table(x, c(1, 2)))

## Fourfold display of x, with frequencies in each table
## standardized to equate the margins for admission and sex.
## Figure 2 in Friendly (1994).
fourfoldplot(x)

## Fourfold display of x, with frequencies in each table
## standardized to equate the margins for admission. but not
## for sex.
## Figure 3 in Friendly (1994).
fourfoldplot(x, margin = 2)
```

`frame` *Create / Start a New Plot Frame*

Description

This function (`frame` is an alias for `plot.new`) causes the completion of plotting in the current plot (if there is one) and an advance to a new graphics frame. This is used in all high-level plotting functions and also useful for skipping plots when a multi-figure region is in use.

Usage

```
plot.new()
frame()
```

Details

The new page is painted with the background colour (`par("bg")`), which is often transparent. For devices with a *canvas* colour (the on-screen devices `X11`, `windows` and `quartz`), the window is first painted with the canvas colour and then the background colour.

There is a hook called `"plot.new"` (see `setHook`) called immediately after advancing the frame, which is used in the testing code to annotate the new page. The hook function(s) are called with no argument. (If the value is a character string, `get` is called on it from within the **graphics** name space.)

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`frame`.)

See Also

`plot.window`, `plot.default`.

`grid` *Add Grid to a Plot*

Description

`grid` adds an `nx` by `ny` rectangular grid to an existing plot.

Usage

```
grid(nx = NULL, ny = nx, col = "lightgray", lty = "dotted",
     lwd = par("lwd"), equilogs = TRUE)
```

Arguments

`nx, ny`	number of cells of the grid in x and y direction. When `NULL`, as per default, the grid aligns with the tick marks on the corresponding *default* axis (i.e., tick-marks as computed by `axTicks`). When `NA`, no grid lines are drawn in the corresponding direction.
`col`	character or (integer) numeric; color of the grid lines.
`lty`	character or (integer) numeric; line type of the grid lines.
`lwd`	non-negative numeric giving line width of the grid lines.
`equilogs`	logical, only used when *log* coordinates and alignment with the axis tick marks are active. Setting `equilogs = FALSE` in that case gives *non equidistant* tick aligned grid lines.

Note

If more fine tuning is required, use `abline(h = ., v = .)` directly.

References

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`plot`, `abline`, `lines`, `points`.

Examples

```
plot(1:3)
grid(NA, 5, lwd = 2) # grid only in y-direction

## maybe change the desired number of tick marks:  par(lab=c(mx,my,7))
op <- par(mfcol = 1:2)
with(iris,
     {
     plot(Sepal.Length, Sepal.Width, col = as.integer(Species),
          xlim = c(4, 8), ylim = c(2, 4.5), panel.first = grid(),
          main = "with(iris,  plot(...., panel.first = grid(), ..) )")
     plot(Sepal.Length, Sepal.Width, col = as.integer(Species),
          panel.first = grid(3, lty=1,lwd=2),
          main = "... panel.first = grid(3, lty=1,lwd=2), ..")
     }
    )
par(op)
```

---

hist *Histograms*

Description

The generic function `hist` computes a histogram of the given data values. If `plot=TRUE`, the resulting object of `class "histogram"` is plotted by `plot.histogram`, before it is returned.

Usage

```
hist(x, ...)

## Default S3 method:
hist(x, breaks = "Sturges",
     freq = NULL, probability = !freq,
     include.lowest = TRUE, right = TRUE,
     density = NULL, angle = 45, col = NULL, border = NULL,
     main = paste("Histogram of" , xname),
     xlim = range(breaks), ylim = NULL,
     xlab = xname, ylab,
     axes = TRUE, plot = TRUE, labels = FALSE,
     nclass = NULL, ...)
```

Arguments

x	a vector of values for which the histogram is desired.
breaks	one of: • a vector giving the breakpoints between histogram cells, • a single number giving the number of cells for the histogram, • a character string naming an algorithm to compute the number of cells (see 'Details'), • a function to compute the number of cells. In the last three cases the number is a suggestion only.
freq	logical; if TRUE, the histogram graphic is a representation of frequencies, the counts component of the result; if FALSE, probability densities, component density, are plotted (so that the histogram has a total area of one). Defaults to TRUE *if and only if* breaks are equidistant (and probability is not specified).
probability	an *alias* for !freq, for S compatibility.
include.lowest	logical; if TRUE, an x[i] equal to the breaks value will be included in the first (or last, for right = FALSE) bar. This will be ignored (with a warning) unless breaks is a vector.
right	logical; if TRUE, the histograms cells are right-closed (left open) intervals.
density	the density of shading lines, in lines per inch. The default value of NULL means that no shading lines are drawn. Non-positive values of density also inhibit the drawing of shading lines.
angle	the slope of shading lines, given as an angle in degrees (counter-clockwise).
col	a colour to be used to fill the bars. The default of NULL yields unfilled bars.
border	the color of the border around the bars. The default is to use the standard foreground color.
main, xlab, ylab	these arguments to title have useful defaults here.
xlim, ylim	the range of x and y values with sensible defaults. Note that xlim is *not* used to define the histogram (breaks), but only for plotting (when plot = TRUE).
axes	logical. If TRUE (default), axes are draw if the plot is drawn.
plot	logical. If TRUE (default), a histogram is plotted, otherwise a list of breaks and counts is returned. In the latter case, a warning is used if (typically graphical) arguments are specified that only apply to the plot = TRUE case.
labels	logical or character. Additionally draw labels on top of bars, if not FALSE; see plot.histogram.
nclass	numeric (integer). For S(-PLUS) compatibility only, nclass is equivalent to breaks for a scalar or character argument.
...	further arguments and graphical parameters passed to plot.histogram and thence to title and axis (if plot=TRUE).

Details

The definition of *histogram* differs by source (with country-specific biases). R's default with equi-spaced breaks (also the default) is to plot the counts in the cells defined by `breaks`. Thus the height of a rectangle is proportional to the number of points falling into the cell, as is the area *provided* the breaks are equally-spaced.

The default with non-equi-spaced breaks is to give a plot of area one, in which the *area* of the rectangles is the fraction of the data points falling in the cells.

If `right = TRUE` (default), the histogram cells are intervals of the form `(a, b]`, i.e., they include their right-hand endpoint, but not their left one, with the exception of the first cell when `include.lowest` is `TRUE`.

For `right = FALSE`, the intervals are of the form `[a, b)`, and `include.lowest` means '*include highest*'.

A numerical tolerance of 10^{-7} times the median bin size is applied when counting entries on the edges of bins.

The default for `breaks` is `"Sturges"`: see `nclass.Sturges`. Other names for which algorithms are supplied are `"Scott"` and `"FD"` / `"Freedman-Diaconis"` (with corresponding functions `nclass.scott` and `nclass.FD`). Case is ignored and partial matching is used. Alternatively, a function can be supplied which will compute the intended number of breaks as a function of `x`.

Value

an object of class `"histogram"` which is a list with components:

`breaks`	the $n + 1$ cell boundaries (= `breaks` if that was a vector).
`counts`	n integers; for each cell, the number of `x[]` inside.
`density`	values $\hat{f}(x_i)$, as estimated density values. If `all(diff(breaks) == 1)`, they are the relative frequencies `counts/n` and in general satisfy $\sum_i \hat{f}(x_i)(b_{i+1} - b_i) = 1$, where b_i = `breaks[i]`.
`intensities`	same as `density`. Deprecated, but retained for compatibility.
`mids`	the n cell midpoints.
`xname`	a character string with the actual `x` argument name.
`equidist`	logical, indicating if the distances between `breaks` are all the same.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Venables, W. N. and Ripley. B. D. (2002) *Modern Applied Statistics with S*. Springer.

See Also

`nclass.Sturges`, `stem`, `density`, `truehist` in package **MASS**.

Typical plots with vertical bars are *not* histograms. Consider `barplot` or `plot(*, type = "h")` for such bar plots.

Examples

```
op <- par(mfrow=c(2, 2))
hist(islands)
utils::str(hist(islands, col="gray", labels = TRUE))

hist(sqrt(islands), breaks = 12, col="lightblue", border="pink")
##-- For non-equidistant breaks, counts should NOT be graphed unscaled:
r <- hist(sqrt(islands), breaks = c(4*0:5, 10*3:5, 70, 100, 140),
          col='blue1')
text(r$mids, r$density, r$counts, adj=c(.5, -.5), col='blue3')
sapply(r[2:3], sum)
sum(r$density * diff(r$breaks)) # == 1
lines(r, lty = 3, border = "purple") # -> lines.histogram(*)
par(op)

require(utils) # for str
str(hist(islands, breaks=12, plot= FALSE)) #-> 10 (~= 12) breaks
str(hist(islands, breaks=c(12,20,36,80,200,1000,17000), plot = FALSE))

hist(islands, breaks=c(12,20,36,80,200,1000,17000), freq = TRUE,
     main = "WRONG histogram") # and warning

require(stats)
set.seed(14)
x <- rchisq(100, df = 4)

## Comparing data with a model distribution should be done with qqplot()!
qqplot(x, qchisq(ppoints(x), df = 4)); abline(0,1, col = 2, lty = 2)

## if you really insist on using hist() ... :
hist(x, freq = FALSE, ylim = c(0, 0.2))
curve(dchisq(x, df = 4), col = 2, lty = 2, lwd = 2, add = TRUE)
```

---

hist.POSIXt *Histogram of a Date or Date-Time Object*

Description

Method for `hist` applied to date or date-time objects.

Usage

```
## S3 method for class 'POSIXt':
hist(x, breaks, ...,
     xlab = deparse(substitute(x)),
     plot = TRUE, freq = FALSE,
     start.on.monday = TRUE, format)

## S3 method for class 'Date':
hist(x, breaks, ...,
     xlab = deparse(substitute(x)),
     plot = TRUE, freq = FALSE,
     start.on.monday = TRUE, format)
```

Arguments

x	an object inheriting from class "POSIXt" or "Date".
breaks	a vector of cut points *or* number giving the number of intervals which x is to be cut into *or* an interval specification, one of "days", "weeks", "months", "quarters" or "years", plus "secs", "mins", "hours" for date-time objects.
...	graphical parameters, or arguments to hist.default such as include.lowest, right and labels.
xlab	a character string giving the label for the x axis, if plotted.
plot	logical. If TRUE (default), a histogram is plotted, otherwise a list of breaks and counts is returned.
freq	logical; if TRUE, the histogram graphic is a representation of frequencies, i.e, the counts component of the result; if FALSE, *relative* frequencies (probabilities) are plotted.
start.on.monday	logical. If breaks = "weeks", should the week start on Mondays or Sundays?
format	for the x-axis labels. See strptime.

Details

Using breaks = "quarters" will create intervals of 3 calendar months, with the intervals beginning on January 1, April 1, July 1 or October 1, based upon min(x) as appropriate.

Value

An object of class "histogram": see hist.

See Also

seq.POSIXt, axis.POSIXct, hist

Examples

```
hist(.leap.seconds, "years", freq = TRUE)
hist(.leap.seconds,
     seq(ISOdate(1970, 1, 1), ISOdate(2010, 1, 1), "5 years"))

## 100 random dates in a 10-week period
random.dates <- as.Date("2001/1/1") + 70*stats::runif(100)
hist(random.dates, "weeks", format = "%d %b")
```

`identify` *Identify Points in a Scatter Plot*

Description

`identify` reads the position of the graphics pointer when the (first) mouse button is pressed. It then searches the coordinates given in `x` and `y` for the point closest to the pointer. If this point is close enough to the pointer, its index will be returned as part of the value of the call.

Usage

```
identify(x, ...)

## Default S3 method:
identify(x, y = NULL, labels = seq_along(x), pos = FALSE,
         n = length(x), plot = TRUE, atpen = FALSE, offset = 0.5,
         tolerance = 0.25, ...)
```

Arguments

`x,y`	coordinates of points in a scatter plot. Alternatively, any object which defines coordinates (a plotting structure, time series etc: see `xy.coords`) can be given as `x`, and `y` left missing.
`labels`	an optional character vector giving labels for the points. Will be coerced using `as.character`, and recycled if necessary to the length of `x`. Excess labels will be discarded, with a warning.
`pos`	if `pos` is `TRUE`, a component is added to the return value which indicates where text was plotted relative to each identified point: see Value.
`n`	the maximum number of points to be identified.
`plot`	logical: if `plot` is `TRUE`, the labels are printed near the points and if `FALSE` they are omitted.
`atpen`	logical: if `TRUE` and `plot = TRUE`, the lower-left corners of the labels are plotted at the points clicked rather than relative to the points.
`offset`	the distance (in character widths) which separates the label from identified points. Negative values are allowed. Not used if `atpen = TRUE`.
`tolerance`	the maximal distance (in inches) for the pointer to be 'close enough' to a point.
`...`	further arguments passed to `par` such as `cex`, `col` and `font`.

Details

`identify` is a generic function, and only the default method is described here.

`identify` is only supported on screen devices such as `X11`, `windows` and `quartz`. On other devices the call will do nothing.

Clicking near (as defined by `tolerance`) a point adds it to the list of identified points. Points can be identified only once, and if the point has already been identified or the click is not near any of the points a message is printed immediately on the R console.

If `plot` is `TRUE`, the point is labelled with the corresponding element of `labels`. If `atpen` is false (the default) the labels are placed below, to the left, above or to the right of the identified point,

depending on where the pointer was relative to the point. If `atpen` is true, the labels are placed with the bottom left of the string's box at the pointer.

For the usual `X11` device the identification process is terminated by pressing any mouse button other than the first. For the `quartz` device the process is terminated by pressing either the pop-up menu equivalent (usually second mouse button or `Ctrl`-click) or the `ESC` key.

On most devices which support `identify`, successful selection of a point is indicated by a bell sound unless `options(locatorBell = FALSE)` has been set.

If the window is resized or hidden and then exposed before the identification process has terminated, any labels drawn by `identify` will disappear. These will reappear once the identification process has terminated and the window is resized or hidden and exposed again. This is because the labels drawn by `identify` are not recorded in the device's display list until the identification process has terminated.

If you interrupt the `identify` call this leaves the graphics device in an undefined state, with points labelled but labels not recorded in the display list. Copying a device in that state will give unpredictable results.

Value

If `pos` is `FALSE`, an integer vector containing the indices of the identified points, in the order they were identified.

If `pos` is `TRUE`, a list containing a component `ind`, indicating which points were identified and a component `pos`, indicating where the labels were placed relative to the identified points (1=below, 2=left, 3=above, 4=right and 0=no offset, used if `atpen = TRUE`).

Technicalities

The algorithm used for placing labels is the same as used by `text` if `pos` is specified there, the difference being that the position of the pointer relative the identified point determines `pos` in `identify`.

For labels placed to the left of a point, the right-hand edge of the string's box is placed `offset` units to the left of the point, and analogously for points to the right. The baseline of the text is placed below the point so as to approximately centre string vertically. For labels placed above or below a point, the string is centered horizontally on the point. For labels placed above, the baseline of the text is placed `offset` units above the point, and for those placed below, the baseline is placed so that the top of the string's box is approximately `offset` units below the point. If you want more precise placement (e.g. centering) use `plot = FALSE` and plot via `text` or `points`: see the examples.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`locator`, `text`.

Examples

```
## A function to use identify to select points, and overplot the
## points with another symbol as they are selected
identifyPch <- function(x, y=NULL, n=length(x), pch=19, ...)
```

```
{
    xy <- xy.coords(x, y); x <- xy$x; y <- xy$y
    sel <- rep(FALSE, length(x)); res <- integer(0)
    while(sum(sel) < n) {
        ans <- identify(x[!sel], y[!sel], n=1, plot=FALSE, ...)
        if(!length(ans)) break
        ans <- which(!sel)[ans]
        points(x[ans], y[ans], pch = pch)
        sel[ans] <- TRUE
        res <- c(res, ans)
    }
    res
}
```

`image` *Display a Color Image*

Description

Creates a grid of colored or gray-scale rectangles with colors corresponding to the values in `z`. This can be used to display three-dimensional or spatial data aka *images*. This is a generic function.

The functions `heat.colors`, `terrain.colors` and `topo.colors` create heat-spectrum (red to white) and topographical color schemes suitable for displaying ordered data, with `n` giving the number of colors desired.

Usage

```
image(x, ...)

## Default S3 method:
image(x, y, z, zlim, xlim, ylim, col = heat.colors(12),
      add = FALSE, xaxs = "i", yaxs = "i", xlab, ylab,
      breaks, oldstyle = FALSE, ...)
```

Arguments

`x,y`	locations of grid lines at which the values in `z` are measured. These must be finite, non-missing and in (strictly) ascending order. By default, equally spaced values from 0 to 1 are used. If `x` is a `list`, its components `x$x` and `x$y` are used for `x` and `y`, respectively. If the list has component `z` this is used for `z`.
`z`	a matrix containing the values to be plotted (`NA`s are allowed). Note that `x` can be used instead of `z` for convenience.
`zlim`	the minimum and maximum `z` values for which colors should be plotted, defaulting to the range of the finite values of `z`. Each of the given colors will be used to color an equispaced interval of this range. The *midpoints* of the intervals cover the range, so that values just outside the range will be plotted.
`xlim, ylim`	ranges for the plotted `x` and `y` values, defaulting to the ranges of `x` and `y`.
`col`	a list of colors such as that generated by `rainbow`, `heat.colors`, `topo.colors`, `terrain.colors` or similar functions.
`add`	logical; if `TRUE`, add to current plot (and disregard the following arguments). This is rarely useful because `image` 'paints' over existing graphics.

`xaxs, yaxs`	style of x and y axis. The default `"i"` is appropriate for images. See `par`.
`xlab, ylab`	each a character string giving the labels for the x and y axis. Default to the 'call names' of `x` or `y`, or to `""` if these were unspecified.
`breaks`	a set of breakpoints for the colours: must give one more breakpoint than colour.
`oldstyle`	logical. If true the midpoints of the colour intervals are equally spaced, and `zlim[1]` and `zlim[2]` were taken to be midpoints. The default is to have colour intervals of equal lengths between the limits.
`...`	graphical parameters for `plot` may also be passed as arguments to this function, as can the plot aspect ratio `asp` and `axes` (see `plot.window`).

Details

The length of `x` should be equal to the `nrow(z)+1` or `nrow(z)`. In the first case `x` specifies the boundaries between the cells: in the second case `x` specifies the midpoints of the cells. Similar reasoning applies to `y`. It probably only makes sense to specify the midpoints of an equally-spaced grid. If you specify just one row or column and a length-one `x` or `y`, the whole user area in the corresponding direction is filled.

Rectangles corresponding to missing values are not plotted (and so are transparent and (unless `add=TRUE`) the default background painted in `par("bg")` will show though and if that is transparent, the canvas colour will be seen).

If `breaks` is specified then `zlim` is unused and the algorithm used follows `cut`, so intervals are closed on the right and open on the left except for the lowest interval.

Notice that `image` interprets the `z` matrix as a table of `f(x[i], y[j])` values, so that the x axis corresponds to row number and the y axis to column number, with column 1 at the bottom, i.e. a 90 degree counter-clockwise rotation of the conventional printed layout of a matrix.

Note

Based on a function by Thomas Lumley ⟨tlumley@u.washington.edu⟩.

See Also

`filled.contour` or `heatmap` which can look nicer (but are less modular), `contour`; The **lattice** equivalent of `image` is `levelplot`.

`heat.colors`, `topo.colors`, `terrain.colors`, `rainbow`, `hsv`, `par`.

Examples

```
require(grDevices) # for colours
x <- y <- seq(-4*pi, 4*pi, len=27)
r <- sqrt(outer(x^2, y^2, "+"))
image(z = z <- cos(r^2)*exp(-r/6), col=gray((0:32)/32))
image(z, axes = FALSE, main = "Math can be beautiful ...",
      xlab = expression(cos(r^2) * e^{-r/6}))
contour(z, add = TRUE, drawlabels = FALSE)

# Volcano data visualized as matrix. Need to transpose and flip
# matrix horizontally.
image(t(volcano)[ncol(volcano):1,])

# A prettier display of the volcano
x <- 10*(1:nrow(volcano))
```

```
y <- 10*(1:ncol(volcano))
image(x, y, volcano, col = terrain.colors(100), axes = FALSE)
contour(x, y, volcano, levels = seq(90, 200, by = 5),
        add = TRUE, col = "peru")
axis(1, at = seq(100, 800, by = 100))
axis(2, at = seq(100, 600, by = 100))
box()
title(main = "Maunga Whau Volcano", font.main = 4)
```

---

layout *Specifying Complex Plot Arrangements*

Description

`layout` divides the device up into as many rows and columns as there are in matrix `mat`, with the column-widths and the row-heights specified in the respective arguments.

Usage

```
layout(mat, widths = rep(1, ncol(mat)),
       heights = rep(1, nrow(mat)), respect = FALSE)

layout.show(n = 1)
lcm(x)
```

Arguments

`mat`	a matrix object specifying the location of the next N figures on the output device. Each value in the matrix must be `0` or a positive integer. If N is the largest positive integer in the matrix, then the integers $\{1, \ldots, N-1\}$ must also appear at least once in the matrix.
`widths`	a vector of values for the widths of columns on the device. Relative widths are specified with numeric values. Absolute widths (in centimetres) are specified with the `lcm()` function (see examples).
`heights`	a vector of values for the heights of rows on the device. Relative and absolute heights can be specified, see `widths` above.
`respect`	either a logical value or a matrix object. If the latter, then it must have the same dimensions as `mat` and each value in the matrix must be either `0` or `1`.
`n`	number of figures to plot.
`x`	a dimension to be interpreted as a number of centimetres.

Details

Figure i is allocated a region composed from a subset of these rows and columns, based on the rows and columns in which i occurs in `mat`.

The `respect` argument controls whether a unit column-width is the same physical measurement on the device as a unit row-height.

There is a limit (currently 50) for the numbers of rows and columns in the layout, and also for the total number of cells (500).

`layout.show(n)` plots (part of) the current layout, namely the outlines of the next `n` figures.

`lcm` is a trivial function, to be used as *the* interface for specifying absolute dimensions for the `widths` and `heights` arguments of `layout()`.

Value

`layout` returns the number of figures, N, see above.

Warnings

These functions are totally incompatible with the other mechanisms for arranging plots on a device: `par(mfrow)`, `par(mfcol)` and `split.screen`.

Author(s)

Paul R. Murrell

References

Murrell, P. R. (1999) Layouts: A mechanism for arranging plots on a page. *Journal of Computational and Graphical Statistics*, **8**, 121-134.

Chapter 5 of Paul Murrell's Ph.D. thesis.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`par` with arguments `mfrow`, `mfcol`, or `mfg`.

Examples

```
def.par <- par(no.readonly = TRUE) # save default, for resetting...

## divide the device into two rows and two columns
## allocate figure 1 all of row 1
## allocate figure 2 the intersection of column 2 and row 2
layout(matrix(c(1,1,0,2), 2, 2, byrow = TRUE))
## show the regions that have been allocated to each plot
layout.show(2)

## divide device into two rows and two columns
## allocate figure 1 and figure 2 as above
## respect relations between widths and heights
nf <- layout(matrix(c(1,1,0,2), 2, 2, byrow=TRUE), respect=TRUE)
layout.show(nf)

## create single figure which is 5cm square
nf <- layout(matrix(1), widths=lcm(5), heights=lcm(5))
layout.show(nf)

##-- Create a scatterplot with marginal histograms -----

x <- pmin(3, pmax(-3, stats::rnorm(50)))
y <- pmin(3, pmax(-3, stats::rnorm(50)))
xhist <- hist(x, breaks=seq(-3,3,0.5), plot=FALSE)
yhist <- hist(y, breaks=seq(-3,3,0.5), plot=FALSE)
```

```
top <- max(c(xhist$counts, yhist$counts))
xrange <- c(-3,3)
yrange <- c(-3,3)
nf <- layout(matrix(c(2,0,1,3),2,2,byrow=TRUE), c(3,1), c(1,3), TRUE)
layout.show(nf)

par(mar=c(3,3,1,1))
plot(x, y, xlim=xrange, ylim=yrange, xlab="", ylab="")
par(mar=c(0,3,1,1))
barplot(xhist$counts, axes=FALSE, ylim=c(0, top), space=0)
par(mar=c(3,0,1,1))
barplot(yhist$counts, axes=FALSE, xlim=c(0, top), space=0, horiz=TRUE)

par(def.par)#- reset to default
```

`legend` *Add Legends to Plots*

Description

This function can be used to add legends to plots. Note that a call to the function `locator(1)` can be used in place of the `x` and `y` arguments.

Usage

```
legend(x, y = NULL, legend, fill = NULL, col = par("col"),
       lty, lwd, pch,
       angle = 45, density = NULL, bty = "o", bg = par("bg"),
       box.lwd = par("lwd"), box.lty = par("lty"), box.col = par("fg"),
       pt.bg = NA, cex = 1, pt.cex = cex, pt.lwd = lwd,
       xjust = 0, yjust = 1, x.intersp = 1, y.intersp = 1,
       adj = c(0, 0.5), text.width = NULL, text.col = par("col"),
       merge = do.lines && has.pch, trace = FALSE,
       plot = TRUE, ncol = 1, horiz = FALSE, title = NULL,
       inset = 0, xpd, title.col = text.col)
```

Arguments

`x, y`	the x and y co-ordinates to be used to position the legend. They can be specified by keyword or in any way which is accepted by `xy.coords`: See 'Details'.
`legend`	a character or expression vector. of length ≥ 1 to appear in the legend. Other objects will be coerced by `as.graphicsAnnot`.
`fill`	if specified, this argument will cause boxes filled with the specified colors (or shaded in the specified colors) to appear beside the legend text.
`col`	the color of points or lines appearing in the legend.
`lty, lwd`	the line types and widths for lines appearing in the legend. One of these two *must* be specified for line drawing.
`pch`	the plotting symbols appearing in the legend, either as vector of 1-character strings, or one (multi character) string. *Must* be specified for symbol drawing.
`angle`	angle of shading lines.

`density` the density of shading lines, if numeric and positive. If `NULL` or negative or `NA` color filling is assumed.

`bty` the type of box to be drawn around the legend. The allowed values are `"o"` (the default) and `"n"`.

`bg` the background color for the legend box. (Note that this is only used if `bty != "n"`.)

`box.lty, box.lwd, box.col`
the line type, width and color for the legend box (if `bty = "o"`).

`pt.bg` the background color for the `points`, corresponding to its argument `bg`.

`cex` character expansion factor **relative** to current `par("cex")`. Used for text, and provides the default for `pt.cex` and `title.cex`.

`pt.cex` expansion factor(s) for the points.

`pt.lwd` line width for the points, defaults to the one for lines, or if that is not set, to `par("lwd")`.

`xjust` how the legend is to be justified relative to the legend x location. A value of 0 means left justified, 0.5 means centered and 1 means right justified.

`yjust` the same as `xjust` for the legend y location.

`x.intersp` character interspacing factor for horizontal (x) spacing.

`y.intersp` the same for vertical (y) line distances.

`adj` numeric of length 1 or 2; the string adjustment for legend text. Useful for y-adjustment when `labels` are plotmath expressions.

`text.width` the width of the legend text in x (`"user"`) coordinates. (Should be positive even for a reversed x axis.) Defaults to the proper value computed by `strwidth(legend)`.

`text.col` the color used for the legend text.

`merge` logical; if `TRUE`, merge points and lines but not filled boxes. Defaults to `TRUE` if there are points and lines.

`trace` logical; if `TRUE`, shows how `legend` does all its magical computations.

`plot` logical. If `FALSE`, nothing is plotted but the sizes are returned.

`ncol` the number of columns in which to set the legend items (default is 1, a vertical legend).

`horiz` logical; if `TRUE`, set the legend horizontally rather than vertically (specifying `horiz` overrides the `ncol` specification).

`title` a character string or length-one expression giving a title to be placed at the top of the legend. Other objects will be coerced by `as.graphicsAnnot`.

`inset` inset distance(s) from the margins as a fraction of the plot region when legend is placed by keyword.

`xpd` if supplied, a value of the graphical parameter `xpd` to be used while the legend is being drawn.

`title.col` color for `title`.

Details

Arguments `x, y, legend` are interpreted in a non-standard way to allow the coordinates to be specified *via* one or two arguments. If `legend` is missing and `y` is not numeric, it is assumed that the second argument is intended to be `legend` and that the first argument specifies the coordinates.

The coordinates can be specified in any way which is accepted by `xy.coords`. If this gives the coordinates of one point, it is used as the top-left coordinate of the rectangle containing the legend. If it gives the coordinates of two points, these specify opposite corners of the rectangle (either pair of corners, in any order).

The location may also be specified by setting `x` to a single keyword from the list `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"` and `"center"`. This places the legend on the inside of the plot frame at the given location. Partial argument matching is used. The optional `inset` argument specifies how far the legend is inset from the plot margins. If a single value is given, it is used for both margins; if two values are given, the first is used for `x`- distance, the second for `y`-distance.

Attribute arguments such as `col`, `pch`, `lty`, etc, are recycled if necessary: `merge` is not.

Points are drawn *after* lines in order that they can cover the line with their background color `pt.bg`, if applicable.

See the examples for how to right-justify labels.

Value

A list with list components

`rect` a list with components

- **`w, h`** positive numbers giving **w**idth and **h**eight of the legend's box.
- **`left, top`** x and y coordinates of upper left corner of the box.

`text` a list with components

- **`x, y`** numeric vectors of length `length(legend)`, giving the x and y coordinates of the legend's text(s).

returned invisibly.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`plot`, `barplot` which uses `legend()`, and `text` for more examples of math expressions.

Examples

```
## Run the example in '?matplot' or the following:
leg.txt <- c("Setosa     Petals", "Setosa     Sepals",
             "Versicolor Petals", "Versicolor Sepals")
y.leg <- c(4.5, 3, 2.1, 1.4, .7)
cexv  <- c(1.2, 1, 4/5, 2/3, 1/2)
matplot(c(1,8), c(0,4.5), type = "n", xlab = "Length", ylab = "Width",
        main = "Petal and Sepal Dimensions in Iris Blossoms")
```

```
for (i in seq(cexv)) {
  text  (1, y.leg[i]-.1, paste("cex=",formatC(cexv[i])), cex=.8, adj = 0)
  legend(3, y.leg[i], leg.txt, pch = "sSvV", col = c(1, 3), cex = cexv[i])
}

## 'merge = TRUE' for merging lines & points:
x <- seq(-pi, pi, len = 65)
plot(x, sin(x), type = "l", ylim = c(-1.2, 1.8), col = 3, lty = 2)
points(x, cos(x), pch = 3, col = 4)
lines(x, tan(x), type = "b", lty = 1, pch = 4, col = 6)
title("legend(..., lty = c(2, -1, 1), pch = c(-1,3,4), merge = TRUE)",
      cex.main = 1.1)
legend(-1, 1.9, c("sin", "cos", "tan"), col = c(3,4,6),
       text.col = "green4", lty = c(2, -1, 1), pch = c(-1, 3, 4),
       merge = TRUE, bg = 'gray90')

## right-justifying a set of labels: thanks to Uwe Ligges
x <- 1:5; y1 <- 1/x; y2 <- 2/x
plot(rep(x, 2), c(y1, y2), type="n", xlab="x", ylab="y")
lines(x, y1); lines(x, y2, lty=2)
temp <- legend("topright", legend = c(" ", " "),
               text.width = strwidth("1,000,000"),
               lty = 1:2, xjust = 1, yjust = 1,
               title = "Line Types")
text(temp$rect$left + temp$rect$w, temp$text$y,
     c("1,000", "1,000,000"), pos=2)

##--- log scaled Examples ------------------------------
leg.txt <- c("a one", "a two")

par(mfrow = c(2,2))
for(ll in c("","x","y","xy")) {
  plot(2:10, log=ll, main=paste("log = '",ll,"'", sep=""))
  abline(1,1)
  lines(2:3,3:4, col=2) #
  points(2,2, col=3)    #
  rect(2,3,3,2, col=4)
  text(c(3,3),2:3, c("rect(2,3,3,2, col=4)",
                     "text(c(3,3),2:3,\"c(rect(...)\")"), adj = c(0,.3))
  legend(list(x=2,y=8), legend = leg.txt, col=2:3, pch=1:2,
         lty=1, merge=TRUE)#, trace=TRUE)
}
par(mfrow=c(1,1))

##-- Math expressions:  ------------------------------
x <- seq(-pi, pi, len = 65)
plot(x, sin(x), type="l", col = 2, xlab = expression(phi),
     ylab = expression(f(phi)))
abline(h=-1:1, v=pi/2*(-6:6), col="gray90")
lines(x, cos(x), col = 3, lty = 2)
ex.cs1 <- expression(plain(sin) * phi,  paste("cos", phi))# 2 ways
utils::str(legend(-3, .9, ex.cs1, lty=1:2, plot=FALSE,
           adj = c(0, .6)))# adj y !
legend(-3, .9, ex.cs1, lty=1:2, col=2:3,    adj = c(0, .6))

require(stats)
x <- rexp(100, rate = .5)
```

```
hist(x, main = "Mean and Median of a Skewed Distribution")
abline(v = mean(x),   col=2, lty=2, lwd=2)
abline(v = median(x), col=3, lty=3, lwd=2)
ex12 <- expression(bar(x) == sum(over(x[i], n), i==1, n),
                   hat(x) == median(x[i], i==1,n))
utils::str(legend(4.1, 30, ex12, col = 2:3, lty=2:3, lwd=2))

## 'Filled' boxes -- for more, see example(plot.factor)
op <- par(bg="white") # to get an opaque box for the legend
plot(cut(weight, 3) ~ group, data = PlantGrowth, col = NULL,
     density = 16*(1:3))
par(op)

## Using 'ncol' :
x <- 0:64/64
matplot(x, outer(x, 1:7, function(x, k) sin(k * pi * x)),
        type = "o", col = 1:7, ylim = c(-1, 1.5), pch = "*")
op <- par(bg="antiquewhite1")
legend(0, 1.5, paste("sin(", 1:7, "pi * x)"), col=1:7, lty=1:7,
       pch = "*", ncol = 4, cex = 0.8)
legend(.8,1.2, paste("sin(", 1:7, "pi * x)"), col=1:7, lty=1:7,
       pch = "*", cex = 0.8)
legend(0, -.1, paste("sin(", 1:4, "pi * x)"), col=1:4, lty=1:4,
       ncol = 2, cex = 0.8)
legend(0, -.4, paste("sin(", 5:7, "pi * x)"), col=4:6,  pch=24,
       ncol = 2, cex = 1.5, lwd = 2, pt.bg = "pink", pt.cex = 1:3)
par(op)

## point covering line :
y <- sin(3*pi*x)
plot(x, y, type="l", col="blue",
    main = "points with bg & legend(*, pt.bg)")
points(x, y, pch=21, bg="white")
legend(.4,1, "sin(c x)", pch=21, pt.bg="white", lty=1, col = "blue")

## legends with titles at different locations
plot(x, y, type='n')
legend("bottomright", "(x,y)", pch=1, title="bottomright")
legend("bottom", "(x,y)", pch=1, title="bottom")
legend("bottomleft", "(x,y)", pch=1, title="bottomleft")
legend("left", "(x,y)", pch=1, title="left")
legend("topleft", "(x,y)", pch=1, title="topleft, inset = .05",
       inset = .05)
legend("top", "(x,y)", pch=1, title="top")
legend("topright", "(x,y)", pch=1, title="topright, inset = .02",
       inset = .02)
legend("right", "(x,y)", pch=1, title="right")
legend("center", "(x,y)", pch=1, title="center")
```

`lines` *Add Connected Line Segments to a Plot*

Description

A generic function taking coordinates given in various ways and joining the corresponding points with line segments.

Usage

```
lines(x, ...)

## Default S3 method:
lines(x, y = NULL, type = "l", ...)
```

Arguments

`x, y`	coordinate vectors of points to join.
`type`	character indicating the type of plotting; actually any of the `types` as in `plot.default`.
`...`	Further graphical parameters (see `par`) may also be supplied as arguments, particularly, line type, `lty`, line width, `lwd`, color, `col` and for `type = "b"`, `pch`. Also the line characteristics `lend`, `ljoin` and `lmitre`.

Details

The coordinates can be passed in a plotting structure (a list with `x` and `y` components), a two-column matrix, a time series, ... See `xy.coords`. If supplied separately, they must be of the same length.

The coordinates can contain `NA` values. If a point contains `NA` in either its `x` or `y` value, it is omitted from the plot, and lines are not drawn to or from such points. Thus missing values can be used to achieve breaks in lines.

For `type = "h"`, `col` can be a vector and will be recycled as needed.

`lwd` can be a vector: its first element will apply to lines but the whole vector to symbols (recycled as necessary).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`points`, particularly for `type %in% c("p","b","o")`, `plot`, and the workhorse function `plot.xy`.

`abline` for drawing (single) straight lines.

`par` for how to specify colors.

Examples

```
# draw a smooth line through a scatter plot
plot(cars, main="Stopping Distance versus Speed")
lines(stats::lowess(cars))
```

`locator` *Graphical Input*

Description

Reads the position of the graphics cursor when the (first) mouse button is pressed.

Usage

```
locator(n = 512, type = "n", ...)
```

Arguments

`n`	the maximum number of points to locate. Valid values start at 1.
`type`	One of `"n"`, `"p"`, `"l"` or `"o"`. If `"p"` or `"o"` the points are plotted; if `"l"` or `"o"` they are joined by lines.
`...`	additional graphics parameters used if `type != "n"` for plotting the locations.

Details

`locator` is only supported on screen devices such as `X11`, `windows` and `quartz`. On other devices the call will do nothing.

Unless the process is terminated prematurely by the user (see below) at most `n` positions are determined.

For the usual `X11` device the identification process is terminated by pressing any mouse button other than the first. For the `quartz` device the process is terminated by pressing the `ESC` key.

The current graphics parameters apply just as if `plot.default` has been called with the same value of `type`. The plotting of the points and lines is subject to clipping, but locations outside the current clipping rectangle will be returned.

On most devices which support `locator`, successful selection of a point is indicated by a bell sound unless `options(locatorBell=FALSE)` has been set.

If the window is resized or hidden and then exposed before the input process has terminated, any lines or points drawn by `locator` will disappear. These will reappear once the input process has terminated and the window is resized or hidden and exposed again. This is because the points and lines drawn by `locator` are not recorded in the device's display list until the input process has terminated.

Value

A list containing `x` and `y` components which are the coordinates of the identified points in the user coordinate system, i.e., the one specified by `par("usr")`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`identify`

matplot *Plot Columns of Matrices*

Description

Plot the columns of one matrix against the columns of another.

Usage

```
matplot(x, y, type = "p", lty = 1:5, lwd = 1, pch = NULL,
        col = 1:6, cex = NULL, bg = NA,
        xlab = NULL, ylab = NULL, xlim = NULL, ylim = NULL,
        ..., add = FALSE, verbose = getOption("verbose"))

matpoints(x, y, type = "p", lty = 1:5, lwd = 1, pch = NULL,
          col = 1:6, ...)

matlines (x, y, type = "l", lty = 1:5, lwd = 1, pch = NULL,
          col = 1:6, ...)
```

Arguments

x,y	vectors or matrices of data for plotting. The number of rows should match. If one of them are missing, the other is taken as y and an x vector of 1:n is used. Missing values (NAs) are allowed.
type	character string (length 1 vector) or vector of 1-character strings indicating the type of plot for each column of y, see plot for all possible types. The first character of type defines the first plot, the second character the second, etc. Characters in type are cycled through; e.g., "pl" alternately plots points and lines.
lty,lwd	vector of line types and widths. The first element is for the first column, the second element for the second column, etc., even if lines are not plotted for all columns. Line types will be used cyclically until all plots are drawn.
pch	character string or vector of 1-characters or integers for plotting characters, see points. The first character is the plotting-character for the first plot, the second for the second, etc. The default is the digits (1 through 9, 0) then the lowercase and uppercase letters.
col	vector of colors. Colors are used cyclically.
cex	vector of character expansion sizes, used cyclically. This works as a multiple of par("cex"). NULL is equivalent to 1.0.
bg	vector of background (fill) colors for the open plot symbols given by pch=21:25 as in points. The default NA corresponds to the one of the underlying function plot.xy.
xlab, ylab	titles for x and y axes, as in plot.
xlim, ylim	ranges of x and y axes, as in plot.
...	Graphical parameters (see par) and any further arguments of plot, typically plot.default, may also be supplied as arguments to this function. Hence, the high-level graphics control arguments described under par and the arguments to title may be supplied to this function.

add	logical. If TRUE, plots are added to current one, using points and lines.
verbose	logical. If TRUE, write one line of what is done.

Details

Points involving missing values are not plotted.

The first column of x is plotted against the first column of y, the second column of x against the second column of y, etc. If one matrix has fewer columns, plotting will cycle back through the columns again. (In particular, either x or y may be a vector, against which all columns of the other argument will be plotted.)

The first element of col, cex, lty, lwd is used to plot the axes as well as the first line.

Because plotting symbols are drawn with lines and because these functions may be changing the line style, you should probably specify lty=1 when using plotting symbols.

Side Effects

Function matplot generates a new plot; matpoints and matlines add to the current one.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

plot, points, lines, matrix, par.

Examples

```
require(grDevices)
matplot((-4:5)^2, main = "Quadratic") # almost identical to plot(*)
sines <- outer(1:20, 1:4, function(x, y) sin(x / 20 * pi * y))
matplot(sines, pch = 1:4, type = "o", col = rainbow(ncol(sines)))
matplot(sines, type = "b", pch = 21:23, col = 2:5, bg = 2:5,
        main = "matplot(...., pch = 21:23, bg = 2:5)")

x <- 0:50/50
matplot(x, outer(x, 1:8, function(x, k) sin(k*pi * x)),
        ylim = c(-2,2), type = "plobcsSh",
        main= "matplot(,type = \"plobcsSh\" )")
## pch & type =  vector of 1-chars :
matplot(x, outer(x, 1:4, function(x, k) sin(k*pi * x)),
        pch = letters[1:4], type = c("b","p","o"))

table(iris$Species) # is data.frame with 'Species' factor
iS <- iris$Species == "setosa"
iV <- iris$Species == "versicolor"
op <- par(bg = "bisque")
matplot(c(1, 8), c(0, 4.5), type= "n", xlab = "Length", ylab = "Width",
        main = "Petal and Sepal Dimensions in Iris Blossoms")
matpoints(iris[iS,c(1,3)], iris[iS,c(2,4)], pch = "sS", col = c(2,4))
matpoints(iris[iV,c(1,3)], iris[iV,c(2,4)], pch = "vV", col = c(2,4))
legend(1, 4, c("    Setosa Petals", "    Setosa Sepals",
               "Versicolor Petals", "Versicolor Sepals"),
```

```
       pch = "sSvV", col = rep(c(2,4), 2))

nam.var <- colnames(iris)[-5]
nam.spec <- as.character(iris[1+50*0:2, "Species"])
iris.S <- array(NA, dim = c(50,4,3),
                dimnames = list(NULL, nam.var, nam.spec))
for(i in 1:3) iris.S[,,i] <- data.matrix(iris[1:50+50*(i-1), -5])

matplot(iris.S[,"Petal.Length",], iris.S[,"Petal.Width",], pch="SCV",
        col = rainbow(3, start = .8, end = .1),
        sub = paste(c("S", "C", "V"), dimnames(iris.S)[[3]],
                    sep = "=", collapse= ",  "),
        main = "Fisher's Iris Data")
par(op)
```

`mosaicplot` *Mosaic Plots*

Description

Plots a mosaic on the current graphics device.

Usage

```
mosaicplot(x, ...)

## Default S3 method:
mosaicplot(x, main = deparse(substitute(x)),
           sub = NULL, xlab = NULL, ylab = NULL,
           sort = NULL, off = NULL, dir = NULL,
           color = NULL, shade = FALSE, margin = NULL,
           cex.axis = 0.66, las = par("las"),
           type = c("pearson", "deviance", "FT"), ...)

## S3 method for class 'formula':
mosaicplot(formula, data = NULL, ...,
           main = deparse(substitute(data)), subset,
           na.action = stats::na.omit)
```

Arguments

`x`	a contingency table in array form, with optional category labels specified in the `dimnames(x)` attribute. The table is best created by the `table()` command.
`main`	character string for the mosaic title.
`sub`	character string for the mosaic sub-title (at bottom).
`xlab,ylab`	x- and y-axis labels used for the plot; by default, the first and second element of `names(dimnames(X))` (i.e., the name of the first and second variable in `X`).
`sort`	vector ordering of the variables, containing a permutation of the integers `1:length(dim(x))` (the default).

off	vector of offsets to determine percentage spacing at each level of the mosaic (appropriate values are between 0 and 20, and the default is 20 times the number of splits for 2-dimensional tables, and 10 otherwise. Rescaled to maximally 50, and recycled if necessary.
dir	vector of split directions ("v" for vertical and "h" for horizontal) for each level of the mosaic, one direction for each dimension of the contingency table. The default consists of alternating directions, beginning with a vertical split.
color	logical or (recycling) vector of colors for color shading, used only when shade is FALSE, or NULL (default). By default, grey boxes are drawn. color=TRUE uses a gamma-corrected grey palette. color=FALSE gives empty boxes with no shading.
shade	a logical indicating whether to produce extended mosaic plots, or a numeric vector of at most 5 distinct positive numbers giving the absolute values of the cut points for the residuals. By default, shade is FALSE, and simple mosaics are created. Using shade = TRUE cuts absolute values at 2 and 4.
margin	a list of vectors with the marginal totals to be fit in the log-linear model. By default, an independence model is fitted. See loglin for further information.
cex.axis	The magnification to be used for axis annotation, as a multiple of par("cex").
las	numeric; the style of axis labels, see par.
type	a character string indicating the type of residual to be represented. Must be one of "pearson" (giving components of Pearson's χ^2), "deviance" (giving components of the likelihood ratio χ^2), or "FT" for the Freeman-Tukey residuals. The value of this argument can be abbreviated.
formula	a formula, such as y ~ x.
data	a data frame (or list), or a contingency table from which the variables in formula should be taken.
...	further arguments to be passed to or from methods.
subset	an optional vector specifying a subset of observations in the data frame to be used for plotting.
na.action	a function which indicates what should happen when the data contains variables to be cross-tabulated, and these variables contain NAs. The default is to omit cases which have an NA in any variable. Since the tabulation will omit all cases containing missing values, this will only be useful if the na.action function replaces missing values.

Details

This is a generic function. It currently has a default method (mosaicplot.default) and a formula interface (mosaicplot.formula).

Extended mosaic displays visualize standardized residuals of a loglinear model for the table by color and outline of the mosaic's tiles. (Standardized residuals are often referred to a standard normal distribution.) Negative residuals are drawn in shaded of red and with broken outlines; positive ones are drawn in blue with solid outlines.

For the formula method, if data is an object inheriting from classes "table" or "ftable", or an array with more than 2 dimensions, it is taken as a contingency table, and hence all entries should be nonnegative. In this case, the left-hand side of formula should be empty, and the variables on the right-hand side should be taken from the names of the dimnames attribute of the contingency table. A marginal table of these variables is computed, and a mosaic of this table is produced.

Otherwise, `data` should be a data frame or matrix, list or environment containing the variables to be cross-tabulated. In this case, after possibly selecting a subset of the data as specified by the `subset` argument, a contingency table is computed from the variables given in `formula`, and a mosaic is produced from this.

See Emerson (1998) for more information and a case study with television viewer data from Nielsen Media Research.

Missing values are not supported except via an `na.action` function when `data` contains variables to be cross-tabulated.

A more flexible and extensible implementation of mosaic plots written in the grid graphics system is provided in the function `mosaic` in the contributed package **vcd** (Meyer, Zeileis and Hornik, 2005).

Author(s)

S-PLUS original by John Emerson ⟨john.emerson@yale.edu⟩. Originally modified and enhanced for R by Kurt Hornik.

References

Hartigan, J.A., and Kleiner, B. (1984) A mosaic of television ratings. *The American Statistician*, **38**, 32–35.

Emerson, J. W. (1998) Mosaic displays in S-PLUS: A general implementation and a case study. *Statistical Computing and Graphics Newsletter (ASA)*, **9**, 1, 17–23.

Friendly, M. (1994) Mosaic displays for multi-way contingency tables. *Journal of the American Statistical Association*, **89**, 190–200.

Meyer, D., Zeileis, A., and Hornik, K. (2005) The strucplot framework: Visualizing multi-way contingency tables with vcd. *Report 22*, Department of Statistics and Mathematics, Wirtschaftsuniversität Wien, Research Report Series. http://epub.wu-wien.ac.at/dyn/openURL?id=oai:epub.wu-wien.ac.at:epub-wu-01_8a1

The home page of Michael Friendly (http://www.math.yorku.ca/SCS/friendly.html) provides information on various aspects of graphical methods for analyzing categorical data, including mosaic plots.

See Also

`assocplot`, `loglin`.

Examples

```
require(stats)
mosaicplot(Titanic, main = "Survival on the Titanic", color = TRUE)
## Formula interface for tabulated data:
mosaicplot(~ Sex + Age + Survived, data = Titanic, color = TRUE)

mosaicplot(HairEyeColor, shade = TRUE)
## Independence model of hair and eye color and sex.  Indicates that
## there are more blue eyed blonde females than expected in the case
## of independence and too few brown eyed blonde females.
## The corresponding model is:
fm <- loglin(HairEyeColor, list(1, 2, 3))
pchisq(fm$pearson, fm$df, lower.tail = FALSE)
```

```
mosaicplot(HairEyeColor, shade = TRUE, margin = list(1:2, 3))
## Model of joint independence of sex from hair and eye color.  Males
## are underrepresented among people with brown hair and eyes, and are
## overrepresented among people with brown hair and blue eyes.
## The corresponding model is:
fm <- loglin(HairEyeColor, list(1:2, 3))
pchisq(fm$pearson, fm$df, lower.tail = FALSE)

## Formula interface for raw data: visualize cross-tabulation of numbers
## of gears and carburettors in Motor Trend car data.
mosaicplot(~ gear + carb, data = mtcars, color = TRUE, las = 1)
# color recycling
mosaicplot(~ gear + carb, data = mtcars, color = 2:3, las = 1)
```

`mtext` *Write Text into the Margins of a Plot*

Description

Text is written in one of the four margins of the current figure region or one of the outer margins of the device region.

Usage

```
mtext(text, side = 3, line = 0, outer = FALSE, at = NA,
      adj = NA, padj = NA, cex = NA, col = NA, font = NA, ...)
```

Arguments

`text`	a character or expression vector specifying the *text* to be written. Other objects are coerced by `as.graphicsAnnot`.
`side`	on which side of the plot (1=bottom, 2=left, 3=top, 4=right).
`line`	on which MARgin line, starting at 0 counting outwards.
`outer`	use outer margins if available.
`at`	give location in user coordinates. If `length(at)==0` (the default), the location will be determined by `adj`.
`adj`	adjustment for each string in reading direction. For strings parallel to the axes, `adj = 0` means left or bottom alignment, and `adj = 1` means right or top alignment. If `adj` is not a finite value (the default), the value of `par("las")` determines the adjustment. For strings plotted parallel to the axis the default is to centre the string.
`padj`	adjustment for each string perpendicular to the reading direction (which is controlled by `adj`). For strings parallel to the axes, `padj = 0` means right or top alignment, and `padj = 1` means left or bottom alignment. If `padj` is not a finite value (the default), the value of `par("las")` determines the adjustment. For strings plotted perpendicular to the axis the default is to centre the string.

cex	character expansion factor. NULL and NA are equivalent to 1.0. This is an absolute measure, not scaled by par("cex") or by setting par("mfrow") or par("mfcol"). Can be a vector.
col	color to use. Can be a vector. NA values (the default) mean use par("col").
font	font for text. Can be a vector. NA values (the default) mean use par("font").
...	Further graphical parameters (see par), including family, las and xpd. (The latter defaults to the figure region unless outer = TRUE, otherwise the device region. It can only be increased.)

Details

The user coordinates in the outer margins always range from zero to one, and are not affected by the user coordinates in the figure region(s) — R differs here from other implementations of S.

All of the named arguments can be vectors, and recycling will take place to plot as many strings as the longest of the vector arguments.

Note that a vector adj has a different meaning from text. adj = 0.5 will centre the string, but for outer=TRUE on the device region rather than the plot region.

Parameter las will determine the orientation of the string(s). For strings plotted perpendicular to the axis the default justification is to place the end of the string nearest the axis on the specified line. (Note that this differs from S, which uses srt if at is supplied and las if it is not. Parameter srt is ignored in R.)

Note that if the text is to be plotted perpendicular to the axis, adj determines the justification of the string *and* the position along the axis unless at is specified.

Side Effects

The given text is written onto the current plot.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

title, text, plot, par; plotmath for details on mathematical annotation.

Examples

```
plot(1:10, (-4:5)^2, main="Parabola Points", xlab="xlab")
mtext("10 of them")
for(s in 1:4)
    mtext(paste("mtext(..., line= -1, {side, col, font} = ",s,
          ", cex = ", (1+s)/2, ")"), line = -1,
          side=s, col=s, font=s, cex= (1+s)/2)
mtext("mtext(..., line= -2)", line = -2)
mtext("mtext(..., line= -2, adj = 0)", line = -2, adj =0)
##--- log axis :
plot(1:10, exp(1:10), log='y', main="log='y'", xlab="xlab")
for(s in 1:4) mtext(paste("mtext(...,side=",s,")"), side=s)
```

pairs *Scatterplot Matrices*

Description

A matrix of scatterplots is produced.

Usage

```
pairs(x, ...)

## S3 method for class 'formula':
pairs(formula, data = NULL, ..., subset,
      na.action = stats::na.pass)

## Default S3 method:
pairs(x, labels, panel = points, ...,
      lower.panel = panel, upper.panel = panel,
      diag.panel = NULL, text.panel = textPanel,
      label.pos = 0.5 + has.diag/3,
      cex.labels = NULL, font.labels = 1,
      row1attop = TRUE, gap = 1)
```

Arguments

`x`	the coordinates of points given as numeric columns of a matrix or dataframe. Logical and factor columns are converted to numeric in the same way that `data.matrix` does.
`formula`	a formula, such as `~ x + y + z`. Each term will give a separate variable in the pairs plot, so terms should be numeric vectors. (A response will be interpreted as another variable, but not treated specially, so it is confusing to use one.)
`data`	a data.frame (or list) from which the variables in `formula` should be taken.
`subset`	an optional vector specifying a subset of observations to be used for plotting.
`na.action`	a function which indicates what should happen when the data contain `NA`s. The default is to pass missing values on to the panel functions, but `na.action = na.omit` will cause cases with missing values in any of the variables to be omitted entirely.
`labels`	the names of the variables.
`panel`	`function(x,y,...)` which is used to plot the contents of each panel of the display.
`...`	arguments to be passed to or from methods. Also, graphical parameters can be given as can arguments to `plot` such as `main`. `par("oma")` will be set appropriately unless specified.
`lower.panel, upper.panel`	separate panel functions to be used below and above the diagonal respectively.
`diag.panel`	optional `function(x, ...)` to be applied on the diagonals.

`text.panel` optional `function(x, y, labels, cex, font, ...)` to be applied on the diagonals.

`label.pos` `y` position of labels in the text panel.

`cex.labels, font.labels`
graphics parameters for the text panel.

`row1attop` logical. Should the layout be matrix-like with row 1 at the top, or graph-like with row 1 at the bottom?

`gap` Distance between subplots, in margin lines.

Details

The ijth scatterplot contains `x[,i]` plotted against `x[,j]`. The scatterplot can be customised by setting panel functions to appear as something completely different. The off-diagonal panel functions are passed the appropriate columns of `x` as `x` and `y`: the diagonal panel function (if any) is passed a single column, and the `text.panel` function is passed a single `(x, y)` location and the column name.

The graphical parameters `pch` and `col` can be used to specify a vector of plotting symbols and colors to be used in the plots.

The graphical parameter `oma` will be set by `pairs.default` unless supplied as an argument.

A panel function should not attempt to start a new plot, but just plot within a given coordinate system: thus `plot` and `boxplot` are not panel functions.

By default, missing values are passed to the panel functions and will often be ignored within a panel. However, for the formula method and `na.action = na.omit`, all cases which contain a missing values for any of the variables are omitted completely (including when the scales are selected).

Author(s)

Enhancements for R 1.0.0 contributed by Dr. Jens Oehlschlaegel-Akiyoshi and R-core members.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
pairs(iris[1:4], main = "Anderson's Iris Data -- 3 species",
      pch = 21, bg = c("red", "green3", "blue")[unclass(iris$Species)])

## formula method
pairs(~ Fertility + Education + Catholic, data = swiss,
      subset = Education < 20, main = "Swiss data, Education < 20")

pairs(USJudgeRatings)

## put histograms on the diagonal
panel.hist <- function(x, ...)
{
    usr <- par("usr"); on.exit(par(usr))
    par(usr = c(usr[1:2], 0, 1.5) )
    h <- hist(x, plot = FALSE)
```

```
    breaks <- h$breaks; nB <- length(breaks)
    y <- h$counts; y <- y/max(y)
    rect(breaks[-nB], 0, breaks[-1], y, col="cyan", ...)
}
pairs(USJudgeRatings[1:5], panel=panel.smooth,
      cex = 1.5, pch = 24, bg="light blue",
      diag.panel=panel.hist, cex.labels = 2, font.labels=2)

## put (absolute) correlations on the upper panels,
## with size proportional to the correlations.
panel.cor <- function(x, y, digits=2, prefix="", cex.cor)
{
    usr <- par("usr"); on.exit(par(usr))
    par(usr = c(0, 1, 0, 1))
    r <- abs(cor(x, y))
    txt <- format(c(r, 0.123456789), digits=digits)[1]
    txt <- paste(prefix, txt, sep="")
    if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
    text(0.5, 0.5, txt, cex = cex.cor * r)
}
pairs(USJudgeRatings, lower.panel=panel.smooth, upper.panel=panel.cor)
```

`panel.smooth` *Simple Panel Plot*

Description

An example of a simple useful `panel` function to be used as argument in e.g., `coplot` or `pairs`.

Usage

```
panel.smooth(x, y, col = par("col"), bg = NA, pch = par("pch"),
             cex = 1, col.smooth = "red", span = 2/3, iter = 3,
             ...)
```

Arguments

`x, y`	numeric vectors of the same length
`col, bg, pch, cex`	numeric or character codes for the color(s), point type and size of `points`; see also `par`.
`col.smooth`	color to be used by `lines` for drawing the smooths.
`span`	smoothing parameter `f` for `lowess`, see there.
`iter`	number of robustness iterations for `lowess`.
`...`	further arguments to `lines`.

See Also

`coplot` and `pairs` where `panel.smooth` is typically used; `lowess` which does the smoothing.

Examples

```
pairs(swiss, panel = panel.smooth, pch = ".")# emphasize the smooths
pairs(swiss, panel = panel.smooth, lwd = 2, cex= 1.5, col="blue")# hmm...
```

---

par *Set or Query Graphical Parameters*

Description

`par` can be used to set or query graphical parameters. Parameters can be set by specifying them as arguments to `par` in `tag = value` form, or by passing them as a list of tagged values.

Usage

```
par(..., no.readonly = FALSE)

<highlevel plot> (..., <tag> = <value>)
```

Arguments

`...`	arguments in `tag = value` form, or a list of tagged values. The tags must come from the graphical parameters described below.
`no.readonly`	logical; if `TRUE` and there are no other arguments, only parameters are returned which can be set by a subsequent `par()` call *on the same device*.

Details

Each device has its own set of graphical parameters. If the current device is the null device, `par` will open a new device before querying/setting parameters. (What device is controlled by `options("device")`.)

Parameters are queried by giving one or more character vectors to `par`.

`par()` (no arguments) or `par(no.readonly=TRUE)` is used to get *all* the graphical parameters (as a named list). Their names are currently taken from the unexported variable `.Pars`.

R.O. indicates ***read-only arguments***: These may only be used in queries and cannot be set. (`"cin"`, `"cra"`, `"csi"`, `"cxy"` and `"din"` are always read-only.)

There are several parameters can only be set by a call to `par()`:

- `"ask"`,
- `"fig"`, `"fin"`,
- `"lheight"`,
- `"mai"`, `"mar"`, `"mex"`, `"mfcol"`, `"mfrow"`, `"mfg"`,
- `"new"`,
- `"oma"`, `"omd"`, `"omi"`,
- `"pin"`, `"plt"`, `"ps"`, `"pty"`,
- `"usr"`,
- `"xlog"`, `"ylog"`

The remaining parameters can also be set as arguments (often via `...`) to high-level plot functions such as `plot.default`, `plot.window`, `points`, `lines`, `abline`, `axis`, `title`, `text`, `mtext`, `segments`, `symbols`, `arrows`, `polygon`, `rect`, `box`, `contour`, `filled.contour` and `image`. Such settings will be active during the execution of the function, only. However, see the comments on `bg` and `cex`, which may be taken as arguments to certain plot functions rather than as graphical parameters.

The meaning of 'character size' is not well-defined: this is set up for the device taking `pointsize` into account but often not the actual font family in use. Internally the corresponding pars (`cra`, `cin`, `cxy` and `csi`) are used only to set the inter-line spacing used to convert `mar` and `oma` to physical margins. (The same inter-line spacing multiplied by `lheight` is used for multi-line strings in `text` and `strheight`.)

Value

When parameters are set, their former values are returned in an invisible named list. Such a list can be passed as an argument to `par` to restore the parameter values. Use `par(no.readonly = TRUE)` for the full list of parameters that can be restored. However, restoring all of these is not wise since they contain several ways to set the same quantities, and these can have conflicting effects if the graphics device has been resized since the parameters were saved. You will reset all of `mfrow`, `mfcol` and `mfg` and will find `mfrow` wins.

When just one parameter is queried, the value of that parameter is returned as (atomic) vector. When two or more parameters are queried, their values are returned in a list, with the list names giving the parameters.

Note the inconsistency: setting one parameter returns a list, but querying one parameter returns a vector.

Graphical Parameters

`adj` The value of `adj` determines the way in which text strings are justified in `text`, `mtext` and `title`. A value of `0` produces left-justified text, `0.5` (the default) centered text and `1` right-justified text. (Any value in $[0, 1]$ is allowed, and on most devices values outside that interval will also work.) Note that the `adj` argument of `text` also allows `adj = c(x, y)` for different adjustment in x- and y- directions. Note that whereas for `text` it refers to positioning of text about a point, for `mtext` and `title` it controls placement within the plot or device region.

`ann` If set to `FALSE`, high-level plotting functions calling `plot.default` do not annotate the plots they produce with axis titles and overall titles. The default is to do annotation.

`ask` logical. If `TRUE` (and the R session is interactive) the user is asked for input, before a new figure is drawn. As this applies to the device, it also affects output by packages **grid** and **lattice**. It can be set even on non-screen devices but may have no effect there.

This not really a graphics parameter, and its use is deprecated in favour of `devAskNewPage`.

`bg` The color to be used for the background of the device region. When called from `par()` it also sets `new=FALSE`. See section 'Color Specification' for suitable values. For many devices the initial value is set from the `bg` argument of the device, and for the rest it is normally `"white"`.

Note that some graphics functions such as `plot.default` and `points` have an *argument* of this name with a different meaning.

`bty` A character string which determined the type of `box` which is drawn about plots. If `bty` is one of `"o"` (the default), `"l"`, `"7"`, `"c"`, `"u"`, or `"]"` the resulting box resembles the corresponding upper case letter. A value of `"n"` suppresses the box.

cex A numerical value giving the amount by which plotting text and symbols should be magnified relative to the default. Note that some graphics functions such as `plot.default` have an *argument* of this name which *multiplies* this graphical parameter, and some functions such as `points` accept a vector of values which are recycled. Other uses will take just the first value if a vector of length greater than one is supplied.

This starts as `1` when a device is opened, and is reset when the layout is changed, e.g. by setting `mfrow`.

cex.axis The magnification to be used for axis annotation relative to the current setting of `cex`.

cex.lab The magnification to be used for x and y labels relative to the current setting of `cex`.

cex.main The magnification to be used for main titles relative to the current setting of `cex`.

cex.sub The magnification to be used for sub-titles relative to the current setting of `cex`.

cin ***R.O.***; character size `(width, height)` in inches. These are the same measurements as `cra`, expressed in different units.

col A specification for the default plotting color. See section 'Color Specification'. (Some functions such as `lines` accept a vector of values which are recycled. Other uses will take just the first value if a vector of length greater than one is supplied.)

col.axis The color to be used for axis annotation. Defaults to `"black"`.

col.lab The color to be used for x and y labels. Defaults to `"black"`.

col.main The color to be used for plot main titles. Defaults to `"black"`.

col.sub The color to be used for plot sub-titles. Defaults to `"black"`.

cra ***R.O.***; size of default character `(width, height)` in 'rasters' (pixels). Some devices have no concept of pixels and so assume an arbitrary pixel size, usually 1/72 inch. These are the same measurements as `cin`, expressed in different units.

crt A numerical value specifying (in degrees) how single characters should be rotated. It is unwise to expect values other than multiples of 90 to work. Compare with `srt` which does string rotation.

csi ***R.O.***; height of (default-sized) characters in inches. The same as `par("cin")[2]`.

cxy ***R.O.***; size of default character `(width, height)` in user coordinate units. `par("cxy")` is `par("cin")/par("pin")` scaled to user coordinates. Note that `c(strwidth(ch), strwidth(ch))` for a given string `ch` is usually much more precise.

din ***R.O.***; the device dimensions, `(width,height)`, in inches.

err (*Unimplemented*; R is silent when points outside the plot region are *not* plotted.) The degree of error reporting desired.

family The name of a font family for drawing text. The maximum allowed length is 200 bytes. This name gets mapped by each graphics device to a device-specific font description. The default value is `""` which means that the default device fonts will be used (and what those are should be listed on the help page for the device). Standard values are `"serif"`, `"sans"` and `"mono"`, and the Hershey font families are also available. (Different devices may define others, and some devices will ignore this setting completely.) This can be specified inline for `text`.

fg The color to be used for the foreground of plots. This is the default color used for things like axes and boxes around plots. When called from `par()` this also sets parameter `col` to the same value. See section 'Color Specification'. A few devices have an argument to set the initial value, which is otherwise `"black"`.

fig A numerical vector of the form `c(x1, x2, y1, y2)` which gives the (NDC) coordinates of the figure region in the display region of the device. If you set this, unlike S, you start a new plot, so to add to an existing plot use `new=TRUE` as well.

fin The figure region dimensions, `(width,height)`, in inches. If you set this, unlike S, you start a new plot.

font An integer which specifies which font to use for text. If possible, device drivers arrange so that 1 corresponds to plain text (the default), 2 to bold face, 3 to italic and 4 to bold italic. Also, font 5 is expected to be the symbol font, in Adobe symbol encoding. On some devices font families can be selected by `family` to choose different sets of 5 fonts.

font.axis The font to be used for axis annotation.

font.lab The font to be used for x and y labels.

font.main The font to be used for plot main titles.

font.sub The font to be used for plot sub-titles.

lab A numerical vector of the form `c(x, y, len)` which modifies the default way that axes are annotated. The values of `x` and `y` give the (approximate) number of tickmarks on the x and y axes and `len` specifies the label length. The default is `c(5, 5, 7)`. Note that this only affects the way the parameters `xaxp` and `yaxp` are set when the user coordinate system is set up, and is not consulted when axes are drawn. `len` *is unimplemented* in R.

las numeric in {0,1,2,3}; the style of axis labels.

- **0:** always parallel to the axis [*default*],
- **1:** always horizontal,
- **2:** always perpendicular to the axis,
- **3:** always vertical.

Also supported by `mtext`. Note that other string/character rotation (via argument `srt` to `par`) does *not* affect the axis labels.

lend The line end style. This can be specified as an integer or string:

- **0** and `"round"` mean rounded line caps [*default*];
- **1** and `"butt"` mean butt line caps;
- **2** and `"square"` mean square line caps.

lheight The line height multiplier. The height of a line of text (used to vertically space multi-line text) is found by multiplying the character height both by the current character expansion and by the line height multiplier. Default value is 1. Used in `text` and `strheight`.

ljoin The line join style. This can be specified as an integer or string:

- **0** and `"round"` mean rounded line joins [*default*];
- **1** and `"mitre"` mean mitred line joins;
- **2** and `"bevel"` mean bevelled line joins.

lmitre The line mitre limit. This controls when mitred line joins are automatically converted into bevelled line joins. The value must be larger than 1 and the default is 10. Not all devices will honour this setting.

lty The line type. Line types can either be specified as an integer (0=blank, 1=solid (default), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash) or as one of the character strings `"blank"`, `"solid"`, `"dashed"`, `"dotted"`, `"dotdash"`, `"longdash"`, or `"twodash"`, where `"blank"` uses 'invisible lines' (i.e., does not draw them).

Alternatively, a string of up to 8 characters (from `c(1:9, "A":"F")`) may be given, giving the length of line segments which are alternatively drawn and skipped. See section 'Line Type Specification'.

Some functions such as `lines` accept a vector of values which are recycled. Other uses will take just the first value if a vector of length greater than one is supplied.

lwd The line width, a *positive* number, defaulting to `1`. The interpretation is device-specific, and some devices do not implement line widths less than one. (See the help on the device for details of the interpretation.)

Some functions such as `lines` accept a vector of values which are recycled. Other uses will take just the first value if a vector of length greater than one is supplied.

mai A numerical vector of the form `c(bottom, left, top, right)` which gives the margin size specified in inches.

mar A numerical vector of the form `c(bottom, left, top, right)` which gives the number of lines of margin to be specified on the four sides of the plot. The default is `c(5, 4, 4, 2) + 0.1`.

mex `mex` is a character size expansion factor which is used to describe coordinates in the margins of plots. Note that this does not change the font size, rather specifies the size of font (as a multiple of `csi`) used to convert between `mar` and `mai`, and between `oma` and `omi`.

This starts as `1` when the device is opened, and is reset when the layout is changed (alongside resetting `cex`).

mfcol, mfrow A vector of the form `c(nr, nc)`. Subsequent figures will be drawn in an `nr`-by-`nc` array on the device by *columns* (`mfcol`), or *rows* (`mfrow`), respectively.

In a layout with exactly two rows and columns the base value of `"cex"` is reduced by a factor of 0.83: if there are three or more of either rows or columns, the reduction factor is 0.66.

Setting a layout resets the base value of `cex` and that of `mex` to `1`.

If either of these is queried it will give the current layout, so querying cannot tell you the order the array will be filled.

Consider the alternatives, `layout` and `split.screen`.

mfg A numerical vector of the form `c(i, j)` where `i` and `j` indicate which figure in an array of figures is to be drawn next (if setting) or is being drawn (if enquiring). The array must already have been set by `mfcol` or `mfrow`.

For compatibility with S, the form `c(i, j, nr, nc)` is also accepted, when `nr` and `nc` should be the current number of rows and number of columns. Mismatches will be ignored, with a warning.

mgp The margin line (in `mex` units) for the axis title, axis labels and axis line. Note that `mgp[1]` affects `title` whereas `mgp[2:3]` affect `axis`. The default is `c(3, 1, 0)`.

mkh The height in inches of symbols to be drawn when the value of `pch` is an integer. *Completely ignored currently*.

new logical, defaulting to `FALSE`. If set to `TRUE`, the next high-level plotting command (actually `plot.new`) should *not clean* the frame before drawing *as if it was on a* **new** *device*. It is an error (ignored with a warning) to try to use `new=TRUE` on a device that does not currently contain a high-level plot.

oma A vector of the form `c(bottom, left, top, right)` giving the size of the outer margins in lines of text.

omd A vector of the form `c(x1, x2, y1, y2)` giving the region *inside* outer margins in NDC (= normalized device coordinates), i.e., as fraction (in $[0, 1]$) of the device region.

omi A vector of the form `c(bottom, left, top, right)` giving the size of the outer margins in inches.

pch Either an integer specifying a symbol or a single character to be used as the default in plotting points. See `points` for possible values and their interpretation. Note that only integers and single-character strings can be set as a graphics parameter (and not `NA` nor `NULL`).

pin The current plot dimensions, `(width,height)`, in inches.

plt A vector of the form `c(x1, x2, y1, y2)` giving the coordinates of the plot region as fractions of the current figure region.

ps integer; the point size of text (but not symbols). Unlike the `pointsize` argument of most devices, this does not change the relationship between `mar` and `mai` (nor `oma` and `omi`).

What is meant by 'point size' is device-specific, but most devices mean a multiple of 1bp, that is 1/72 of an inch.

pty A character specifying the type of plot region to be used; `"s"` generates a square plotting region and `"m"` generates the maximal plotting region.

smo (*Unimplemented*) a value which indicates how smooth circles and circular arcs should be.

srt The string rotation in degrees. See the comment about `crt`. Only supported by `text`.

tck The length of tick marks as a fraction of the smaller of the width or height of the plotting region. If `tck >= 0.5` it is interpreted as a fraction of the relevant side, so if `tck = 1` grid lines are drawn. The default setting (`tck = NA`) is to use `tcl = -0.5`.

tcl The length of tick marks as a fraction of the height of a line of text. The default value is `-0.5`; setting `tcl = NA` sets `tck = -0.01` which is S' default.

usr A vector of the form `c(x1, x2, y1, y2)` giving the extremes of the user coordinates of the plotting region. When a logarithmic scale is in use (i.e., `par("xlog")` is true, see below), then the x-limits will be `10 ^ par("usr")[1:2]`. Similarly for the y-axis.

xaxp A vector of the form `c(x1, x2, n)` giving the coordinates of the extreme tick marks and the number of intervals between tick-marks when `par("xlog")` is false. Otherwise, when *log* coordinates are active, the three values have a different meaning: For a small range, `n` is *negative*, and the ticks are as in the linear case, otherwise, `n` is in `1:3`, specifying a case number, and `x1` and `x2` are the lowest and highest power of 10 inside the user coordinates, `10 ^ par("usr")[1:2]`. (The `"usr"` coordinates are log10-transformed here!)

n=1 will produce tick marks at 10^j for integer j,

n=2 gives marks $k10^j$ with $k \in \{1, 5\}$,

n=3 gives marks $k10^j$ with $k \in \{1, 2, 5\}$.

See `axTicks()` for a pure R implementation of this.

This parameter is reset when a user coordinate system is set up, for example by starting a new page or by calling `plot.window` or setting `par("usr")`: `n` is taken from `par("lab")`. It affects the default behaviour of subsequent calls to `axis` for sides 1 or 3.

xaxs The style of axis interval calculation to be used for the x-axis. Possible values are `"r"`, `"i"`, `"e"`, `"s"`, `"d"`. The styles are generally controlled by the range of data or `xlim`, if given. Style `"r"` (regular) first extends the data range by 4 percent at each end and then finds an axis with pretty labels that fits within the extended range. Style `"i"` (internal) just finds an axis with pretty labels that fits within the original data range. Style `"s"` (standard) finds an axis with pretty labels within which the original data range fits. Style `"e"` (extended) is like style `"s"`, except that it is also ensures that there is room for plotting symbols within the bounding box. Style `"d"` (direct) specifies that the current axis should be used on subsequent plots. (*Only `"r"` and `"i"` styles are currently implemented*)

xaxt A character which specifies the x axis type. Specifying `"n"` suppresses plotting of the axis. The standard value is `"s"`: for compatibility with S values `"l"` and `"t"` are accepted but are equivalent to `"s"`: any value other than `"n"` implies plotting.

xlog A logical value (see `log` in `plot.default`). If `TRUE`, a logarithmic scale is in use (e.g., after `plot(*, log = "x")`). For a new device, it defaults to `FALSE`, i.e., linear scale.

xpd A logical value or `NA`. If `FALSE`, all plotting is clipped to the plot region, if `TRUE`, all plotting is clipped to the figure region, and if `NA`, all plotting is clipped to the device region. See also `clip`.

yaxp A vector of the form `c(y1, y2, n)` giving the coordinates of the extreme tick marks and the number of intervals between tick-marks unless for log coordinates, see `xaxp` above.

yaxs The style of axis interval calculation to be used for the y-axis. See `xaxs` above.

yaxt A character which specifies the y axis type. Specifying `"n"` suppresses plotting.

ylog A logical value; see `xlog` above.

Color Specification

Colors can be specified in several different ways. The simplest way is with a character string giving the color name (e.g., `"red"`). A list of the possible colors can be obtained with the function `colors`. Alternatively, colors can be specified directly in terms of their RGB components with a string of the form `"#RRGGBB"` where each of the pairs `RR`, `GG`, `BB` consist of two hexadecimal digits giving a value in the range `00` to `FF`. Colors can also be specified by giving an index into a small table of colors, the `palette`. This provides compatibility with S. Index `0` corresponds to the background color. (Because apparently some people have been assuming it, it is also possible to specify integers as character strings, e.g. `"3"`.)

Additionally, `"transparent"` or (integer) `NA` is *transparent*, useful for filled areas (such as the background!), and just invisible for things like lines or text. Semi-transparent colors are available for use on devices that support them.

The functions `rgb`, `hsv`, `hcl`, `gray` and `rainbow` provide additional ways of generating colors.

Line Type Specification

Line types can either be specified by giving an index into a small built-in table of line types (1 = solid, 2 = dashed, etc, see `lty` above) or directly as the lengths of on/off stretches of line. This is done with a string of an even number (up to eight) of characters, namely *non-zero* (hexadecimal) digits which give the lengths in consecutive positions in the string. For example, the string `"33"` specifies three units on followed by three off and `"3313"` specifies three units on followed by three off followed by one on and finally three off. The 'units' here are (on most devices) proportional to `lwd`, and with `lwd = 1` are in pixels or points or 1/96 inch.

The five standard dash-dot line types (`lty = 2:6`) correspond to `c("44", "13", "1343", "73", "2262")`.

Note that `NA` is not a valid value for `lty`.

Note

The effect of restoring all the (settable) graphics parameters as in the examples is hard to predict if the device has been resized. Several of them are attempting to set the same things in different ways, and those last in the alphabet will win. In particular, the settings of `mai`, `mar`, `pin`, `plt` and `pty` interact, as do the outer margin settings, the figure layout and figure region size.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`plot.default` for some high-level plotting parameters; `colors`; `clip`; `options` for other setup parameters; graphic devices `x11`, `postscript` and setting up device regions by `layout` and `split.screen`.

Examples

```
op <- par(mfrow = c(2, 2), # 2 x 2 pictures on one plot
          pty = "s")       # square plotting region,
                           # independent of device size

## At end of plotting, reset to previous settings:
par(op)

## Alternatively,
op <- par(no.readonly = TRUE) # the whole list of settable par's.
## do lots of plotting and par(.) calls, then reset:
par(op)
## Note this is not in general good practice

par("ylog") # FALSE
plot(1 : 12, log = "y")
par("ylog") # TRUE

plot(1:2, xaxs = "i") # 'inner axis' w/o extra space
par(c("usr", "xaxp"))

( nr.prof <-
  c(prof.pilots=16,lawyers=11,farmers=10,salesmen=9,physicians=9,
    mechanics=6,policemen=6,managers=6,engineers=5,teachers=4,
    housewives=3,students=3,armed.forces=1))
par(las = 3)
barplot(rbind(nr.prof)) # R 0.63.2: shows alignment problem
par(las = 0)# reset to default

require(grDevices) # for gray
## 'fg' use:
plot(1:12, type = "b", main="'fg' : axes, ticks and box in gray",
     fg = gray(0.7), bty="7" , sub=R.version.string)

ex <- function() {
   old.par <- par(no.readonly = TRUE) # all par settings which
                                      # could be changed.
   on.exit(par(old.par))
   ## ...
   ## ... do lots of par() settings and plots
   ## ...
   invisible() #-- now,  par(old.par)  will be executed
}
ex()
```

`persp` *Perspective Plots*

Description

This function draws perspective plots of surfaces over the x–y plane. `persp` is a generic function.

Usage

```
persp(x, ...)

## Default S3 method:
persp(x = seq(0, 1, length.out = nrow(z)),
      y = seq(0, 1, length.out = ncol(z)),
      z, xlim = range(x), ylim = range(y),
      zlim = range(z, na.rm = TRUE),
      xlab = NULL, ylab = NULL, zlab = NULL,
      main = NULL, sub = NULL,
      theta = 0, phi = 15, r = sqrt(3), d = 1,
      scale = TRUE, expand = 1,
      col = "white", border = NULL, ltheta = -135, lphi = 0,
      shade = NA, box = TRUE, axes = TRUE, nticks = 5,
      ticktype = "simple", ...)
```

Arguments

`x, y`	locations of grid lines at which the values in `z` are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If `x` is a `list`, its components `x$x` and `x$y` are used for `x` and `y`, respectively.
`z`	a matrix containing the values to be plotted (`NA`s are allowed). Note that `x` can be used instead of `z` for convenience.
`xlim, ylim, zlim`	x-, y- and z-limits. The plot is produced so that the rectangular volume defined by these limits is visible.
`xlab, ylab, zlab`	titles for the axes. N.B. These must be character strings; expressions are not accepted. Numbers will be coerced to character strings.
`main, sub`	main and sub title, as for `title`.
`theta, phi`	angles defining the viewing direction. `theta` gives the azimuthal direction and `phi` the colatitude.
`r`	the distance of the eyepoint from the centre of the plotting box.
`d`	a value which can be used to vary the strength of the perspective transformation. Values of `d` greater than 1 will lessen the perspective effect and values less and 1 will exaggerate it.
`scale`	before viewing the x, y and z coordinates of the points defining the surface are transformed to the interval [0,1]. If `scale` is `TRUE` the x, y and z coordinates are transformed separately. If `scale` is `FALSE` the coordinates are scaled so that aspect ratios are retained. This is useful for rendering things like DEM information.
`expand`	a expansion factor applied to the `z` coordinates. Often used with `0 < expand < 1` to shrink the plotting box in the `z` direction.
`col`	the color(s) of the surface facets. Transparent colours are ignored. This is recycled to the $(nx-1)(ny-1)$ facets.
`border`	the color of the line drawn around the surface facets. The default, `NULL`, corresponds to `par("fg")`. A value of `NA` will disable the drawing of borders: this is sometimes useful when the surface is shaded.

ltheta, lphi	if finite values are specified for ltheta and lphi, the surface is shaded as though it was being illuminated from the direction specified by azimuth ltheta and colatitude lphi.
shade	the shade at a surface facet is computed as ((1+d)/2)^shade, where d is the dot product of a unit vector normal to the facet and a unit vector in the direction of a light source. Values of shade close to one yield shading similar to a point light source model and values close to zero produce no shading. Values in the range 0.5 to 0.75 provide an approximation to daylight illumination.
box	should the bounding box for the surface be displayed. The default is TRUE.
axes	should ticks and labels be added to the box. The default is TRUE. If box is FALSE then no ticks or labels are drawn.
ticktype	character: "simple" draws just an arrow parallel to the axis to indicate direction of increase; "detailed" draws normal ticks as per 2D plots.
nticks	the (approximate) number of tick marks to draw on the axes. Has no effect if ticktype is "simple".
...	additional graphical parameters (see par).

Details

The plots are produced by first transforming the coordinates to the interval [0,1]. The surface is then viewed by looking at the origin from a direction defined by theta and phi. If theta and phi are both zero the viewing direction is directly down the negative y axis. Changing theta will vary the azimuth and changing phi the colatitude.

There is a hook called "persp" (see setHook) called after the plot is completed, which is used in the testing code to annotate the plot page. The hook function(s) are called with no argument.

Notice that persp interprets the z matrix as a table of f(x[i], y[j]) values, so that the x axis corresponds to row number and the y axis to column number, with column 1 at the bottom, so that with the standard rotation angles, the top left corner of the matrix is displayed at the left hand side, closest to the user.

The sizes and fonts of the axis labels and the annotations for ticktype="detailed" are controlled by graphics parameters "cex.lab"/"font.lab" and "cex.axis"/"font.axis" respectively. (This changed in R 2.5.0.)

Value

persp() returns the *viewing transformation matrix*, say VT, a 4×4 matrix suitable for projecting 3D coordinates (x, y, z) into the 2D plane using homogeneous 4D coordinates (x, y, z, t). It can be used to superimpose additional graphical elements on the 3D plot, by lines() or points(), using the simple function trans3d().

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

contour and image; trans3d.

Examples

```
require(grDevices) # for trans3d
## More examples in  demo(persp) !!
##                   -----------

# (1) The Obligatory Mathematical surface.
#     Rotated sinc function.

x <- seq(-10, 10, length= 30)
y <- x
f <- function(x,y) { r <- sqrt(x^2+y^2); 10 * sin(r)/r }
z <- outer(x, y, f)
z[is.na(z)] <- 1
op <- par(bg = "white")
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue")
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue",
      ltheta = 120, shade = 0.75, ticktype = "detailed",
      xlab = "X", ylab = "Y", zlab = "Sinc( r )"
) -> res
round(res, 3)

# (2) Add to existing persp plot - using trans3d() :

xE <- c(-10,10); xy <- expand.grid(xE, xE)
points(trans3d(xy[,1], xy[,2], 6, pmat = res), col = 2, pch =16)
lines (trans3d(x, y=10, z= 6 + sin(x), pmat = res), col = 3)

phi <- seq(0, 2*pi, len = 201)
r1 <- 7.725 # radius of 2nd maximum
xr <- r1 * cos(phi)
yr <- r1 * sin(phi)
lines(trans3d(xr,yr, f(xr,yr), res), col = "pink", lwd = 2)
## (no hidden lines)

# (3) Visualizing a simple DEM model

z <- 2 * volcano        # Exaggerate the relief
x <- 10 * (1:nrow(z))   # 10 meter spacing (S to N)
y <- 10 * (1:ncol(z))   # 10 meter spacing (E to W)
## Don't draw the grid lines :  border = NA
par(bg = "slategray")
persp(x, y, z, theta = 135, phi = 30, col = "green3", scale = FALSE,
      ltheta = -120, shade = 0.75, border = NA, box = FALSE)

# (4) Surface colours corresponding to z-values

par(bg = "white")
x <- seq(-1.95, 1.95, length = 30)
y <- seq(-1.95, 1.95, length = 35)
z <- outer(x, y, function(a,b) a*b^2)
nrz <- nrow(z)
ncz <- ncol(z)
# Create a function interpolating colors in the range of specified colors
jet.colors <- colorRampPalette( c("blue", "green") )
# Generate the desired number of colors from this palette
nbcol <- 100
```

```
color <- jet.colors(nbcol)
# Compute the z-value at the facet centres
zfacet <- z[-1, -1] + z[-1, -ncz] + z[-nrz, -1] + z[-nrz, -ncz]
# Recode facet z-values into color indices
facetcol <- cut(zfacet, nbcol)
persp(x, y, z, col=color[facetcol], phi=30, theta=-30)

par(op)
```

`pie` *Pie Charts*

Description

Draw a pie chart.

Usage

```
pie(x, labels = names(x), edges = 200, radius = 0.8,
    clockwise = FALSE, init.angle = if(clockwise) 90 else 0,
    density = NULL, angle = 45, col = NULL, border = NULL,
    lty = NULL, main = NULL, ...)
```

Arguments

`x`	a vector of non-negative numerical quantities. The values in `x` are displayed as the areas of pie slices.
`labels`	one or more expressions or character strings giving names for the slices. Other objects are coerced by `as.graphicsAnnot`. For empty or `NA` (after coercion to character) labels, no label nor pointing line is drawn.
`edges`	the circular outline of the pie is approximated by a polygon with this many edges.
`radius`	the pie is drawn centered in a square box whose sides range from -1 to 1. If the character strings labeling the slices are long it may be necessary to use a smaller radius.
`clockwise`	logical indicating if slices are drawn clockwise or counter clockwise (i.e., mathematically positive direction), the latter is default.
`init.angle`	number specifying the *starting angle* (in degrees) for the slices. Defaults to 0 (i.e., '3 o'clock') unless `clockwise` is true where `init.angle` defaults to 90 (degrees), (i.e., '12 o'clock').
`density`	the density of shading lines, in lines per inch. The default value of `NULL` means that no shading lines are drawn. Non-positive values of `density` also inhibit the drawing of shading lines.
`angle`	the slope of shading lines, given as an angle in degrees (counter-clockwise).
`col`	a vector of colors to be used in filling or shading the slices. If missing a set of 6 pastel colours is used, unless `density` is specified when `par("fg")` is used.
`border, lty`	(possibly vectors) arguments passed to `polygon` which draws each slice.
`main`	an overall title for the plot.
`...`	graphical parameters can be given as arguments to `pie`. They will affect the main title and labels only.

Note

Pie charts are a very bad way of displaying information. The eye is good at judging linear measures and bad at judging relative areas. A bar chart or dot chart is a preferable way of displaying this type of data.

Cleveland (1985), page 264: "Data that can be shown by pie charts always can be shown by a dot chart. This means that judgements of position along a common scale can be made instead of the less accurate angle judgements." This statement is based on the empirical investigations of Cleveland and McGill as well as investigations by perceptual psychologists.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1985) *The elements of graphing data*. Wadsworth: Monterey, CA, USA.

See Also

`dotchart`.

Examples

```
require(grDevices)
pie(rep(1, 24), col = rainbow(24), radius = 0.9)

pie.sales <- c(0.12, 0.3, 0.26, 0.16, 0.04, 0.12)
names(pie.sales) <- c("Blueberry", "Cherry",
    "Apple", "Boston Cream", "Other", "Vanilla Cream")
pie(pie.sales) # default colours
pie(pie.sales,
    col = c("purple", "violetred1", "green3", "cornsilk", "cyan", "white"))
pie(pie.sales, col = gray(seq(0.4,1.0,length=6)))
pie(pie.sales, density = 10, angle = 15 + 10 * 1:6)
pie(pie.sales, clockwise=TRUE, main="pie(*, clockwise=TRUE)")
segments(0,0, 0,1, col= "red", lwd = 2)
text(0,1, "init.angle = 90", col= "red")

n <- 200
pie(rep(1,n), labels="", col=rainbow(n), border=NA,
    main = "pie(*, labels=\"\", col=rainbow(n), border=NA,..")
```

`plot` *Generic X-Y Plotting*

Description

Generic function for plotting of R objects. For more details about the graphical parameter arguments, see `par`.

Usage

```
plot(x, y, ...)
```

Arguments

`x`	the coordinates of points in the plot. Alternatively, a single plotting structure, function or *any* R *object with a* `plot` *method* can be provided.
`y`	the y coordinates of points in the plot, *optional* if `x` is an appropriate structure.
`...`	Arguments to be passed to methods, such as graphical parameters (see `par`). Many methods will accept the following arguments:

`type` what type of plot should be drawn. Possible types are

- `"p"` for **p**oints,
- `"l"` for **l**ines,
- `"b"` for **b**oth,
- `"c"` for the lines part alone of `"b"`,
- `"o"` for both '**o**verplotted',
- `"h"` for '**h**istogram' like (or 'high-density') vertical lines,
- `"s"` for stair **s**teps,
- `"S"` for other **s**teps, see 'Details' below,
- `"n"` for no plotting.

All other `types` give a warning or an error; using, e.g., `type = "punkte"` being equivalent to `type = "p"` for S compatibility.

`main` an overall title for the plot: see `title`.

`sub` a sub title for the plot: see `title`.

`xlab` a title for the x axis: see `title`.

`ylab` a title for the y axis: see `title`.

`asp` the y/x aspect ratio, see `plot.window`.

Details

For simple scatter plots, `plot.default` will be used. However, there are `plot` methods for many R objects, including `function`s, `data.frame`s, `density` objects, etc. Use `methods(plot)` and the documentation for these.

The two step types differ in their x-y preference: Going from $(x1, y1)$ to $(x2, y2)$ with $x1 < x2$, `type = "s"` moves first horizontal, then vertical, whereas `type = "S"` moves the other way around.

See Also

`plot.default`, `plot.formula` and other methods; `points`, `lines`, `par`.

Examples

```
require(stats)
plot(cars)
lines(lowess(cars))

plot(sin, -pi, 2*pi)

## Discrete Distribution Plot:
plot(table(rpois(100,5)), type = "h", col = "red", lwd=10,
     main="rpois(100,lambda=5)")

## Simple quantiles/ECDF, see ecdf() {library(stats)} for a better one:
```

```
plot(x <- sort(rnorm(47)), type = "s", main = "plot(x, type = \"s\")")
points(x, cex = .5, col = "dark red")
```

`plot.data.frame` *Plot Method for Data Frames*

Description

`plot.data.frame`, a method for the `plot` generic. It is designed for a quick look at numeric data frames.

Usage

```
## S3 method for class 'data.frame':
plot(x, ...)
```

Arguments

`x`	object of class `data.frame`.
`...`	further arguments to `stripchart`, `plot.default` or `pairs`.

Details

This is intended for data frames with *numeric* columns. For more than two columns it first calls `data.matrix` to convert the data frame to a numeric matrix and then calls `pairs` to produce a scatterplot matrix). This can fail and may well be inappropriate: for example numerical conversion of dates will lose their special meaning and a warning will be given.

For a two-column data frame it plots the second column against the first by the most appropriate method for the first column.

For a single numeric column it uses `stripchart`, and for other single-column data frames tries to find a plot method for the single column.

See Also

`data.frame`

Examples

```
plot(OrchardSprays[1], method="jitter")
plot(OrchardSprays[c(4,1)])
plot(OrchardSprays)

plot(iris)
plot(iris[5:4])
plot(women)
```

`plot.default` *The Default Scatterplot Function*

Description

Draw a scatter plot with decorations such as axes and titles in the active graphics window.

Usage

```
## Default S3 method:
plot(x, y = NULL, type = "p",  xlim = NULL, ylim = NULL,
     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
     ann = par("ann"), axes = TRUE, frame.plot = axes,
     panel.first = NULL, panel.last = NULL, asp = NA, ...)
```

Arguments

`x, y`	the `x` and `y` arguments provide the x and y coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function `xy.coords` for details. If supplied separately, they must be of the same length.
`type`	1-character string giving the type of plot desired. The following values are possible, for details, see `plot`: `"p"` for points, `"l"` for lines, `"o"` for overplotted points and lines, `"b"`, `"c"`) for (empty if `"c"`) points joined by lines, `"s"` and `"S"` for stair steps and `"h"` for histogram-like vertical lines. Finally, `"n"` does not produce any points or lines.
`xlim`	the x limits (x1, x2) of the plot. Note that `x1 > x2` is allowed and leads to a 'reversed axis'.
`ylim`	the y limits of the plot.
`log`	a character string which contains `"x"` if the x axis is to be logarithmic, `"y"` if the y axis is to be logarithmic and `"xy"` or `"yx"` if both axes are to be logarithmic.
`main`	a main title for the plot, see also `title`.
`sub`	a sub title for the plot.
`xlab`	a label for the x axis, defaults to a description of `x`.
`ylab`	a label for the y axis, defaults to a description of `y`.
`ann`	a logical value indicating whether the default annotation (title and x and y axis labels) should appear on the plot.
`axes`	a logical value indicating whether both axes should be drawn on the plot. Use graphical parameter `"xaxt"` or `"yaxt"` to suppress just one of the axes.
`frame.plot`	a logical indicating whether a box should be drawn around the plot.
`panel.first`	an expression to be evaluated after the plot axes are set up but before any plotting takes place. This can be useful for drawing background grids or scatterplot smooths.
`panel.last`	an expression to be evaluated after plotting has taken place.
`asp`	the y/x aspect ratio, see `plot.window`.
`...`	other graphical parameters (see `par` and section 'Details' below).

Details

Commonly used graphical parameters are:

col The colors for lines and points. Multiple colors can be specified so that each point can be given its own color. If there are fewer colors than points they are recycled in the standard fashion. Lines will all be plotted in the first colour specified.

bg a vector of background colors for open plot symbols, see points. Note: this is **not** the same setting as par("bg").

pch a vector of plotting characters or symbols: see points.

cex a numerical vector giving the amount by which plotting characters and symbols should be scaled relative to the default. This works as a multiple of par("cex"). NULL and NA are equivalent to 1.0. Note that this does not affect annotation: see below.

lty the line type, see par.

cex.main, col.lab, font.sub, etc settings for main- and sub-title and axis annotation, see title and par.

lwd the line width, see par.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1985) *The Elements of Graphing Data.* Monterey, CA: Wadsworth.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

plot, plot.window, xy.coords.

Examples

```
Speed <- cars$speed
Distance <- cars$dist
plot(Speed, Distance, panel.first = grid(8,8),
     pch = 0, cex = 1.2, col = "blue")
plot(Speed, Distance,
     panel.first = lines(stats::lowess(Speed, Distance), lty = "dashed"),
     pch = 0, cex = 1.2, col = "blue")

## Show the different plot types
x <- 0:12
y <- sin(pi/5 * x)
op <- par(mfrow = c(3,3), mar = .1+ c(2,2,3,1))
for (tp in c("p","l","b",  "c","o","h",  "s","S","n")) {
   plot(y ~ x, type = tp,
        main = paste("plot(*, type = \"",tp,"\")",sep=""))
   if(tp == "S") {
      lines(x,y, type = "s", col = "red", lty = 2)
      mtext("lines(*, type = \"s\", ...)", col = "red", cex=.8)
   }
}
par(op)
```

```
##--- Log-Log Plot  with  custom axes
lx <- seq(1,5, length=41)
yl <- expression(e^{-frac(1,2) * {log[10](x)}^2})
y <- exp(-.5*lx^2)
op <- par(mfrow=c(2,1), mar=par("mar")+c(0,1,0,0))
plot(10^lx, y, log="xy", type="l", col="purple",
     main="Log-Log plot", ylab=yl, xlab="x")
plot(10^lx, y, log="xy", type="o", pch='.', col="forestgreen",
     main="Log-Log plot with custom axes", ylab=yl, xlab="x",
     axes = FALSE, frame.plot = TRUE)
my.at <- 10^(1:5)
axis(1, at = my.at, labels = formatC(my.at, format="fg"))
at.y <- 10^(-5:-1)
axis(2, at = at.y, labels = formatC(at.y, format="fg"), col.axis="red")
par(op)
```

`plot.design` *Plot Univariate Effects of a 'Design' or Model*

Description

Plot univariate effects of one ore more `factor`s, typically for a designed experiment as analyzed by `aov()`. Further, in S this a method of the `plot` generic function for `design` objects.

Usage

```
plot.design(x, y = NULL, fun = mean, data = NULL, ...,
            ylim = NULL, xlab = "Factors", ylab = NULL,
            main = NULL, ask = NULL, xaxt = par("xaxt"),
            axes = TRUE, xtick = FALSE)
```

Arguments

`x`	either a data frame containing the design factors and optionally the response, or a `formula` or `terms` object.
`y`	the response, if not given in x.
`fun`	a function (or name of one) to be applied to each subset. It must return one number for a numeric (vector) input.
`data`	data frame containing the variables referenced by `x` when that is formula like.
`...`	graphical arguments such as `col`, see `par`.
`ylim`	range of y values, as in `plot.default`.
`xlab`	x axis label, see `title`.
`ylab`	y axis label with a 'smart' default.
`main`	main title, see `title`.
`ask`	logical indicating if the user should be asked before a new page is started – in the case of multiple y's.
`xaxt`	character giving the type of x axis.
`axes`	logical indicating if axes should be drawn.
`xtick`	logical indicating if ticks (one per factor) should be drawn on the x axis.

Details

The supplied function will be called once for each level of each factor in the design and the plot will show these summary values. The levels of a particular factor are shown along a vertical line, and the overall value of `fun()` for the response is drawn as a horizontal line.

This is a new R implementation which will not be completely compatible to the earlier S implementations. This is not a bug but might still change.

Note

A big effort was taken to make this closely compatible to the S version. However, `col` (and `fg`) specification has different effects.

Author(s)

Roberto Frisullo and Martin Maechler

References

Chambers, J. M. and Hastie, T. J. eds (1992) *Statistical Models in S*. Chapman & Hall, London, **the** *white book*, pp. 546–7 (and 163–4).

Freeny, A. E. and Landwehr, J. M. (1990) Displays for data from large designed experiments; Computer Science and Statistics: Proc. 22nd SympİnterFace, 117–126, Springer Verlag.

See Also

`interaction.plot` for a 'standard graphic' of designed experiments.

Examples

```
require(stats)
plot.design(warpbreaks)# automatic for data frame with one numeric var.

Form <- breaks ~ wool + tension
summary(fm1 <- aov(Form, data = warpbreaks))
plot.design(       Form, data = warpbreaks, col = 2)# same as above

## More than one y :
utils::str(esoph)
plot.design(esoph) ## two plots; if interactive you are "ask"ed

## or rather, compare mean and median:
op <- par(mfcol = 1:2)
plot.design(ncases/ncontrols ~ ., data = esoph, ylim = c(0, 0.8))
plot.design(ncases/ncontrols ~ ., data = esoph, ylim = c(0, 0.8),
            fun = median)
par(op)
```

plot.factor *Plotting Factor Variables*

Description

This functions implements a scatterplot method for `factor` arguments of the *generic* `plot` function. Actually, `boxplot` is used when `y` is numeric and a `spineplot` when `y` is a factor. For a single factor `x` (i.e., with `y` missing) a simple `barplot` is produced.

Usage

```
## S3 method for class 'factor':
plot(x, y, legend.text = NULL, ...)
```

Arguments

`x, y`	numeric or factor. `y` may be missing.
`legend.text`	character vector for annotation of y axis, defaults to `levels(y)`. Can be used instead of `yaxlabels` (for backward compatibility).
`...`	Further arguments to `plot`, see also `par`.

See Also

`plot.default`, `plot.formula`, `barplot`, `boxplot`, `spineplot`.

Examples

```
require(grDevices)
plot(PlantGrowth)                                  # -> plot.data.frame
plot(weight ~ group, data = PlantGrowth)           # numeric vector ~ factor
plot(cut(weight, 2) ~ group, data = PlantGrowth) # factor ~ factor
## passing "..." to spineplot() eventually:
plot(cut(weight, 3) ~ group, data = PlantGrowth,
     col = hcl(c(0, 120, 240), 50, 70))

plot(PlantGrowth$group, axes=FALSE, main="no axes")# extremely silly
```

plot.formula *Formula Notation for Scatterplots*

Description

Specify a scatterplot or add points or lines via a formula.

Usage

```
## S3 method for class 'formula':
plot(formula, data = parent.frame(), ..., subset,
             ylab = varnames[response], ask = dev.interactive())

## S3 method for class 'formula':
points(formula, data = parent.frame(), ..., subset)

## S3 method for class 'formula':
lines(formula, data = parent.frame(), ..., subset)
```

Arguments

formula	a formula, such as y ~ x.
data	a data.frame (or list) from which the variables in formula should be taken.
...	Arguments to be passed to or from other methods. horizontal = TRUE is also accepted.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
ylab	the y label of the plot(s).
ask	logical, see par.

Details

Both the terms in the formula and the ... arguments are evaluated in data enclosed in parent.frame() if data is a list or a data frame. The terms of the formula and those arguments in ... that are of the same length as data are subjected to the subsetting specified in subset. If the formula in plot.formula contains more than one non-response term, a series of plots of y against each term is given. A plot against the running index can be specified as plot(y ~ 1).

Missing values are not considered in these methods, and in particular cases with missing values are not removed.

If y is an object (i.e. has a class attribute) then plot.formula looks for a plot method for that class first. Otherwise, the class of x will determine the type of the plot. For factors this will be a parallel boxplot, and argument horizontal = TRUE can be used (see boxplot).

Value

These functions are invoked for their side effect of drawing in the active graphics device.

See Also

plot.default, points, lines, plot.factor.

Examples

```
op <- par(mfrow=c(2,1))
plot(Ozone ~ Wind, data = airquality, pch=as.character(Month))
plot(Ozone ~ Wind, data = airquality, pch=as.character(Month),
     subset = Month != 7)
par(op)
```

plot.histogram *Plot Histograms*

Description

These are methods for objects of class "histogram", typically produced by hist.

Usage

```
## S3 method for class 'histogram':
plot(x, freq = equidist, density = NULL, angle = 45,
               col = NULL, border = par("fg"), lty = NULL,
               main = paste("Histogram of",
                            paste(x$xname, collapse="\n")),
               sub = NULL, xlab = x$xname, ylab,
               xlim = range(x$breaks), ylim = NULL,
               axes = TRUE, labels = FALSE, add = FALSE, ann = TRUE, ...)

## S3 method for class 'histogram':
lines(x, ...)
```

Arguments

x	a histogram object, or a list with components density, mid, etc, see hist for information about the components of x.
freq	logical; if TRUE, the histogram graphic is to present a representation of frequencies, i.e, x$counts; if FALSE, *relative* frequencies (probabilities), i.e., x$density, are plotted. The default is true for equidistant breaks and false otherwise.
col	a colour to be used to fill the bars. The default of NULL yields unfilled bars.
border	the color of the border around the bars.
angle, density	select shading of bars by lines: see rect.
lty	the line type used for the bars, see also lines.
main, sub, xlab, ylab	these arguments to title have useful defaults here.
xlim, ylim	the range of x and y values with sensible defaults.
axes	logical, indicating if axes should be drawn.
labels	logical or character. Additionally draw labels on top of bars, if not FALSE; if TRUE, draw the counts or rounded densities; if labels is a character, draw itself.
add	logical. If TRUE, only the bars are added to the current plot. This is what lines.histogram(*) does.
ann	logical. Should annotations (titles and axis titles) be plotted?
...	further graphical parameters to title and axis.

Details

lines.histogram(*) is the same as plot.histogram(*, add = TRUE).

See Also

hist, stem, density.

Examples

```
(wwt <- hist(women$weight, nclass = 7, plot = FALSE))
plot(wwt, labels = TRUE) # default main & xlab using wwt$xname
plot(wwt, border = "dark blue", col = "light blue",
     main = "Histogram of 15 women's weights", xlab = "weight [pounds]")

## Fake "lines" example, using non-default labels:
w2 <- wwt; w2$counts <- w2$counts - 1
lines(w2, col = "Midnight Blue", labels = ifelse(w2$counts, "> 1", "1"))
```

---

plot.table *Plot Methods for 'table' Objects*

Description

This is a method of the generic `plot` function for (contingency) `table` objects. Whereas for two- and more dimensional tables, a `mosaicplot` is drawn, one-dimensional ones are plotted as bars.

Usage

```
## S3 method for class 'table':
plot(x, type = "h", ylim = c(0, max(x)), lwd = 2,
     xlab = NULL, ylab = NULL, frame.plot = is.num, ...)
```

Arguments

`x`	a `table` (like) object.
`type`	plotting type.
`ylim`	range of y-axis.
`lwd`	line width for bars when `type = "h"` is used in the 1D case.
`xlab, ylab`	x- and y-axis labels.
`frame.plot`	logical indicating if a frame (`box`) should be drawn in the 1D case. Defaults to true when `x` has `dimnames` coerce-able to numbers.
`...`	further graphical arguments, see `plot.default`.

See Also

`plot.factor`, the `plot` method for factors.

Examples

```
## 1-d tables
(Poiss.tab <- table(N = stats::rpois(200, lambda = 5)))
plot(Poiss.tab, main = "plot(table(rpois(200, lambda = 5)))")

plot(table(state.division))

## 4-D :
plot(Titanic, main ="plot(Titanic, main= *)")
```

---

plot.window *Set up World Coordinates for Graphics Window*

Description

This function sets up the world coordinate system for a graphics window. It is called by higher level functions such as `plot.default` (*after* `plot.new`).

Usage

```
plot.window(xlim, ylim, log = "", asp = NA, ...)
```

Arguments

`xlim, ylim`	numeric vectors of length 2, giving the x and y coordinates ranges.
`log`	character; indicating which axes should be in log scale.
`asp`	numeric, giving the **asp**ect ratio y/x, see below.
`...`	further graphical parameters as in `par`. The relevant ones are `xaxs`, `yaxs` and `lab`.

Details

asp: If `asp` is a finite positive value then the window is set up so that one data unit in the x direction is equal in length to `asp` $\times$ one data unit in the y direction.

Note that in this case, `par("usr")` is no longer determined by, e.g., `par("xaxs")`, but rather by `asp` and the device's aspect ratio. (See what happens if you interactively resize the plot device after running the example below!)

The special case `asp == 1` produces plots where distances between points are represented accurately on screen. Values with `asp > 1` can be used to produce more accurate maps when using latitude and longitude.

To reverse an axis, use `xlim` or `ylim` of the form `c(hi, lo)`.

The function attempts to produce a plausible set of scales if one or both of `xlim` and `ylim` is of length one or the two values given are identical, but it is better to avoid that case.

Usually, one should rather use the higher level functions such as `plot`, `hist`, `image`, ..., instead and refer to their help pages for explanation of the arguments.

A side-effect of the call is to set up the `usr`, `xaxp` and `yaxp` graphical parameters. (It is for the latter two that `lab` is used.)

See Also

`xy.coords`, `plot.xy`, `plot.default`.

Examples

```
##--- An example for the use of 'asp' :
require(stats)  # normally loaded
loc <- cmdscale(eurodist)
rx <- range(x <- loc[,1])
ry <- range(y <- -loc[,2])
plot(x, y, type="n", asp=1, xlab="", ylab="")
abline(h = pretty(rx, 10), v = pretty(ry, 10), col = "lightgray")
text(x, y, labels(eurodist), cex=0.8)
```

`plot.xy` *Basic Internal Plot Function*

Description

This is *the* internal function that does the basic plotting of points and lines. Usually, one should rather use the higher level functions instead and refer to their help pages for explanation of the arguments.

Usage

```
plot.xy(xy, type, pch = par("pch"), lty = par("lty"),
        col = par("col"), bg = NA,
        cex = 1, lwd = par("lwd"), ...)
```

Arguments

`xy`	A four-element list as results from `xy.coords`.
`type`	1 character code: see `plot.default`. `NULL` is accepted as a synonym for `"p"`.
`pch`	character or integer code for kind of points, see `points.default`.
`lty`	line type code, see `lines`.
`col`	color code or name, see `colors`, `palette`. Here `NULL` means colour 0.
`bg`	background (fill) color for the open plot symbols 21:25: see `points.default`.
`cex`	character expansion.
`lwd`	line width, also used for (non-filled) plot symbols, see `lines` and `points`.
`...`	further graphical parameters such as `xpd`, `lend`, `ljoin` and `lmitre`.

Details

The arguments `pch`, `col`, `bg`, `cex`, `lwd` may be vectors and may be recycled, depending on `type`: see `points` and `lines` for specifics. In particular note that `lwd` is treated as a vector for points and as a single (first) value for lines.

`cex` is a numeric factor in addition to `par("cex")` which affects symbols and characters as drawn by `type` `"p"`, `"o"`, `"b"` and `"c"`.

See Also

plot, plot.default, points, lines.

Examples

```
points.default # to see how it calls "plot.xy(xy.coords(x, y), ...)"
```

---

points *Add Points to a Plot*

Description

points is a generic function to draw a sequence of points at the specified coordinates. The specified character(s) are plotted, centered at the coordinates.

Usage

```
points(x, ...)

## Default S3 method:
points(x, y = NULL, type = "p", ...)
```

Arguments

x, y	coordinate vectors of points to plot.
type	character indicating the type of plotting; actually any of the types as in plot.default.
...	Further graphical parameters may also be supplied as arguments. See 'Details'.

Details

The coordinates can be passed in a plotting structure (a list with x and y components), a two-column matrix, a time series, ... See xy.coords. If supplied separately, they must be of the same length.

Graphical parameters commonly used are

pch plotting 'character', i.e., symbol to use. This can either be a single character or an integer code for one of a set of graphics symbols. The full set of S symbols is available with pch=0:18, see the examples below.

Value pch="." (equivalently pch = 46) is handled specially. It is a rectangle of side 0.01 inch (scaled by cex). In addition, if cex = 1 (the default), each side is at least one pixel (1/72 inch on the pdf, postscript and xfig devices).

For other text symbols, cex = 1 corresponds to the default fontsize of the device, often specified by an argument pointsize. For pch in 0:25 the default size is about 75% of the character height (see par("cin")).

col color code or name, see par.

bg background (fill) color for the open plot symbols given by pch=21:25.

cex character (or symbol) expansion: a numerical vector. This works as a multiple of par("cex").

lwd line width for drawing symbols see par.

Others less commonly used are `lty` and `lwd` for types such as `"b"` and `"l"`.

Graphical parameters `pch`, `col`, `bg`, `cex` and `lwd` can be vectors (which will be recycled as needed) giving a value for each point plotted. If lines are to be plotted (e.g. for `type = "b"`/ the first element of `lwd` is used.

Points whose `x`, `y`, `pch`, `col` or `cex` value is `NA` are omitted from the plot.

'pch' values

Values of `pch` are stored internally as integers. The interpretation is

- `NA_integer_`: no symbol.
- `0:18`: S-compatible vector symbols.
- `0:25`: R vector symbols.
- `26:31`: unused (and ignored).
- `32:127`: ASCII characters.
- `128:255` native characters *only in a single-byte locale and for the symbol font*. (`128:159` are only used on Windows.)
- `-32 ...` Unicode point (where supported).

Note that unlike S (which uses octagons), symbols `1`, `10`, `13` and `16` use circles. The filled shapes do not include a border.

The following R plotting symbols are can be obtained with `pch=19:25`: those with `21:25` can be colored and filled with different colors: `col` gives the border color and `bg` the background color.

- `pch=19`: solid circle,
- `pch=20`: bullet (smaller circle),
- `pch=21`: filled circle,
- `pch=22`: filled square,
- `pch=23`: filled diamond,
- `pch=24`: filled triangle point-up,
- `pch=25`: filled triangle point down.

Note that all of these both fill the shape and draw a border. Some care in interpretation is needed when semi-transparent colours are used for both fill and border (and the result might be device-specific and even viewer-specific for `pdf`).

Values `pch=26:31` are currently unused and `pch=32:127` give the ASCII characters. In a single-byte locale `pch=128:255` give the corresponding character (if any) in the locale's character set. Where supported by the OS, negative values specify a Unicode point, so e.g. `-0x2642L` is a 'male sign' and `-0x20ACL` is the Euro.

A character string consisting of a single character is converted to an integer: `32:127` for ASCII characters, and usually to the Unicode point number otherwise. (In non-Latin-1 single-byte locales, `128:255` will be used for 8-bit characters.)

If `pch` supplied is a logical, integer or character `NA` or an empty character string the point is omitted from the plot.

If `pch` is `NULL` or otherwise of length 0, `par("pch")` is used.

If the symbol font (`par(font = 5)`) is used, numerical values should be used for `pch`: the range is `c(32:126, 160:254)` in all locales (but `240` is not defined (used for 'apple' on Mac OS) and `160`, Euro, may not be present).

Note

A single-byte encoding may include the characters in `pch=128:255`, and if it does, a font may not include all (or even any) of them.

Not all negative numbers are valid as Unicode points, and no check is done. A display device is likely to use a rectangle for (or omit) Unicode points that do not exist or which it does not have a glyph.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`plot`, `lines`, and the underlying workhorse function `plot.xy`.

Examples

```
require(stats) # for rnorm
plot(-4:4, -4:4, type = "n")# setting up coord. system
points(rnorm(200), rnorm(200), col = "red")
points(rnorm(100)/2, rnorm(100)/2, col = "blue", cex = 1.5)

op <- par(bg = "light blue")
x <- seq(0,2*pi, len=51)
## something "between type='b' and type='o'":
plot(x, sin(x), type="o", pch=21, bg=par("bg"), col = "blue", cex=.6,
 main='plot(..., type="o", pch=21, bg=par("bg"))')
par(op)

##-------- Showing all the extra & some char graphics symbols ---------
pchShow <-
  function(extras = c("*",".", "o","O","0","+","-","|","%","#"),
           cex = 3, ## good for both .Device=="postscript" and "x11"
           col = "red3", bg = "gold", coltext = "brown", cextext = 1.2,
           main = paste("plot symbols :  points (...  pch = *, cex =",
                        cex,")"))
  {
    nex <- length(extras)
    np  <- 26 + nex
    ipch <- 0:(np-1)
    k <- floor(sqrt(np))
    dd <- c(-1,1)/2
    rx <- dd + range(ix <- ipch %/% k)
    ry <- dd + range(iy <- 3 + (k-1)- ipch %% k)
    pch <- as.list(ipch) # list with integers & strings
    if(nex > 0) pch[26+ 1:nex] <- as.list(extras)
    plot(rx, ry, type="n", axes = FALSE, xlab = "", ylab = "",
         main = main)
    abline(v = ix, h = iy, col = "lightgray", lty = "dotted")
    for(i in 1:np) {
      pc <- pch[[i]]
      ## 'col' symbols with a 'bg'-colored interior (where available) :
      points(ix[i], iy[i], pch = pc, col = col, bg = bg, cex = cex)
      if(cextext > 0)
```

```
            text(ix[i] - 0.3, iy[i], pc, col = coltext, cex = cextext)
        }
      }

pchShow()
pchShow(c("o","O","0"), cex = 2.5)
pchShow(NULL, cex = 4, cextext = 0, main = NULL)


## ------------ test code for various pch specifications -------------
# Try this in various font families (including Hershey)
# and locales.  Use sign=-1 asserts we want Latin-1.
# Standard cases in a MBCS locale will not plot the top half.
TestChars <- function(sign=1, font=1, ...)
{
   if(font == 5) { sign <- 1; r <- c(32:126, 160:254)
   } else if (l10n_info()$MBCS) r <- 32:126 else r <- 32:255
   if (sign == -1) r <- c(32:126, 160:255)
   par(pty="s")
   plot(c(-1,16), c(-1,16), type="n", xlab="", ylab="",
        xaxs="i", yaxs="i")
   grid(17, 17, lty=1)
   for(i in r) try(points(i%%16, i%/%16, pch=sign*i, font=font,...))
}
TestChars()
try(TestChars(sign=-1)) # needs MBCS support
TestChars(font=5) # Euro might be at 160.  Mac OS has apple at 240.
```

`polygon` *Polygon Drawing*

Description

`polygon` draws the polygons whose vertices are given in `x` and `y`.

Usage

```
polygon(x, y = NULL, density = NULL, angle = 45,
        border = NULL, col = NA, lty = par("lty"), ...)
```

Arguments

`x,y`	vectors containing the coordinates of the vertices of the polygon.
`density`	the density of shading lines, in lines per inch. The default value of `NULL` means that no shading lines are drawn. A zero value of `density` means no shading nor filling whereas negative values (and `NA`) suppress shading (and so allow color filling).
`angle`	the slope of shading lines, given as an angle in degrees (counter-clockwise).
`col`	the color for filling the polygon. The default, `NA`, is to leave polygons unfilled, unless `density` is specified. (For back-compatibility, `NULL` is equivalent to `NA`.) If `density` is specified with a positive value this gives the color of the shading lines.

`border`	the color to draw the border. The default, `NULL`, means to use `par("fg")`. Use `border = NA` to omit borders. For compatibility with S, `border` can also be logical, in which case `FALSE` is equivalent to `NA` (borders omitted) and `TRUE` is equivalent to `NULL` (use the foreground colour),
`lty`	the line type to be used, as in `par`.
`...`	graphical parameters such as `xpd`, `lend`, `ljoin` and `lmitre` can be given as arguments.

Details

The coordinates can be passed in a plotting structure (a list with `x` and `y` components), a two-column matrix, ... See `xy.coords`.

It is assumed that the polygon is to be closed by joining the last point to the first point.

The coordinates can contain missing values. The behaviour is similar to that of `lines`, except that instead of breaking a line into several lines, `NA` values break the polygon into several complete polygons (including closing the last point to the first point). See the examples below.

When multiple polygons are produced, the values of `density`, `angle`, `col`, `border`, and `lty` are recycled in the usual manner.

Bugs

The present shading algorithm can produce incorrect results for self-intersecting polygons.

Author(s)

The code implementing polygon shading was donated by Kevin Buhr ⟨buhr@stat.wisc.edu⟩.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`segments` for even more flexibility, `lines`, `rect`, `box`, `abline`.

`par` for how to specify colors.

Examples

```
x <- c(1:9,8:1)
y <- c(1,2*(5:3),2,-1,17,9,8,2:9)
op <- par(mfcol=c(3,1))
for(xpd in c(FALSE,TRUE,NA)) {
  plot(1:10, main = paste("xpd =", xpd))
  box("figure", col = "pink", lwd=3)
  polygon(x,y, xpd=xpd, col="orange", lty=2, lwd=2, border="red")
}
par(op)

n <- 100
```

```
xx <- c(0:n, n:0)
yy <- c(c(0,cumsum(stats::rnorm(n))), rev(c(0,cumsum(stats::rnorm(n)))))
plot   (xx, yy, type="n", xlab="Time", ylab="Distance")
polygon(xx, yy, col="gray", border = "red")
title("Distance Between Brownian Motions")

# Multiple polygons from NA values
# and recycling of col, border, and lty
op <- par(mfrow=c(2,1))
plot(c(1,9), 1:2, type="n")
polygon(1:9, c(2,1,2,1,1,2,1,2,1),
        col=c("red", "blue"),
        border=c("green", "yellow"),
        lwd=3, lty=c("dashed", "solid"))
plot(c(1,9), 1:2, type="n")
polygon(1:9, c(2,1,2,1,NA,2,1,2,1),
        col=c("red", "blue"),
        border=c("green", "yellow"),
        lwd=3, lty=c("dashed", "solid"))
par(op)

# Line-shaded polygons
plot(c(1,9), 1:2, type="n")
polygon(1:9, c(2,1,2,1,NA,2,1,2,1),
        density=c(10, 20), angle=c(-45, 45))
```

`rect` *Draw One or More Rectangles*

Description

`rect` draws a rectangle (or sequence of rectangles) with the given coordinates, fill and border colors.

Usage

```
rect(xleft, ybottom, xright, ytop, density = NULL, angle = 45,
     col = NA, border = NULL, lty = par("lty"), lwd = par("lwd"),
     ...)
```

Arguments

`xleft`	a vector (or scalar) of left x positions.
`ybottom`	a vector (or scalar) of bottom y positions.
`xright`	a vector (or scalar) of right x positions.
`ytop`	a vector (or scalar) of top y positions.
`density`	the density of shading lines, in lines per inch. The default value of `NULL` means that no shading lines are drawn. A zero value of `density` means no shading lines whereas negative values (and `NA`) suppress shading (and so allow color filling).
`angle`	angle (in degrees) of the shading lines.

col	color(s) to fill or shade the rectangle(s) with. The default NA (or also NULL) means do not fill, i.e., draw transparent rectangles, unless density is specified.
border	color for rectangle border(s). The default means par("fg"). Use border = NA to omit borders. If there are shading lines, border = TRUE means use the same colour for the border as for the shading lines.
lty	line type for borders and shading; defaults to "solid".
lwd	line width for borders and shading.
...	graphical parameters such as xpd, lend, ljoin and lmitre can be given as arguments.

Details

The positions supplied, i.e., xleft, ..., are relative to the current plotting region. If the x-axis goes from 100 to 200 then xleft must be larger than 100 and xright must be less than 200. The position vectors will be recycled to the length of the longest.

It is a graphics primitive used in hist, barplot, legend, etc.

See Also

box for the standard box around the plot; polygon and segments for flexible line drawing.

par for how to specify colors.

Examples

```
require(grDevices)
## set up the plot region:
op <- par(bg = "thistle")
plot(c(100, 250), c(300, 450), type = "n", xlab="", ylab="",
     main = "2 x 11 rectangles; 'rect(100+i,300+i,  150+i,380+i)'")
i <- 4*(0:10)
## draw rectangles with bottom left (100, 300)+i
## and top right (150, 380)+i
rect(100+i, 300+i, 150+i, 380+i, col=rainbow(11, start=.7,end=.1))
rect(240-i, 320+i, 250-i, 410+i, col=heat.colors(11), lwd=i/5)
## Background alternating  ( transparent / "bg" ) :
j <- 10*(0:5)
rect(125+j, 360+j,   141+j, 405+j/2, col = c(NA,0),
     border = "gold", lwd = 2)
rect(125+j, 296+j/2, 141+j, 331+j/5, col = c(NA,"midnightblue"))
mtext("+  2 x 6 rect(*, col = c(NA,0)) and  col = c(NA,\"m..blue\"))")

## an example showing colouring and shading
plot(c(100, 200), c(300, 450), type= "n", xlab="", ylab="")
rect(100, 300, 125, 350) # transparent
rect(100, 400, 125, 450, col="green", border="blue") # coloured
rect(115, 375, 150, 425, col=par("bg"), border="transparent")
rect(150, 300, 175, 350, density=10, border="red")
rect(150, 400, 175, 450, density=30, col="blue",
     angle=-30, border="transparent")

legend(180, 450, legend=1:4, fill=c(NA, "green", par("fg"), "blue"),
       density=c(NA, NA, 10, 30), angle=c(NA, NA, 30, -30))

par(op)
```

`rug` *Add a Rug to a Plot*

Description

Adds a *rug* representation (1-d plot) of the data to the plot.

Usage

```
rug(x, ticksize = 0.03, side = 1, lwd = 0.5, col = par("fg"),
    quiet = getOption("warn") < 0, ...)
```

Arguments

`x`	A numeric vector
`ticksize`	The length of the ticks making up the 'rug'. Positive lengths give inwards ticks.
`side`	On which side of the plot box the rug will be plotted. Normally 1 (bottom) or 3 (top).
`lwd`	The line width of the ticks. Some devices will round the default width up to `1`.
`col`	The colour the ticks are plotted in.
`quiet`	logical indicating if there should be a warning about clipped values.
`...`	further arguments, passed to `axis`, such as `line` or `pos` for specifying the location of the rug.

Details

Because of the way `rug` is implemented, only values of `x` that fall within the plot region are included. There will be a warning if any finite values are omitted, but non-finite values are omitted silently.

Prior to R 2.8.0 `rug` re-drew the axis like: it no longer does so.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`jitter` which you may want for ties in `x`.

Examples

```
require(stats)# both 'density' and its default method
with(faithful, {
    plot(density(eruptions, bw = 0.15))
    rug(eruptions)
    rug(jitter(eruptions, amount = 0.01), side = 3, col = "light blue")
})
```

`screen` *Creating and Controlling Multiple Screens on a Single Device*

Description

`split.screen` defines a number of regions within the current device which can, to some extent, be treated as separate graphics devices. It is useful for generating multiple plots on a single device. Screens can themselves be split, allowing for quite complex arrangements of plots.

`screen` is used to select which screen to draw in.

`erase.screen` is used to clear a single screen, which it does by filling with the background colour.

`close.screen` removes the specified screen definition(s).

Usage

```
split.screen(figs, screen, erase = TRUE)
screen(n = , new = TRUE)
erase.screen(n = )
close.screen(n, all.screens = FALSE)
```

Arguments

`figs`	A two-element vector describing the number of rows and the number of columns in a screen matrix *or* a matrix with 4 columns. If a matrix, then each row describes a screen with values for the left, right, bottom, and top of the screen (in that order) in NDC units, that is 0 at the lower left corner of the device surface, and 1 at the upper right corner.
`screen`	A number giving the screen to be split. It defaults to the current screen if there is one, otherwise the whole device region.
`erase`	logical: should be selected screen be cleared?
`n`	A number indicating which screen to prepare for drawing (`screen`), erase (`erase.screen`), or close (`close.screen`). (`close.screen` will accept a vector of screen numbers.)
`new`	A logical value indicating whether the screen should be erased as part of the preparation for drawing in the screen.
`all.screens`	A logical value indicating whether all of the screens should be closed.

Details

The first call to `split.screen` places R into split-screen mode. The other split-screen functions only work within this mode. While in this mode, certain other commands should be avoided (see the Warnings section below). Split-screen mode is exited by the command `close.screen(all = TRUE)`.

If the current screen is closed, `close.screen` sets the current screen to be the next larger screen number if there is one, otherwise to the first available screen.

Value

`split.screen` returns a vector of screen numbers for the newly-created screens. With no arguments, `split.screen` returns a vector of valid screen numbers.

`screen` invisibly returns the number of the selected screen. With no arguments, `screen` returns the number of the current screen.

`close.screen` returns a vector of valid screen numbers.

`screen`, `erase.screen`, and `close.screen` all return `FALSE` if R is not in split-screen mode.

Warnings

The recommended way to use these functions is to completely draw a plot and all additions (i.e. points and lines) to the base plot, prior to selecting and plotting on another screen. The behavior associated with returning to a screen to add to an existing plot is unpredictable and may result in problems that are not readily visible.

These functions are totally incompatible with the other mechanisms for arranging plots on a device: `par(mfrow)`, `par(mfcol)` and `layout()`.

The functions are also incompatible with some plotting functions, such as `coplot`, which make use of these other mechanisms.

`erase.screen` will appear not to work if the background colour is transparent (as it is by default on most devices).

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`par`, `layout`, `Devices`, `dev.*`

Examples

```
if (interactive()) {
par(bg = "white")           # default is likely to be transparent
split.screen(c(2,1))        # split display into two screens
split.screen(c(1,3), screen = 2) # now split the bottom half into 3
screen(1) # prepare screen 1 for output
plot(10:1)
screen(4) # prepare screen 4 for output
plot(10:1)
close.screen(all = TRUE)    # exit split-screen mode

split.screen(c(2,1))        # split display into two screens
split.screen(c(1,2),2)      # split bottom half in two
plot(1:10)                  # screen 3 is active, draw plot
erase.screen()              # forgot label, erase and redraw
plot(1:10, ylab= "ylab 3")
screen(1)                   # prepare screen 1 for output
plot(1:10)
screen(4)                   # prepare screen 4 for output
plot(1:10, ylab="ylab 4")
```

```
screen(1, FALSE)            # return to screen 1, but do not clear
plot(10:1, axes=FALSE, lty=2, ylab="")  # overlay second plot
axis(4)                     # add tic marks to right-hand axis
title("Plot 1")
close.screen(all = TRUE)    # exit split-screen mode
}
```

`segments` *Add Line Segments to a Plot*

Description

Draw line segments between pairs of points.

Usage

```
segments(x0, y0, x1, y1,
         col = par("fg"), lty = par("lty"), lwd = par("lwd"),
         ...)
```

Arguments

`x0, y0`	coordinates of points **from** which to draw.
`x1, y1`	coordinates of points **to** which to draw.
`col, lty, lwd`	usual graphical parameters as in `par`, possibly vectors. `NA` values in `col` cause the segment to be omitted.
`...`	further graphical parameters (from `par`), such as `xpd` and the line characteristics `lend`, `ljoin` and `lmitre`.

Details

For each `i`, a line segment is drawn between the point `(x0[i], y0[i])` and the point `(x1[i], y1[i])`. The coordinate vectors will be recycled to the length of the longest.

The graphical parameters `col`, `lty` and `lwd` can be vectors of length greater than one and will be recycled if necessary.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`arrows`, `polygon` for slightly easier and less flexible line drawing, and `lines` for the usual polygons.

Examples

```
x <- stats::runif(12); y <- stats::rnorm(12)
i <- order(x,y); x <- x[i]; y <- y[i]
plot(x, y, main="arrows(.) and segments(.)")
## draw arrows from point to point :
s <- seq(length(x)-1)# one shorter than data
arrows(x[s], y[s], x[s+1], y[s+1], col= 1:3)
s <- s[-length(s)]
segments(x[s], y[s], x[s+2], y[s+2], col= 'pink')
```

---

spineplot *Spine Plots and Spinograms*

Description

Spine plots are a special cases of mosaic plots, and can be seen as a generalization of stacked (or highlighted) bar plots. Analogously, spinograms are an extension of histograms.

Usage

```
spineplot(x, ...)

## Default S3 method:
spineplot(x, y = NULL,
          breaks = NULL, tol.ylab = 0.05, off = NULL,
          ylevels = NULL, col = NULL,
          main = "", xlab = NULL, ylab = NULL,
          xaxlabels = NULL, yaxlabels = NULL,
          xlim = NULL, ylim = c(0, 1), ...)

## S3 method for class 'formula':
spineplot(formula, data = list(),
          breaks = NULL, tol.ylab = 0.05, off = NULL,
          ylevels = NULL, col = NULL,
          main = "", xlab = NULL, ylab = NULL,
          xaxlabels = NULL, yaxlabels = NULL,
          xlim = NULL, ylim = c(0, 1), ...,
          subset = NULL)
```

Arguments

`x`	an object, the default method expects either a single variable (interpreted to be the explanatory variable) or a 2-way table. See details.
`y`	a `"factor"` interpreted to be the dependent variable
`formula`	a `"formula"` of type `y ~ x` with a single dependent `"factor"` and a single explanatory variable.
`data`	an optional data frame.
`breaks`	if the explanatory variable is numeric, this controls how it is discretized. `breaks` is passed to `hist` and can be a list of arguments.

`tol.ylab` convenience tolerance parameter for y-axis annotation. If the distance between two labels drops under this threshold, they are plotted equidistantly.

`off` vertical offset between the bars (in per cent). It is fixed to `0` for spinograms and defaults to `2` for spine plots.

`ylevels` a character or numeric vector specifying in which order the levels of the dependent variable should be plotted.

`col` a vector of fill colors of the same length as `levels(y)`. The default is to call `gray.colors`.

`main, xlab, ylab` character strings for annotation

`xaxlabels, yaxlabels` character vectors for annotation of x and y axis. Default to `levels(y)` and `levels(x)`, respectively for the spine plot. For `xaxlabels` in the spinogram, the breaks are used.

`xlim, ylim` the range of x and y values with sensible defaults.

`...` additional arguments passed to `rect`.

`subset` an optional vector specifying a subset of observations to be used for plotting.

Details

`spineplot` creates either a spinogram or a spine plot. It can be called via `spineplot(x, y)` or `spineplot(y ~ x)` where `y` is interpreted to be the dependent variable (and has to be categorical) and `x` the explanatory variable. `x` can be either categorical (then a spine plot is created) or numerical (then a spinogram is plotted). Additionally, `spineplot` can also be called with only a single argument which then has to be a 2-way table, interpreted to correspond to `table(x, y)`.

Both, spine plots and spinograms, are essentially mosaic plots with special formatting of spacing and shading. Conceptually, they plot $P(y|x)$ against $P(x)$. For the spine plot (where both x and y are categorical), both quantities are approximated by the corresponding empirical relative frequencies. For the spinogram (where x is numerical), x is first discretized (by calling `hist` with `breaks` argument) and then empirical relative frequencies are taken.

Thus, spine plots can also be seen as a generalization of stacked bar plots where not the heights but the widths of the bars corresponds to the relative frequencies of `x`. The heights of the bars then correspond to the conditional relative frequencies of `y` in every `x` group. Analogously, spinograms extend stacked histograms.

Value

The table visualized is returned invisibly.

Author(s)

Achim Zeileis ⟨Achim.Zeileis@R-project.org⟩

References

Friendly, M. (1994), Mosaic displays for multi-way contingency tables. *Journal of the American Statistical Association*, **89**, 190–200.

Hartigan, J.A., and Kleiner, B. (1984), A mosaic of television ratings. *The American Statistician*, **38**, 32–35.

Hofmann, H., Theus, M. (2005), *Interactive graphics for visualizing conditional distributions*, Unpublished Manuscript.

Hummel, J. (1996), Linked bar charts: Analysing categorical data graphically. *Computational Statistics*, **11**, 23–33.

See Also

mosaicplot, hist, cdplot

Examples

```
## treatment and improvement of patients with rheumatoid arthritis
treatment <- factor(rep(c(1, 2), c(43, 41)), levels = c(1, 2),
                    labels = c("placebo", "treated"))
improved <- factor(rep(c(1, 2, 3, 1, 2, 3), c(29, 7, 7, 13, 7, 21)),
                   levels = c(1, 2, 3),
                   labels = c("none", "some", "marked"))

## (dependence on a categorical variable)
(spineplot(improved ~ treatment))

## applications and admissions by department at UC Berkeley
## (two-way tables)
(spineplot(margin.table(UCBAdmissions, c(3, 2)),
           main = "Applications at UCB"))
(spineplot(margin.table(UCBAdmissions, c(3, 1)),
           main = "Admissions at UCB"))

## NASA space shuttle o-ring failures
fail <- factor(c(2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1,
                 1, 1, 1, 2, 1, 1, 1, 1, 1),
               levels = c(1, 2), labels = c("no", "yes"))
temperature <- c(53, 57, 58, 63, 66, 67, 67, 67, 68, 69, 70, 70,
                 70, 70, 72, 73, 75, 75, 76, 76, 78, 79, 81)

## (dependence on a numerical variable)
(spineplot(fail ~ temperature))
(spineplot(fail ~ temperature, breaks = 3))
(spineplot(fail ~ temperature, breaks = quantile(temperature)))

## highlighting for failures
spineplot(fail ~ temperature, ylevels = 2:1)
```

---

stars *Star (Spider/Radar) Plots and Segment Diagrams*

Description

Draw star plots or segment diagrams of a multivariate data set. With one single location, also draws 'spider' (or 'radar') plots.

Usage

```
stars(x, full = TRUE, scale = TRUE, radius = TRUE,
      labels = dimnames(x)[[1]], locations = NULL,
      nrow = NULL, ncol = NULL, len = 1,
      key.loc = NULL, key.labels = dimnames(x)[[2]],
      key.xpd = TRUE,
      xlim = NULL, ylim = NULL, flip.labels = NULL,
      draw.segments = FALSE,
      col.segments = 1:n.seg, col.stars = NA,
      axes = FALSE, frame.plot = axes,
      main = NULL, sub = NULL, xlab = "", ylab = "",
      cex = 0.8, lwd = 0.25, lty = par("lty"), xpd = FALSE,
      mar = pmin(par("mar"),
                 1.1+ c(2*axes+ (xlab != ""),
                 2*axes+ (ylab != ""), 1,0)),
      add = FALSE, plot = TRUE, ...)
```

Arguments

`x`	matrix or data frame of data. One star or segment plot will be produced for each row of `x`. Missing values (`NA`) are allowed, but they are treated as if they were 0 (after scaling, if relevant).
`full`	logical flag: if `TRUE`, the segment plots will occupy a full circle. Otherwise, they occupy the (upper) semicircle only.
`scale`	logical flag: if `TRUE`, the columns of the data matrix are scaled independently so that the maximum value in each column is 1 and the minimum is 0. If `FALSE`, the presumption is that the data have been scaled by some other algorithm to the range $[0, 1]$.
`radius`	logical flag: in `TRUE`, the radii corresponding to each variable in the data will be drawn.
`labels`	vector of character strings for labeling the plots. Unlike the S function `stars`, no attempt is made to construct labels if `labels = NULL`.
`locations`	Either two column matrix with the x and y coordinates used to place each of the segment plots; or numeric of length 2 when all plots should be superimposed (for a 'spider plot'). By default, `locations = NULL`, the segment plots will be placed in a rectangular grid.
`nrow, ncol`	integers giving the number of rows and columns to use when `locations` is `NULL`. By default, `nrow == ncol`, a square layout will be used.
`len`	scale factor for the length of radii or segments.
`key.loc`	vector with x and y coordinates of the unit key.
`key.labels`	vector of character strings for labeling the segments of the unit key. If omitted, the second component of `dimnames(x)` is used, if available.
`key.xpd`	clipping switch for the unit key (drawing and labeling), see `par("xpd")`.
`xlim`	vector with the range of x coordinates to plot.
`ylim`	vector with the range of y coordinates to plot.
`flip.labels`	logical indicating if the label locations should flip up and down from diagram to diagram. Defaults to a somewhat smart heuristic.

draw.segments	logical. If TRUE draw a segment diagram.
col.segments	color vector (integer or character, see par), each specifying a color for one of the segments (variables). Ignored if draw.segments = FALSE.
col.stars	color vector (integer or character, see par), each specifying a color for one of the stars (cases). Ignored if draw.segments = TRUE.
axes	logical flag: if TRUE axes are added to the plot.
frame.plot	logical flag: if TRUE, the plot region is framed.
main	a main title for the plot.
sub	a sub title for the plot.
xlab	a label for the x axis.
ylab	a label for the y axis.
cex	character expansion factor for the labels.
lwd	line width used for drawing.
lty	line type used for drawing.
xpd	logical or NA indicating if clipping should be done, see par(xpd = .).
mar	argument to par(mar = *), typically choosing smaller margins than by default.
...	further arguments, passed to the first call of plot(), see plot.default and to box() if frame.plot is true.
add	logical, if TRUE *add* stars to current plot.
plot	logical, if FALSE, nothing is plotted.

Details

Missing values are treated as 0.

Each star plot or segment diagram represents one row of the input x. Variables (columns) start on the right and wind counterclockwise around the circle. The size of the (scaled) column is shown by the distance from the center to the point on the star or the radius of the segment representing the variable.

Only one page of output is produced.

Note

This code started life as spatial star plots by David A. Andrews. See http://www.udallas.edu:8080/~andrews/software/software.html.

Prior to 1.4.1, scaling only shifted the maximum to 1, although documented as here.

Author(s)

Thomas S. Dye

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`symbols` for another way to draw stars and other symbols.

Examples

```
require(grDevices)
stars(mtcars[, 1:7], key.loc = c(14, 2),
      main = "Motor Trend Cars : stars(*, full = F)", full = FALSE)
stars(mtcars[, 1:7], key.loc = c(14, 1.5),
      main = "Motor Trend Cars : full stars()",flip.labels=FALSE)

## 'Spider' or 'Radar' plot:
stars(mtcars[, 1:7], locations = c(0,0), radius = FALSE,
      key.loc=c(0,0), main="Motor Trend Cars", lty = 2)

## Segment Diagrams:
palette(rainbow(12, s = 0.6, v = 0.75))
stars(mtcars[, 1:7], len = 0.8, key.loc = c(12, 1.5),
      main = "Motor Trend Cars", draw.segments = TRUE)
stars(mtcars[, 1:7], len = 0.6, key.loc = c(1.5, 0),
      main = "Motor Trend Cars", draw.segments = TRUE,
      frame.plot=TRUE, nrow = 4, cex = .7)

## scale linearly (not affinely) to [0, 1]
USJudge <- apply(USJudgeRatings, 2, function(x) x/max(x))
Jnam <- row.names(USJudgeRatings)
Snam <- abbreviate(substring(Jnam,1,regexpr("[,.]",Jnam) - 1), 7)
stars(USJudge, labels = Jnam, scale = FALSE,
      key.loc = c(13, 1.5), main = "Judge not ...", len = 0.8)
stars(USJudge, labels = Snam, scale = FALSE,
      key.loc = c(13, 1.5), radius = FALSE)

loc <- stars(USJudge, labels = NULL, scale = FALSE,
             radius = FALSE, frame.plot = TRUE,
             key.loc = c(13, 1.5), main = "Judge not ...", len = 1.2)
text(loc, Snam, col = "blue", cex = 0.8, xpd = TRUE)

## 'Segments':
stars(USJudge, draw.segments = TRUE, scale = FALSE, key.loc = c(13,1.5))

## 'Spider':
stars(USJudgeRatings, locations=c(0,0), scale=FALSE,radius = FALSE,
      col.stars=1:10, key.loc = c(0,0), main="US Judges rated")
## 'Radar-Segments'
stars(USJudgeRatings[1:10,], locations = 0:1, scale=FALSE,
      draw.segments = TRUE, col.segments=0, col.stars=1:10,key.loc= 0:1,
       main="US Judges 1-10 ")
palette("default")
stars(cbind(1:16,10*(16:1)),draw.segments=TRUE,
      main = "A Joke -- do *not* use symbols on 2D data!")
```

`stem` *Stem-and-Leaf Plots*

Description

`stem` produces a stem-and-leaf plot of the values in `x`. The parameter `scale` can be used to expand the scale of the plot. A value of `scale=2` will cause the plot to be roughly twice as long as the default.

Usage

```
stem(x, scale = 1, width = 80, atom = 1e-08)
```

Arguments

`x`	a numeric vector.
`scale`	This controls the plot length.
`width`	The desired width of plot.
`atom`	a tolerance.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Examples

```
stem(islands)
stem(log10(islands))
```

`stripchart` *1-D Scatter Plots*

Description

`stripchart` produces one dimensional scatter plots (or dot plots) of the given data. These plots are a good alternative to `boxplot`s when sample sizes are small.

Usage

```
stripchart(x, ...)

## S3 method for class 'formula':
stripchart(x, data = NULL, dlab = NULL, ...,
           subset, na.action = NULL)

## Default S3 method:
stripchart(x, method = "overplot", jitter = 0.1, offset = 1/3,
           vertical = FALSE, group.names, add = FALSE,
           at = NULL, xlim = NULL, ylim = NULL,
           ylab=NULL, xlab=NULL, dlab="", glab="",
           log = "", pch = 0, col = par("fg"), cex = par("cex"),
           axes = TRUE, frame.plot = axes, ...)
```

Arguments

x	the data from which the plots are to be produced. In the default method the data can be specified as a single numeric vector, or as list of numeric vectors, each corresponding to a component plot. In the formula method, a symbolic specification of the form y ~ g can be given, indicating the observations in the vector y are to be grouped according to the levels of the factor g. NAs are allowed in the data.
data	a data.frame (or list) from which the variables in x should be taken.
subset	an optional vector specifying a subset of observations to be used for plotting.
na.action	a function which indicates what should happen when the data contain NAs. The default is to ignore missing values in either the response or the group.
...	additional parameters passed to the default method, or by it to plot, axis and title to control the appearance of the plot.
method	the method to be used to separate coincident points. The default method "overplot" causes such points to be overplotted, but it is also possible to specify "jitter" to jitter the points, or "stack" have coincident points stacked. The last method only makes sense for very granular data.
jitter	when method="jitter" is used, jitter gives the amount of jittering applied.
offset	when stacking is used, points are stacked this many line-heights (symbol widths) apart.
vertical	when vertical is TRUE the plots are drawn vertically rather than the default horizontal.
group.names	group labels which will be printed alongside (or underneath) each plot.
add	logical, if true *add* the chart to the current plot.
at	numeric vector giving the locations where the charts should be drawn, particularly when add = TRUE; defaults to 1:n where n is the number of boxes.
ylab, xlab	labels: see title.
dlab, glab	alternate way to specify axis labels: see 'Details'.
xlim, ylim	plot limits: see plot.window.
log	on which axes to use a log scale: see plot.default
pch, col, cex	Graphical parameters: see par.
axes, frame.plot	Axis control: see plot.default

Details

Extensive examples of the use of this kind of plot can be found in Box, Hunter and Hunter or Seber and Wild.

The dlab and glab labels may be used instead of xlab and ylab if those are not specified. dlab applies to the continuous data axis (the X axis unless vertical is TRUE), glab to the group axis.

Examples

```
x <- stats::rnorm(50)
xr <- round(x, 1)
stripchart(x) ; m <- mean(par("usr")[1:2])
text(m, 1.04, "stripchart(x, \"overplot\")")
stripchart(xr, method = "stack", add = TRUE, at = 1.2)
text(m, 1.35, "stripchart(round(x,1), \"stack\")")
stripchart(xr, method = "jitter", add = TRUE, at = 0.7)
text(m, 0.85, "stripchart(round(x,1), \"jitter\")")

stripchart(decrease ~ treatment,
    main = "stripchart(OrchardSprays)",
    vertical = TRUE, log = "y", data = OrchardSprays)

stripchart(decrease ~ treatment, at = c(1:8)^2,
    main = "stripchart(OrchardSprays)",
    vertical = TRUE, log = "y", data = OrchardSprays)
```

---

strwidth *Plotting Dimensions of Character Strings and Math Expressions*

Description

These functions compute the width or height, respectively, of the given strings or mathematical expressions `s[i]` on the current plotting device in *user* coordinates, *inches* or as fraction of the figure width `par("fin")`.

Usage

```
strwidth(s, units = "user", cex = NULL, font = NULL, vfont = NULL, ...)
strheight(s, units = "user", cex = NULL, font = NULL, vfont = NULL, ...)
```

Arguments

`s`	a character or expression vector whose dimensions are to be determined. Other objects are coerced by `as.graphicsAnnot`.
`units`	character indicating in which units `s` is measured; should be one of `"user"`, `"inches"`, `"figure"`; partial matching is performed.
`cex`	numeric **c**haracter **ex**pansion factor; multiplied by `par("cex")` yields the final character size; the default `NULL` is equivalent to `1`.
`font, vfont, ...`	additional information about the font, possibly including the graphics parameter `"family"`: see `text`.

Details

Where an element of `s` is a multi-line string (that is, contains newlines `\n`), the width and height are of an enclosing rectangle of the string as plotted by `text`. The inter-line spacing is controlled by `cex`, `par("lheight")` and the 'point size' (but not the actual font in use).

Measurements in `"user"` units (the default) are only available after `plot.new` has been called – otherwise an error is thrown.

Value

Numeric vector with the same length as `s`, giving the width or height for each `s[i]`. `NA` strings are given width and height 0 (as they are not plotted).

See Also

`text`, `nchar`

Examples

```
str.ex <- c("W","w","I",".","WwI.")
op <- par(pty='s'); plot(1:100,1:100, type="n")
sw <- strwidth(str.ex); sw
all.equal(sum(sw[1:4]), sw[5])
#- since the last string contains the others

sw.i <- strwidth(str.ex, "inches"); 25.4 * sw.i # width in [mm]
unique(sw / sw.i)
# constant factor: 1 value
mean(sw.i / strwidth(str.ex, "fig")) / par('fin')[1] # = 1: are the same

## See how letters fall in classes
##  -- depending on graphics device and font!
all.lett <- c(letters, LETTERS)
shL <- strheight(all.lett, units = "inches") * 72 # 'big points'
table(shL) # all have same heights ...
mean(shL)/par("cin")[2] # around 0.6

(swL <- strwidth(all.lett, units="inches") * 72) # 'big points'
split(all.lett, factor(round(swL, 2)))

sumex <- expression(sum(x[i], i=1,n), e^{i * pi} == -1)
strwidth(sumex)
strheight(sumex)

par(op)#- reset to previous setting
```

---

sunflowerplot *Produce a Sunflower Scatter Plot*

Description

Multiple points are plotted as 'sunflowers' with multiple leaves ('petals') such that overplotting is visualized instead of accidental and invisible.

Usage

```
sunflowerplot(x, y = NULL, number, log = "", digits = 6,
              xlab = NULL, ylab = NULL, xlim = NULL, ylim = NULL,
              add = FALSE, rotate = FALSE,
              pch = 16, cex = 0.8, cex.fact = 1.5,
              col = par("col"), bg = NA, size = 1/8, seg.col = 2,
              seg.lwd = 1.5, ...)
```

Arguments

`x`	numeric vector of `x`-coordinates of length `n`, say, or another valid plotting structure, as for `plot.default`, see also `xy.coords`.
`y`	numeric vector of `y`-coordinates of length `n`.
`number`	integer vector of length `n`. `number[i]` = number of replicates for `(x[i],y[i])`, may be 0. Default (`missing(number)`): compute the exact multiplicity of the points `x[],y[]`, via `xyTable()`.
`log`	character indicating log coordinate scale, see `plot.default`.
`digits`	when `number` is computed (i.e., not specified), `x` and `y` are rounded to `digits` significant digits before multiplicities are computed.
`xlab,ylab`	character label for x-, or y-axis, respectively.
`xlim,ylim`	`numeric(2)` limiting the extents of the x-, or y-axis.
`add`	logical; should the plot be added on a previous one ? Default is `FALSE`.
`rotate`	logical; if `TRUE`, randomly rotate the sunflowers (preventing artefacts).
`pch`	plotting character to be used for points (`number[i]==1`) and center of sunflowers.
`cex`	numeric; character size expansion of center points (s. `pch`).
`cex.fact`	numeric *shrinking* factor to be used for the center points *when there are flower leaves*, i.e., `cex / cex.fact` is used for these.
`col, bg`	colors for the plot symbols, passed to `plot.default`.
`size`	of sunflower leaves in inches, 1[in] := 2.54[cm]. Default: 1/8, approximately 3.2mm.
`seg.col`	color to be used for the **seg**ments which make the sunflowers leaves, see `par(col=)`; `col = "gold"` reminds of real sunflowers.
`seg.lwd`	numeric; the line width for the leaves' segments.
`...`	further arguments to `plot` [if `add=FALSE`].

Details

For `number[i]==1`, a (slightly enlarged) usual plotting symbol (`pch`) is drawn. For `number[i] > 1`, a small plotting symbol is drawn and `number[i]` equi-angular 'rays' emanate from it.

If `rotate=TRUE` and `number[i] >= 2`, a random direction is chosen (instead of the y-axis) for the first ray. The goal is to `jitter` the orientations of the sunflowers in order to prevent artefactual visual impressions.

Value

A list with three components of same length,

`x`	x coordinates
`y`	y coordinates
`number`	number

Use `xyTable()` (from package **grDevices**) if you are only interested in this return value.

Side Effects

A scatter plot is drawn with 'sunflowers' as symbols.

Author(s)

Andreas Ruckstuhl, Werner Stahel, Martin Maechler, Tim Hesterberg, 1989–1993. Port to R by Martin Maechler ⟨maechler@stat.math.ethz.ch⟩.

References

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis.* Wadsworth.

Schilling, M. F. and Watkins, A. E. (1994) A suggestion for sunflower plots. *The American Statistician*, **48**, 303–305.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`density`, `xyTable`

Examples

```
require(stats)
require(grDevices)

## 'number' is computed automatically:
sunflowerplot(iris[, 3:4])
## Imitating  Chambers et al., p.109, closely:
sunflowerplot(iris[, 3:4], cex=.2, cex.fact=1, size=.035, seg.lwd=.8)

sunflowerplot(x=sort(2*round(rnorm(100))), y= round(rnorm(100),0),
              main = "Sunflower Plot of Rounded N(0,1)")
## Similarly using a "xyTable" argument:
xyT <- xyTable(x=sort(2*round(rnorm(100))), y= round(rnorm(100),0),
               digits=3)
utils::str(xyT, vec.len=20)
sunflowerplot(xyT, main = "2nd Sunflower Plot of Rounded N(0,1)")

## A 'marked point process' {explicit 'number' argument}:
sunflowerplot(rnorm(100), rnorm(100), number = rpois(n=100,lambda=2),
              main="Sunflower plot (marked point process)",
              rotate=TRUE, col = "blue4")
```

---

symbols — *Draw Symbols (Circles, Squares, Stars, Thermometers, Boxplots) on a Plot*

Description

This function draws symbols on a plot. One of six symbols; *circles*, *squares*, *rectangles*, *stars*, *thermometers*, and *boxplots*, can be plotted at a specified set of x and y coordinates. Specific aspects of the symbols, such as relative size, can be customized by additional parameters.

Usage

```
symbols(x, y = NULL, circles, squares, rectangles, stars,
        thermometers, boxplots, inches = TRUE, add = FALSE,
        fg = par("col"), bg = NA,
        xlab = NULL, ylab = NULL, main = NULL,
        xlim = NULL, ylim = NULL, ...)
```

Arguments

`x, y`	the x and y co-ordinates for the centres of the symbols. They can be specified in any way which is accepted by `xy.coords`.
`circles`	a vector giving the radii of the circles.
`squares`	a vector giving the length of the sides of the squares.
`rectangles`	a matrix with two columns. The first column gives widths and the second the heights of rectangles.
`stars`	a matrix with three or more columns giving the lengths of the rays from the center of the stars. `NA` values are replaced by zeroes.
`thermometers`	a matrix with three or four columns. The first two columns give the width and height of the thermometer symbols. If there are three columns, the third is taken as a proportion: the thermometers are filled (using colour `fg`) from their base to this proportion of their height. If there are four columns, the third and fourth columns are taken as proportions and the thermometers are filled between these two proportions of their heights. The part of the box not filled in `fg` will be filled in the background colour (default transparent) given by `bg`.
`boxplots`	a matrix with five columns. The first two columns give the width and height of the boxes, the next two columns give the lengths of the lower and upper whiskers and the fifth the proportion (with a warning if not in [0,1]) of the way up the box that the median line is drawn.
`inches`	`TRUE`, `FALSE` or a positive number. See 'Details'.
`add`	if `add` is `TRUE`, the symbols are added to an existing plot, otherwise a new plot is created.
`fg`	colour(s) the symbols are to be drawn in.
`bg`	if specified, the symbols are filled with colour(s), the vector `bg` being recycled to the number of symbols. The default is to leave the symbols unfilled.
`xlab`	the x label of the plot if `add` is not true. Defaults to the `deparse`d expression used for `x`.
`ylab`	the y label of the plot. Unused if `add = TRUE`.
`main`	a main title for the plot. Unused if `add = TRUE`.
`xlim`	numeric vector of length 2 giving the x limits for the plot. Unused if `add = TRUE`.
`ylim`	numeric vector of length 2 giving the y limits for the plot. Unused if `add = TRUE`.
`...`	graphics parameters can also be passed to this function, as can the plot aspect ratio `asp` (see `plot.window`).

Details

Observations which have missing coordinates or missing size parameters are not plotted. The exception to this is *stars*. In that case, the length of any ray which is `NA` is reset to zero.

Argument `inches` controls the sizes of the symbols. If `TRUE` (the default), the symbols are scaled so that the largest dimension of any symbol is one inch. If a positive number is given the symbols are scaled to make largest dimension this size in inches (so `TRUE` and `1` are equivalent). If `inches` is `FALSE`, the units are taken to be those of the appropriate axes. (For circles, squares and stars the units of the x axis are used. For boxplots, the lengths of the whiskers are regarded as dimensions alongside width and height when scaling by `inches`, and are otherwise interpreted in the units of the y axis.)

Circles of radius zero are plotted at radius one pixel (which is device-dependent).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

W. S. Cleveland (1985) *The Elements of Graphing Data.* Monterey, California: Wadsworth.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

`stars` for drawing *stars* with a bit more flexibility.

If you are thinking about doing 'bubble plots' by `symbols(*, circles=*)`, you should *really* consider using `sunflowerplot` instead.

Examples

```
require(stats); require(grDevices)
x <- 1:10
y <- sort(10*runif(10))
z <- runif(10)
z3 <- cbind(z, 2*runif(10), runif(10))
symbols(x, y, thermometers=cbind(.5, 1, z), inches=.5, fg = 1:10)
symbols(x, y, thermometers = z3, inches=FALSE)
text(x,y, apply(format(round(z3, digits=2)), 1, paste, collapse = ","),
     adj = c(-.2,0), cex = .75, col = "purple", xpd=NA)

## Note that  example(trees)  shows more sensible plots!
N <- nrow(trees)
with(trees, {
## Girth is diameter in inches
symbols(Height, Volume, circles=Girth/24, inches=FALSE,
        main="Trees' Girth")# xlab and ylab automatically
## Colours too:
palette(rainbow(N, end = 0.9))
symbols(Height, Volume, circles=Girth/16, inches=FALSE, bg = 1:N,
        fg="gray30", main="symbols(*, circles=Girth/16, bg = 1:N)")
palette("default")
})
```

`text` *Add Text to a Plot*

Description

`text` draws the strings given in the vector `labels` at the coordinates given by `x` and `y`. `y` may be missing since `xy.coords(x,y)` is used for construction of the coordinates.

Usage

```
text(x, ...)

## Default S3 method:
text (x, y = NULL, labels = seq_along(x), adj = NULL,
      pos = NULL, offset = 0.5, vfont = NULL,
      cex = 1, col = NULL, font = NULL, ...)
```

Arguments

`x, y`	numeric vectors of coordinates where the text `labels` should be written. If the length of `x` and `y` differs, the shorter one is recycled.
`labels`	a character vector or expression specifying the *text* to be written. An attempt is made to coerce other language objects (names and calls) to expressions, and vectors and other classed objects to character vectors by `as.character`. If `labels` is longer than `x` and `y`, the coordinates are recycled to the length of `labels`.
`adj`	one or two values in $[0, 1]$ which specify the x (and optionally y) adjustment of the labels. On most devices values outside that interval will also work.
`pos`	a position specifier for the text. If specified this overrides any `adj` value given. Values of `1`, `2`, `3` and `4`, respectively indicate positions below, to the left of, above and to the right of the specified coordinates.
`offset`	when `pos` is specified, this value gives the offset of the label from the specified coordinate in fractions of a character width.
`vfont`	`NULL` for the current font family, or a character vector of length 2 for Hershey vector fonts. The first element of the vector selects a typeface and the second element selects a style. Ignored if `labels` is an expression.
`cex`	numeric **c**haracter **ex**pansion factor; multiplied by `par("cex")` yields the final character size. `NULL` and `NA` are equivalent to `1.0`.
`col, font`	the color and (if `vfont = NULL`) font to be used, possibly vectors. These default to the values of the global graphical parameters in `par()`.
`...`	further graphical parameters (from `par`), such as `srt`, `family` and `xpd`.

Details

`labels` must be of type `character` or `expression` (or be coercible to such a type). In the latter case, quite a bit of mathematical notation is available such as sub- and superscripts, greek letters, fractions, etc.

`adj` allows *adj*ustment of the text with respect to `(x,y)`. Values of 0, 0.5, and 1 specify left/bottom, middle and right/top alignment, respectively. The default is for centered text, i.e., `adj`

= c(0.5, 0.5). Accurate vertical centering needs character metric information on individual characters which is only available on some devices. Vertical alignment is done slightly differently for character strings and for expressions: adj=c(0,0) means to left-justify and to align on the baseline for strings but on the bottom of the bounding box for expressions. This also affects vertical centering: for strings the centering excludes any descenders whereas for expressions it includes them.

The pos and offset arguments can be used in conjunction with values returned by identify to recreate an interactively labelled plot.

Text can be rotated by using graphical parameters srt (see par); this rotates about the centre set by adj.

Graphical parameters col, cex and font can be vectors and will then be applied cyclically to the labels (and extra values will be ignored). NA values of font are replaced by par("font").

Labels whose x, y, labels, cex or col value is NA are omitted from the plot.

What happens when font = 5 (the symbol font) is selected can be both device- and locale-dependent. Most often labels will be interpreted in the Adobe symbol encoding, so e.g. "d" is delta, and "\300" is aleph.

Euro symbol

The Euro symbol was introduced relatively recently and may not be available in older fonts. In recent versions of Adobe symbol fonts it is character 160, so text(x, y, "\xA0", font = 5) may work. People using Western European locales on Unix-alikes can probably select ISO-8895-15 (Latin-9) which has the Euro as character 165: this can also be used for postscript and pdf. It is \u20ac in Unicode, which can be used in UTF-8 locales.

The Euro should be rendered correctly by X11 in UTF-8 locales, but the corresponding single-byte encoding in postscript and pdf will need to be selected as ISOLatin9.enc.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

See Also

mtext, title, Hershey for details on Hershey vector fonts, plotmath for details and more examples on mathematical annotation.

Examples

```
plot(-1:1,-1:1, type = "n", xlab = "Re", ylab = "Im")
K <- 16; text(exp(1i * 2 * pi * (1:K) / K), col = 2)

## The following two examples use latin1 characters: these may not
## appear correctly (or be omitted entirely).
plot(1:10, 1:10, main = "text(...) examples\n~~~~~~~~~~~~~~~",
     sub = "R is GNU ©, but not ® ...")
mtext("«Latin-1 accented chars»: éè øØ å<Å æ<Æ", side=3)
points(c(6,2), c(2,1), pch = 3, cex = 4, col = "red")
text(6, 2, "the text is CENTERED around (x,y) = (6,2) by default",
     cex = .8)
text(2, 1, "or Left/Bottom - JUSTIFIED at (2,1) by 'adj = c(0,0)'",
```

```
     adj = c(0,0))
text(4, 9, expression(hat(beta) == (X^t * X)^{-1} * X^t * y))
text(4, 8.4, "expression(hat(beta) == (X^t * X)^{-1} * X^t * y)",
     cex = .75)
text(4, 7, expression(bar(x) == sum(frac(x[i], n), i==1, n)))

## Two more latin1 examples
text(5,10.2,
     "Le français, c'est façile: Règles, Liberté, Egalité, Fraternité...")
text(5,9.8,
     "Jetz no chli züritüütsch: (noch ein bißchen Zürcher deutsch)")
```

`title` *Plot Annotation*

Description

This function can be used to add labels to a plot. Its first four principal arguments can also be used as arguments in most high-level plotting functions. They must be of type `character` or `expression`. In the latter case, quite a bit of mathematical notation is available such as sub- and superscripts, greek letters, fractions, etc: see plotmath

Usage

```
title(main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
      line = NA, outer = FALSE, ...)
```

Arguments

`main`	The main title (on top) using font and size (character expansion) `par("font.main")` and color `par("col.main")`.
`sub`	Sub-title (at bottom) using font and size `par("font.sub")` and color `par("col.sub")`.
`xlab`	X axis label using font and character expansion `par("font.lab")` and color `par("col.lab")`.
`ylab`	Y axis label, same font attributes as `xlab`.
`line`	specifying a value for `line` overrides the default placement of labels, and places them this many lines outwards from the plot edge.
`outer`	a logical value. If `TRUE`, the titles are placed in the outer margins of the plot.
`...`	further graphical parameters from `par`. Use e.g., `col.main` or `cex.sub` instead of just `col` or `cex`. `adj` controls the justification of the titles. `xpd` can be used to set the clipping region: this defaults to the figure region unless `outer = TRUE`, otherwise the device region and can only be increased. `mgp` controls the default placing of the axis titles.

Details

The labels passed to `title` can be character strings or language objects (names, calls or expressions), or a list containing the string to be plotted, and a selection of the optional modifying graphical parameters `cex=`, `col=` and `font=`. Other objects will be coerced by `as.graphicsAnnot`.

The position of `main` defaults to being vertically centered in (outer) margin 3 and justified horizontally according to `par("adj")` on the plot region (device region for `outer=TRUE`).

The positions of `xlab`, `ylab` and `sub` are `line` (default for `xlab` and `ylab` being `par("mgp")[1]` and increased by `1` for `sub`) lines (of height `par("mex")`) into the appropriate margin, justified in the text direction according to `par("adj")` on the plot/device region.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`mtext`, `text`; `plotmath` for details on mathematical annotation.

Examples

```
plot(cars, main = "") # here, could use main directly
title(main = "Stopping Distance versus Speed")

plot(cars, main = "")
title(main = list("Stopping Distance versus Speed", cex=1.5,
                  col="red", font=3))

## Specifying "..." :
plot(1, col.axis = "sky blue", col.lab = "thistle")
title("Main Title", sub = "sub title",
      cex.main = 2,   font.main= 4, col.main= "blue",
      cex.sub = 0.75, font.sub = 3, col.sub = "red")

x <- seq(-4, 4, len = 101)
y <- cbind(sin(x), cos(x))
matplot(x, y, type = "l", xaxt = "n",
        main = expression(paste(plain(sin) * phi, "  and  ",
                                plain(cos) * phi)),
        ylab = expression("sin" * phi, "cos" * phi), # only 1st is taken
        xlab = expression(paste("Phase Angle ", phi)),
        col.main = "blue")
axis(1, at = c(-pi, -pi/2, 0, pi/2, pi),
     labels = expression(-pi, -pi/2, 0, pi/2, pi))
abline(h = 0, v = pi/2 * c(-1,1), lty = 2, lwd = .1, col = "gray70")
```

`units` *Graphical Units*

Description

`xinch` and `yinch` convert the specified number of inches given as their arguments into the correct units for plotting with graphics functions. Usually, this only makes sense when normal coordinates are used, i.e., *no* `log` scale (see the `log` argument to `par`).

`xyinch` does the same for a pair of numbers `xy`, simultaneously.

Usage

```
xinch(x = 1, warn.log = TRUE)
yinch(y = 1, warn.log = TRUE)
xyinch(xy = 1, warn.log = TRUE)
```

Arguments

`x,y`	numeric vector
`xy`	numeric of length 1 or 2.
`warn.log`	logical; if `TRUE`, a warning is printed in case of active log scale.

Examples

```
all(c(xinch(),yinch()) == xyinch()) # TRUE
xyinch()
xyinch #- to see that is really   delta{"usr"} / "pin"

## plot labels offset 0.12 inches to the right
## of plotted symbols in a plot
with(mtcars, {
    plot(mpg, disp, pch=19, main= "Motor Trend Cars")
    text(mpg + xinch(0.12), disp, row.names(mtcars),
         adj = 0, cex = .7, col = 'blue')
    })
```

`xspline` *Draw an X-spline*

Description

Draw an X-spline, a curve drawn relative to control points.

Usage

```
xspline(x, y = NULL, shape = 0, open = TRUE, repEnds = TRUE, draw = TRUE,
        border = par("fg"), col = NA, ...)
```

Arguments

`x,y`	vectors containing the coordinates of the vertices of the polygon. See `xy.coords` for alternatives.
`shape`	A numeric vector of values between -1 and 1, which control the shape of the spline relative to the control points.
`open`	A logical value indicating whether the spline is an open or a closed shape.

repEnds	For open X-splines, a logical value indicating whether the first and last control points should be replicated for drawing the curve. Ignored for closed X-splines.
draw	logical: should the X-spline be drawn? If false, a set of line segments to draw the curve is returned, and nothing is drawn.
border	the color to draw the curve. Use `border = NA` to omit borders.
col	the color for filling the shape. The default, NA, is to leave unfilled.
...	graphical parameters such as lty, xpd, lend, ljoin and lmitre can be given as arguments.

Details

An X-spline is a line drawn relative to control points. For each control point, the line may pass through (interpolate) the control point or it may only approach (approximate) the control point; the behaviour is determined by a shape parameter for each control point.

If the shape parameter is greater than zero, the spline approximates the control points (and is very similar to a cubic B-spline when the shape is 1). If the shape parameter is less than zero, the spline interpolates the control points (and is very similar to a Catmull-Rom spline when the shape is -1). If the shape parameter is 0, the spline forms a sharp corner at that control point.

For open X-splines, the start and end control points must have a shape of 0 (and non-zero values are silently converted to zero).

For open X-splines, by default the start and end control points are replicated before the curve is drawn. A curve is drawn between (interpolating or approximating) the second and third of each set of four control points, so this default behaviour ensures that the resulting curve starts at the first control point you have specified and ends at the last control point. The default behaviour can be turned off via the repEnds argument.

Value

If `draw = TRUE`, NULL otherwise a list with elements x and y which could be passed to lines, polygon and so on.

Invisible in both cases.

Note

Two-dimensional splines need to be created in an isotropic coordinate system. Device coordinates are used (with an anisotropy correction if needed.)

References

Blanc, C. and Schlick, C. (1995), *X-splines : A Spline Model Designed for the End User*, in *Proceedings of SIGGRAPH 95*, pp. 377–386. http://dept-info.labri.fr/~schlick/DOC/sig1.html

See Also

polygon.

par for how to specify colors.

Examples

```
## based on examples in ?grid.xspline

xsplineTest <- function(s, open = TRUE,
                        x = c(1,1,3,3)/4,
                        y = c(1,3,3,1)/4, ...) {
    plot(c(0,1), c(0,1), type="n", axes=FALSE, xlab="", ylab="")
    points(x, y, pch=19)
    xspline(x, y, s, open, ...)
    text(x+0.05*c(-1,-1,1,1), y+0.05*c(-1,1,1,-1), s)
}
op <- par(mfrow=c(3,3), mar=rep(0,4), oma=c(0,0,2,0))
xsplineTest(c(0, -1, -1, 0))
xsplineTest(c(0, -1,  0, 0))
xsplineTest(c(0, -1,  1, 0))
xsplineTest(c(0,  0, -1, 0))
xsplineTest(c(0,  0,  0, 0))
xsplineTest(c(0,  0,  1, 0))
xsplineTest(c(0,  1, -1, 0))
xsplineTest(c(0,  1,  0, 0))
xsplineTest(c(0,  1,  1, 0))
title("Open X-splines", outer=TRUE)

par(mfrow=c(3,3), mar=rep(0,4), oma=c(0,0,2,0))
xsplineTest(c(0, -1, -1, 0), FALSE, col="grey80")
xsplineTest(c(0, -1,  0, 0), FALSE, col="grey80")
xsplineTest(c(0, -1,  1, 0), FALSE, col="grey80")
xsplineTest(c(0,  0, -1, 0), FALSE, col="grey80")
xsplineTest(c(0,  0,  0, 0), FALSE, col="grey80")
xsplineTest(c(0,  0,  1, 0), FALSE, col="grey80")
xsplineTest(c(0,  1, -1, 0), FALSE, col="grey80")
xsplineTest(c(0,  1,  0, 0), FALSE, col="grey80")
xsplineTest(c(0,  1,  1, 0), FALSE, col="grey80")
title("Closed X-splines", outer=TRUE)

par(op)

x <- sort(stats::rnorm(5))
y <- sort(stats::rnorm(5))
plot(x, y, pch=19)
res <- xspline(x, y, 1, draw=FALSE)
lines(res)
## the end points may be very close together,
## so use last few for direction
nr <- length(res$x)
arrows(res$x[1], res$y[1], res$x[4], res$y[4], code=1, length=0.1)
arrows(res$x[nr-3], res$y[nr-3], res$x[nr], res$y[nr],
       code = 2, length = 0.1)
```

Chapter 5

The `grid` package

`grid-package` *The Grid Graphics Package*

Description

A rewrite of the graphics layout capabilities, plus some support for interaction.

Details

This package contains a graphics system which supplements S-style graphics (see the **graphics** package).

Further information is available in the following vignettes:

`grid`	Introduction to `grid` (`../doc/grid.pdf`)
`displaylist`	Display Lists in `grid` (`../doc/displaylist.pdf`)
`frame`	Frames and packing grobs (`../doc/frame.pdf`)
`grobs`	Working with `grid` grobs (`../doc/grobs.pdf`)
`interactive`	Editing `grid` Graphics (`../doc/interactive.pdf`)
`locndimn`	Locations versus Dimensions (`../doc/locndimn.pdf`)
`moveline`	Demonstrating move-to and line-to (`../doc/moveline.pdf`)
`nonfinite`	How `grid` responds to non-finite values (`../doc/nonfinite.pdf`)
`plotexample`	Writing `grid` Code (`../doc/plotexample.pdf`)
`rotated`	Rotated Viewports (`../doc/rotated.pdf`)
`saveload`	Persistent representations (`../doc/saveload.pdf`)
`sharing`	Modifying multiple grobs simultaneously (`../doc/sharing.pdf`)
`viewports`	Working with `grid` viewports (`../doc/viewports.pdf`)

For a complete list of functions with individual help pages, use `library(help="grid")`.

Author(s)

Paul Murrell ⟨paul@stat.auckland.ac.nz⟩

Maintainer: R Core Team ⟨R-core@r-project.org⟩

References

Murrell, P. (2005) *R Graphics*. Chapman & Hall/CRC Press.

`absolute.size` *Absolute Size of a Grob*

Description

This function converts a unit object into absolute units. Absolute units are unaffected, but non-absolute units are converted into `"null"` units.

Usage

```
absolute.size(unit)
```

Arguments

`unit`	An object of class `"unit"`.

Details

Absolute units are things like `"inches"`, `"cm"`, and `"lines"`. Non-absolute units are `"npc"` and `"native"`.

This function is designed to be used in `widthDetails` and `heightDetails` methods.

Value

An object of class `"unit"`.

Author(s)

Paul Murrell

See Also

`widthDetails` and `heightDetails` methods.

arrow *Describe arrows to add to a line.*

Description

Produces a description of what arrows to add to a line. The result can be passed to a function that draws a line, e.g., `grid.lines`.

Usage

```
arrow(angle = 30, length = unit(0.25, "inches"),
      ends = "last", type = "open")
```

Arguments

`angle`	The angle of the arrow head in degrees (smaller numbers produce narrower, pointier arrows). Essentially describes the width of the arrow head.
`length`	A unit specifying the length of the arrow head (from tip to base).
`ends`	One of `"last"`, `"first"`, or `"both"`, indicating which ends of the line to draw arrow heads.
`type`	One of `"open"` or `"closed"` indicating whether the arrow head should be a closed triangle.

Examples

```
arrow()
```

convertNative *Convert a Unit Object to Native units*

Description

This function is deprecated in grid version 0.8 and will be made defunct in grid version 1.9

You should use the `convertUnit()` function or one of its close allies instead.

This function returns a numeric vector containing the specified x or y locations or dimensions, converted to "user" or "data" units, relative to the current viewport.

Usage

```
convertNative(unit, dimension="x", type="location")
```

Arguments

`unit`	A unit object.
`dimension`	Either "x" or "y".
`type`	Either "location" or "dimension".

Value

A numeric vector.

WARNING

If you draw objects based on output from these conversion functions, then resize your device, the objects will be drawn incorrectly – the base R display list will not recalculate these conversions. This means that you can only rely on the results of these calculations if the size of your device is fixed.

Author(s)

Paul Murrell

See Also

grid.convert, unit

Examples

```
grid.newpage()
pushViewport(viewport(width=unit(.5, "npc"),
                      height=unit(.5, "npc")))
grid.rect()
w <- convertNative(unit(1, "inches"))
h <- convertNative(unit(1, "inches"), "y")
# This rectangle starts off life as 1in square, but if you
# resize the device it will no longer be 1in square
grid.rect(width=unit(w, "native"), height=unit(h, "native"),
          gp=gpar(col="red"))
popViewport(1)

# How to use grid.convert(), etc instead
convertNative(unit(1, "inches")) ==
  convertX(unit(1, "inches"), "native", valueOnly=TRUE)
convertNative(unit(1, "inches"), "y", "dimension") ==
  convertHeight(unit(1, "inches"), "native", valueOnly=TRUE)
```

---

dataViewport *Create a Viewport with Scales based on Data*

Description

This is a convenience function for producing a viewport with x- and/or y-scales based on numeric values passed to the function.

Usage

```
dataViewport(xData = NULL, yData = NULL, xscale = NULL,
             yscale = NULL, extension = 0.05, ...)
```

Arguments

xData	A numeric vector of data.
yData	A numeric vector of data.
xscale	A numeric vector (length 2).
yscale	A numeric vector (length 2).
extension	A numeric. If length greater than 1, then first value is used to extend the xscale and second value is used to extend the yscale.
...	All other arguments will be passed to a call to the viewport() function.

Details

If xscale is not specified then the values in x are used to generate an x-scale based on the range of x, extended by the proportion specified in extension. Similarly for the y-scale.

Value

A grid viewport object.

Author(s)

Paul Murrell

See Also

viewport and plotViewport.

---

drawDetails *Customising grid Drawing*

Description

These generic hook functions are called whenever a grid grob is drawn. They provide an opportunity for customising the drawing of a new class derived from grob (or gTree).

Usage

```
drawDetails(x, recording)
draw.details(x, recording)
preDrawDetails(x)
postDrawDetails(x)
```

Arguments

x	A grid grob.
recording	A logical value indicating whether a grob is being added to the display list or redrawn from the display list.

Details

These functions are called by the `grid.draw` methods for grobs and gTrees.

`preDrawDetails` is called first during the drawing of a grob. This is where any additional viewports should be pushed (see, for example, `grid:::preDrawDetails.frame`). Note that the default behaviour for grobs is to push any viewports in the `vp` slot, and for gTrees is to also push and up any viewports in the `childrenvp` slot so there is typically nothing to do here.

`drawDetails` is called next and is where any additional calculations and graphical output should occur (see, for example, `grid:::drawDetails.xaxis`. Note that the default behaviour for gTrees is to draw all grobs in the `children` slot so there is typically nothing to do here.

`postDrawDetails` is called last and should reverse anything done in `preDrawDetails` (i.e., pop or up any viewports that were pushed; again, see, for example, `grid:::postDrawDetails.frame`). Note that the default behaviour for grobs is to pop any viewports that were pushed so there is typically nothing to do here.

Note that `preDrawDetails` and `postDrawDetails` are also called in the calculation of `"grobwidth"` and `"grobheight"` units.

Value

None of these functions are expected to return a value.

Author(s)

Paul Murrell

See Also

`grid.draw`

`editDetails` *Customising grid Editing*

Description

This generic hook function is called whenever a grid grob is edited via `grid.edit` or `editGrob`. This provides an opportunity for customising the editing of a new class derived from grob (or gTree).

Usage

```
editDetails(x, specs)
```

Arguments

`x`	A grid grob.
`specs`	A list of named elements. The names indicate the grob slots to modify and the values are the new values for the slots.

Details

This function is called by `grid.edit` and `editGrob`. A method should be written for classes derived from grob or gTree if a change in a slot has an effect on other slots in the grob or children of a gTree (e.g., see `grid:::editDetails.xaxis`).

Note that the slot already has the new value.

Value

The function MUST return the modified grob.

Author(s)

Paul Murrell

See Also

`grid.edit`

---

gEdit *Create and Apply Edit Objects*

Description

The functions `gEdit` and `gEditList` create objects representing an edit operation (essentially a list of arguments to `editGrob`).

The functions `applyEdit` and `applyEdits` apply one or more edit operations to a graphical object.

These functions are most useful for developers creating new graphical functions and objects.

Usage

```
gEdit(...)
gEditList(...)
applyEdit(x, edit)
applyEdits(x, edits)
```

Arguments

`...`	one or more arguments to the `editGrob` function (for `gEdit`) or one or more `"gEdit"` objects (for `gEditList`).
`x`	a grob (grid graphical object).
`edit`	a `"gEdit"` object.
`edits`	either a `"gEdit"` object or a `"gEditList"` object.

Value

`gEdit` returns an object of class `"gEdit"`.

`gEditList` returns an object of class `"gEditList"`.

`applyEdit` and `applyEditList` return the modified grob.

Author(s)

Paul Murrell

See Also

grob editGrob

Examples

```
grid.rect(gp=gpar(col="red"))
# same thing, but more verbose
grid.draw(applyEdit(rectGrob(), gEdit(gp=gpar(col="red"))))
```

---

getNames *List the names of grobs on the display list*

Description

Returns a character vector containing the names of all top-level grobs on the display list.

Usage

```
getNames()
```

Value

A character vector.

Author(s)

Paul Murrell

Examples

```
grid.grill()
getNames()
```

---

gpar *Handling Grid Graphical Parameters*

Description

`gpar()` should be used to create a set of graphical parameter settings. It returns an object of class `"gpar"`. This is basically a list of name-value pairs.

`get.gpar()` can be used to query the current graphical parameter settings.

Usage

```
gpar(...)
get.gpar(names = NULL)
```

Arguments

...	Any number of named arguments.
names	A character vector of valid graphical parameter names.

Details

All grid viewports and (predefined) graphical objects have a slot called gp, which contains a "gpar" object. When a viewport is pushed onto the viewport stack and when a graphical object is drawn, the settings in the "gpar" object are enforced. In this way, the graphical output is modified by the gp settings until the graphical object has finished drawing, or until the viewport is popped off the viewport stack, or until some other viewport or graphical object is pushed or begins drawing.

Valid parameter names are:

col	Colour for lines and borders.
fill	Colour for filling rectangles, polygons, ...
alpha	Alpha channel for transparency
lty	Line type
lwd	Line width
lex	Multiplier applied to line width
lineend	Line end style (round, butt, square)
linejoin	Line join style (round, mitre, bevel)
linemitre	Line mitre limit (number greater than 1)
fontsize	The size of text (in points)
cex	Multiplier applied to fontsize
fontfamily	The font family
fontface	The font face (bold, italic, ...)
lineheight	The height of a line as a multiple of the size of text
font	Font face (alias for fontface; for backward compatibility)

Colours can be specified in one of the forms returned by rgb, as a name (see colors) or as a positive integer index into the current palette (with zero or negative values being taken as transparent).

The alpha setting is combined with the alpha channel for individual colours by multiplying (with both alpha settings normalised to the range 0 to 1).

The size of text is fontsize*cex. The size of a line is fontsize*cex*lineheight.

The cex setting is cumulative; if a viewport is pushed with a cex of 0.5 then another viewport is pushed with a cex of 0.5, the effective cex is 0.25.

The alpha and lex settings are also cumulative.

Changes to the fontfamily may be ignored by some devices, but is supported by PostScript, PDF, X11, Windows, and Quartz. The fontfamily may be used to specify one of the Hershey Font families (e.g., HersheySerif) and this specification will be honoured on all devices.

The specification of fontface can be an integer or a string. If an integer, then it follows the R base graphics standard: 1 = plain, 2 = bold, 3 = italic, 4 = bold italic. If a string, then valid values are: "plain", "bold", "italic", "oblique", and "bold.italic". For the special case of the HersheySerif font family, "cyrillic", "cyrillic.oblique", and "EUC" are also available.

Specifying the value NULL for a parameter is the same as not specifying any value for that parameter, except for col and fill, where NULL indicates not to draw a border or not to fill an area (respectively).

All parameter values can be vectors of multiple values. (This will not always make sense – for example, viewports will only take notice of the first parameter value.)

The `gamma` parameter is defunct since R 2.7.0.

`get.gpar()` returns all current graphical parameter settings.

Value

An object of class `"gpar"`.

Author(s)

Paul Murrell

See Also

`Hershey`.

Examples

```
gp <- get.gpar()
utils::str(gp)
## These *do* nothing but produce a "gpar" object:
gpar(col = "red")
gpar(col = "blue", lty = "solid", lwd = 3, fontsize = 16)
get.gpar(c("col", "lty"))
grid.newpage()
vp <- viewport(w = .8, h = .8, gp = gpar(col="blue"))
grid.draw(gTree(children=gList(rectGrob(gp = gpar(col="red")),
                     textGrob(paste("The rect is its own colour (red)",
                                    "but this text is the colour",
                                    "set by the gTree (green)",
                                    sep = "\n"))),
      gp = gpar(col="green"), vp = vp))
grid.text("This text is the colour set by the viewport (blue)",
          y = 1, just = c("center", "bottom"),
          gp = gpar(fontsize=20), vp = vp)
grid.newpage()
## example with multiple values for a parameter
pushViewport(viewport())
grid.points(1:10/11, 1:10/11, gp = gpar(col=1:10))
popViewport()
```

---

gPath *Concatenate Grob Names*

Description

This function can be used to generate a grob path for use in `grid.edit` and friends.

A grob path is a list of nested grob names.

Usage

```
gPath(...)
```

Arguments

`...`	Character values which are grob names.

Details

Grob names must only be unique amongst grobs which share the same parent in a gTree.

This function can be used to generate a specification for a grob that includes the grob's parent's name (and the name of its parent and so on).

For interactive use, it is possible to directly specify a path, but it is strongly recommended that this function is used otherwise in case the path separator is changed in future versions of grid.

Value

A `gPath` object.

See Also

`grob`, `editGrob`, `addGrob`, `removeGrob`, `getGrob`, `setGrob`

Examples

```
gPath("g1", "g2")
```

`Grid` *Grid Graphics*

Description

General information about the grid graphics package.

Details

Grid graphics provides an alternative to the standard R graphics. The user is able to define arbitrary rectangular regions (called *viewports*) on the graphics device and define a number of coordinate systems for each region. Drawing can be specified to occur in any viewport using any of the available coordinate systems.

Grid graphics and standard R graphics do not mix!

Type `library(help = grid)` to see a list of (public) Grid graphics functions.

Author(s)

Paul Murrell

See Also

`viewport`, `grid.layout`, and `unit`.

Examples

```
## Diagram of a simple layout
grid.show.layout(grid.layout(4,2,
                     heights=unit(rep(1, 4),
                                  c("lines", "lines", "lines", "null")),
                     widths=unit(c(1, 1), "inches")))
## Diagram of a sample viewport
grid.show.viewport(viewport(x=0.6, y=0.6,
                            w=unit(1, "inches"), h=unit(1, "inches")))
## A flash plotting example
grid.multipanel(vp=viewport(0.5, 0.5, 0.8, 0.8))
```

Grid Viewports *Create a Grid Viewport*

Description

These functions create viewports, which describe rectangular regions on a graphics device and define a number of coordinate systems within those regions.

Usage

```
viewport(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
         width = unit(1, "npc"), height = unit(1, "npc"),
         default.units = "npc", just = "centre",
         gp = gpar(), clip = "inherit",
         xscale = c(0, 1), yscale = c(0, 1),
         angle = 0,
         layout = NULL,
         layout.pos.row = NULL, layout.pos.col = NULL,
         name = NULL)
vpList(...)
vpStack(...)
vpTree(parent, children)
```

Arguments

`x`	A numeric vector or unit object specifying x-location.
`y`	A numeric vector or unit object specifying y-location.
`width`	A numeric vector or unit object specifying width.
`height`	A numeric vector or unit object specifying height.
`default.units`	A string indicating the default units to use if `x`, `y`, `width`, or `height` are only given as numeric vectors.
`just`	A string or numeric vector specifying the justification of the viewport relative to its (x, y) location. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible string values are: `"left"`, `"right"`, `"centre"`, `"center"`, `"bottom"`, and `"top"`. For numeric values, 0 means left alignment and 1 means right alignment.

gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
clip	One of "on", "inherit", or "off", indicating whether to clip to the extent of this viewport, inherit the clipping region from the parent viewport, or turn clipping off altogether. For back-compatibility, a logical value of TRUE corresponds to "on" and FALSE corresponds to "inherit".
xscale	A numeric vector of length two indicating the minimum and maximum on the x-scale.
yscale	A numeric vector of length two indicating the minimum and maximum on the y-scale.
angle	A numeric value indicating the angle of rotation of the viewport. Positive values indicate the amount of rotation, in degrees, anticlockwise from the positive x-axis.
layout	A Grid layout object which splits the viewport into subregions.
layout.pos.row	A numeric vector giving the rows occupied by this viewport in its parent's layout.
layout.pos.col	A numeric vector giving the columns occupied by this viewport in its parent's layout.
name	A character value to uniquely identify the viewport once it has been pushed onto the viewport tree.
...	Any number of grid viewport objects.
parent	A grid viewport object.
children	A vpList object.

Details

The location and size of a viewport are relative to the coordinate systems defined by the viewport's parent (either a graphical device or another viewport). The location and size can be specified in a very flexible way by specifying them with unit objects. When specifying the location of a viewport, specifying both layout.pos.row and layout.pos.col as NULL indicates that the viewport ignores its parent's layout and specifies its own location and size (via its locn). If only one of layout.pos.row and layout.pos.col is NULL, this means occupy ALL of the appropriate row(s)/column(s). For example, layout.pos.row = 1 and layout.pos.col = NULL means occupy all of row 1. Specifying non-NULL values for both layout.pos.row and layout.pos.col means occupy the intersection of the appropriate rows and columns. If a vector of length two is specified for layout.pos.row or layout.pos.col, this indicates a range of rows or columns to occupy. For example, layout.pos.row = c(1, 3) and layout.pos.col = c(2, 4) means occupy cells in the intersection of rows 1, 2, and 3, and columns, 2, 3, and 4.

Clipping obeys only the most recent viewport clip setting. For example, if you clip to viewport1, then clip to viewport2, the clipping region is determined wholly by viewport2, the size and shape of viewport1 is irrelevant (until viewport2 is popped of course).

If a viewport is rotated (because of its own angle setting or because it is within another viewport which is rotated) then the clip flag is ignored.

Viewport names need not be unique. When pushed, viewports sharing the same parent must have unique names, which means that if you push a viewport with the same name as an existing viewport, the existing viewport will be replaced in the viewport tree. A viewport name can be any string, but

grid uses the reserved name "ROOT" for the top-level viewport. Also, when specifying a viewport name in downViewport and seekViewport, it is possible to provide a viewport path, which consists of several names concatenated using the separator (currently ::). Consequently, it is not advisable to use this separator in viewport names.

The viewports in a vpList are pushed in parallel. The viewports in a vpStack are pushed in series. When a vpTree is pushed, the parent is pushed first, then the children are pushed in parallel.

Value

An R object of class viewport.

Author(s)

Paul Murrell

See Also

Grid, pushViewport, popViewport, downViewport, seekViewport, upViewport, unit, grid.layout, grid.show.layout.

Examples

```
# Diagram of a sample viewport
grid.show.viewport(viewport(x=0.6, y=0.6,
                   w=unit(1, "inches"), h=unit(1, "inches")))
# Demonstrate viewport clipping
clip.demo <- function(i, j, clip1, clip2) {
  pushViewport(viewport(layout.pos.col=i,
                        layout.pos.row=j))
  pushViewport(viewport(width=0.6, height=0.6, clip=clip1))
  grid.rect(gp=gpar(fill="white"))
  grid.circle(r=0.55, gp=gpar(col="red", fill="pink"))
  popViewport()
  pushViewport(viewport(width=0.6, height=0.6, clip=clip2))
  grid.polygon(x=c(0.5, 1.1, 0.6, 1.1, 0.5, -0.1, 0.4, -0.1),
               y=c(0.6, 1.1, 0.5, -0.1, 0.4, -0.1, 0.5, 1.1),
               gp=gpar(col="blue", fill="light blue"))
  popViewport(2)
}

grid.newpage()
grid.rect(gp=gpar(fill="grey"))
pushViewport(viewport(layout=grid.layout(2, 2)))
clip.demo(1, 1, FALSE, FALSE)
clip.demo(1, 2, TRUE, FALSE)
clip.demo(2, 1, FALSE, TRUE)
clip.demo(2, 2, TRUE, TRUE)
popViewport()
# Demonstrate turning clipping off
grid.newpage()
pushViewport(viewport(w=.5, h=.5, clip="on"))
grid.rect()
grid.circle(r=.6, gp=gpar(lwd=10))
pushViewport(viewport(clip="inherit"))
grid.circle(r=.6, gp=gpar(lwd=5, col="grey"))
```

```
pushViewport(viewport(clip="off"))
grid.circle(r=.6)
popViewport(3)
# Demonstrate vpList, vpStack, and vpTree
grid.newpage()
tree <- vpTree(viewport(w=0.8, h=0.8, name="A"),
               vpList(vpStack(viewport(x=0.1, y=0.1, w=0.5, h=0.5,
                                       just=c("left", "bottom"), name="B"),
                              viewport(x=0.1, y=0.1, w=0.5, h=0.5,
                                       just=c("left", "bottom"), name="C"),
                              viewport(x=0.1, y=0.1, w=0.5, h=0.5,
                                       just=c("left", "bottom"), name="D")),
                      viewport(x=0.5, w=0.4, h=0.9,
                               just="left", name="E")))
pushViewport(tree)
for (i in LETTERS[1:5]) {
  seekViewport(i)
  grid.rect()
  grid.text(current.vpTree(FALSE),
            x=unit(1, "mm"), y=unit(1, "npc") - unit(1, "mm"),
            just=c("left", "top"),
            gp=gpar(fontsize=8))
}
```

`grid.add` *Add a Grid Graphical Object*

Description

Add a grob to a gTree or a descendant of a gTree.

Usage

```
grid.add(gPath, child, strict = FALSE, grep = FALSE,
         global = FALSE, allDevices = FALSE, redraw = TRUE)

addGrob(gTree, child, gPath = NULL, strict = FALSE, grep = FALSE,
        global = FALSE)

setChildren(x, children)
```

Arguments

`gTree, x`	A gTree object.
`gPath`	A gPath object. For `grid.add` this specifies a gTree on the display list. For `addGrob` this specifies a descendant of the specified gTree.
`child`	A grob object.
`children`	A gList object.
`strict`	A boolean indicating whether the gPath must be matched exactly.
`grep`	A boolean indicating whether the `gPath` should be treated as a regular expression. Values are recycled across elements of the `gPath` (e.g., `c(TRUE, FALSE)` means that every odd element of the `gPath` will be treated as a regular expression).

global	A boolean indicating whether the function should affect just the first match of the gPath, or whether all matches should be affected.
allDevices	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
redraw	A logical value to indicate whether to redraw the grob.

Details

addGrob copies the specified grob and returns a modified grob.

grid.add destructively modifies a grob on the display list. If redraw is TRUE it then redraws everything to reflect the change.

setChildren is a basic function for setting all children of a gTree at once (instead of repeated calls to addGrob).

Value

addGrob returns a grob object; grid.add returns NULL.

Author(s)

Paul Murrell

See Also

grob, getGrob, addGrob, removeGrob.

---

grid.arrows *Draw Arrows*

Description

Functions to create and draw arrows at either end of a line, or at either end of a line.to, lines, or segments grob.

These functions have been deprecated in favour of arrow arguments to the line-related primitives.

Usage

```
grid.arrows(x = c(0.25, 0.75), y = 0.5, default.units = "npc",
            grob = NULL,
            angle = 30, length = unit(0.25, "inches"),
            ends = "last", type = "open", name = NULL,
            gp = gpar(), draw = TRUE, vp = NULL)

arrowsGrob(x = c(0.25, 0.75), y = 0.5, default.units = "npc",
            grob = NULL,
            angle = 30, length = unit(0.25, "inches"),
            ends = "last", type = "open", name = NULL,
            gp = gpar(), vp = NULL)
```

Arguments

x	A numeric vector or unit object specifying x-values.
y	A numeric vector or unit object specifying y-values.
default.units	A string indicating the default units to use if x or y are only given as numeric vectors.
grob	A grob to add arrows to; currently can only be a line.to, lines, or segments grob.
angle	A numeric specifying (half) the width of the arrow head (in degrees).
length	A unit object specifying the length of the arrow head.
ends	One of "first", "last", or "both", indicating which end of the line to add arrow heads.
type	Either "open" or "closed" to indicate the type of arrow head.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create an arrows grob (a graphical object describing arrows), but only grid.arrows() draws the arrows (and then only if draw is TRUE).

If the grob argument is specified, this overrides any x and/or y arguments.

Value

An arrows grob. grid.arrows() returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport, grid.line.to, grid.lines, grid.segments

Examples

```
## Not run:
## to avoid lots of deprecation warnings
pushViewport(viewport(layout=grid.layout(2, 4)))
pushViewport(viewport(layout.pos.col=1,
                      layout.pos.row=1))
grid.rect(gp=gpar(col="grey"))
grid.arrows()
popViewport()
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=1))
grid.rect(gp=gpar(col="grey"))
grid.arrows(angle=15, type="closed")
```

```
popViewport()
pushViewport(viewport(layout.pos.col=3,
                      layout.pos.row=1))
grid.rect(gp=gpar(col="grey"))
grid.arrows(angle=5, length=unit(0.1, "npc"),
            type="closed", gp=gpar(fill="white"))
popViewport()
pushViewport(viewport(layout.pos.col=4,
                      layout.pos.row=1))
grid.rect(gp=gpar(col="grey"))
grid.arrows(x=unit(0:80/100, "npc"),
            y=unit(1 - (0:80/100)^2, "npc"))
popViewport()
pushViewport(viewport(layout.pos.col=1,
                      layout.pos.row=2))
grid.rect(gp=gpar(col="grey"))
grid.arrows(ends="both")
popViewport()
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=2))
grid.rect(gp=gpar(col="grey"))
# Recycling arguments
grid.arrows(x=unit(1:10/11, "npc"), y=unit(1:3/4, "npc"))
popViewport()
pushViewport(viewport(layout.pos.col=3,
                      layout.pos.row=2))
grid.rect(gp=gpar(col="grey"))
# Drawing arrows on a segments grob
gs <- segmentsGrob(x0=unit(1:4/5, "npc"),
                   x1=unit(1:4/5, "npc"))
grid.arrows(grob=gs, length=unit(0.1, "npc"),
            type="closed", gp=gpar(fill="white"))
popViewport()
pushViewport(viewport(layout.pos.col=4,
                      layout.pos.row=2))
grid.rect(gp=gpar(col="grey"))
# Arrows on a lines grob
# Name these because going to grid.edit them later
gl <- linesGrob(name="curve", x=unit(0:80/100, "npc"),
                y=unit((0:80/100)^2, "npc"))
grid.arrows(name="arrowOnLine", grob=gl, angle=15, type="closed",
            gp=gpar(fill="black"))
popViewport()
pushViewport(viewport(layout.pos.col=1,
                      layout.pos.row=2))
grid.move.to(x=0.5, y=0.8)
popViewport()
pushViewport(viewport(layout.pos.col=4,
                      layout.pos.row=1))
# Arrows on a line.to grob
glt <- lineToGrob(x=0.5, y=0.2, gp=gpar(lwd=3))
grid.arrows(grob=glt, ends="first", gp=gpar(lwd=3))
popViewport(2)
grid.edit(gPath("arrowOnLine", "curve"), y=unit((0:80/100)^3, "npc"))
## End(Not run)
```

grid.circle *Draw a Circle*

Description

Functions to create and draw a circle.

Usage

```
grid.circle(x=0.5, y=0.5, r=0.5, default.units="npc", name=NULL,
            gp=gpar(), draw=TRUE, vp=NULL)
circleGrob(x=0.5, y=0.5, r=0.5, default.units="npc", name=NULL,
            gp=gpar(), vp=NULL)
```

Arguments

x	A numeric vector or unit object specifying x-locations.
y	A numeric vector or unit object specifying y-locations.
r	A numeric vector or unit object specifying radii.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a circle grob (a graphical object describing a circle), but only grid.circle() draws the circle (and then only if draw is TRUE).

The radius may be given in any units; if the units are *relative* (e.g., "npc" or "native") then the radius will be different depending on whether it is interpreted as a width or as a height. In such cases, the smaller of these two values will be the result. To see the effect, type grid.circle() and adjust the size of the window.

Value

A circle grob. grid.circle() returns the value invisibly.

Warning

Negative values for the radius are silently converted to their absolute value.

Author(s)

Paul Murrell

See Also

Grid, viewport

---

grid.clip *Set the Clipping Region*

Description

These functions set the clipping region within the current viewport *without* altering the current coordinate system.

Usage

```
grid.clip(...)
clipGrob(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
         width = unit(1, "npc"), height = unit(1, "npc"),
         just = "centre", hjust = NULL, vjust = NULL,
         default.units = "npc", name = NULL, vp = NULL)
```

Arguments

x	A numeric vector or unit object specifying x-location.
y	A numeric vector or unit object specifying y-location.
width	A numeric vector or unit object specifying width.
height	A numeric vector or unit object specifying height.
just	The justification of the clip rectangle relative to its (x, y) location. If there are two values, the first value specifes horizontal justification and the second value specifies vertical justification. Possible string values are: "left", "right", "centre", "center", "bottom", and "top". For numeric values, 0 means left alignment and 1 means right alignment.
hjust	A numeric vector specifying horizontal justification. If specified, overrides the just setting.
vjust	A numeric vector specifying vertical justification. If specified, overrides the just setting.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
vp	A Grid viewport object (or NULL).
...	Arguments passed to clipGrob.

Details

Both functions create a clip rectangle (a graphical object describing a clip rectangle), but only grid.clip enforces the clipping.

Pushing or popping a viewport *always* overrides the clip region set by a clip grob, regardless of whether that viewport explicitly enforces a clipping region.

Value

`clipGrob` returns a clip grob.

Author(s)

Paul Murrell

See Also

Grid, `viewport`

Examples

```
# draw across entire viewport, but clipped
grid.clip(x = 0.3, width = 0.1)
grid.lines(gp=gpar(col="green", lwd=5))
# draw across entire viewport, but clipped (in different place)
grid.clip(x = 0.7, width = 0.1)
grid.lines(gp=gpar(col="red", lwd=5))
# Viewport sets new clip region
pushViewport(viewport(width=0.5, height=0.5, clip=TRUE))
grid.lines(gp=gpar(col="grey", lwd=3))
# Return to original viewport;  get
# clip region from previous grid.clip()
# (NOT from previous viewport clip region)
popViewport()
grid.lines(gp=gpar(col="black"))
```

`grid.collection` *Create a Coherent Group of Grid Graphical Objects*

Description

This function is deprecated; please use `gTree`.

This function creates a graphical object which contains several other graphical objects. When it is drawn, it draws all of its children.

It may be convenient to name the elements of the collection.

Usage

```
grid.collection(..., gp=gpar(), draw=TRUE, vp=NULL)
```

Arguments

`...`	Zero or more objects of class `"grob"`.
`gp`	An object of class `gpar`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`draw`	A logical value to indicate whether to produce graphical output.
`vp`	A Grid viewport object (or NULL).

Value

A collection grob.

Author(s)

Paul Murrell

See Also

grid.grob.

`grid.convert` *Convert Between Different grid Coordinate Systems*

Description

These functions take a unit object and convert it to an equivalent unit object in a different coordinate system.

Usage

```
convertX(x, unitTo, valueOnly = FALSE)
convertY(x, unitTo, valueOnly = FALSE)
convertWidth(x, unitTo, valueOnly = FALSE)
convertHeight(x, unitTo, valueOnly = FALSE)
convertUnit(x, unitTo,
            axisFrom = "x", typeFrom = "location",
            axisTo = axisFrom, typeTo = typeFrom,
            valueOnly = FALSE)
grid.convertX(x, unitTo, valueOnly = FALSE)
grid.convertY(x, unitTo, valueOnly = FALSE)
grid.convertWidth(x, unitTo, valueOnly = FALSE)
grid.convertHeight(x, unitTo, valueOnly = FALSE)
grid.convert(x, unitTo,
            axisFrom = "x", typeFrom = "location",
            axisTo = axisFrom, typeTo = typeFrom,
            valueOnly = FALSE)
```

Arguments

`x`	A unit object.
`unitTo`	The coordinate system to convert the unit to. See the `unit` function for valid coordinate systems.
`axisFrom`	Either `"x"` or `"y"` to indicate whether the unit object represents a value in the x- or y-direction.
`typeFrom`	Either `"location"` or `"dimension"` to indicate whether the unit object represents a location or a length.
`axisTo`	Same as `axisFrom`, but applies to the unit object that is to be created.
`typeTo`	Same as `typeFrom`, but applies to the unit object that is to be created.
`valueOnly`	A logical indicating. If `TRUE` then the function does not return a unit object, but rather only the converted numeric values.

Details

The `convertUnit` function allows for general-purpose conversions. The other four functions are just more convenient front-ends to it for the most common conversions.

The conversions occur within the current viewport.

It is not currently possible to convert to all valid coordinate systems (e.g., "strwidth" or "grobwidth"). I'm not sure if all of these are impossible, they just seem implausible at this stage.

In normal usage of grid, these functions should not be necessary. If you want to express a location or dimension in inches rather than user coordinates then you should simply do something like `unit(1, "inches")` rather than something like `unit(0.134, "native")`.

In some cases, however, it is necessary for the user to perform calculations on a unit value and this function becomes necessary. In such cases, please take note of the warning below.

The grid.* versions are just previous incarnations which have been deprecated.

Value

A unit object in the specified coordinate system (unless `valueOnly` is `TRUE` in which case the returned value is a numeric).

Warning

The conversion is only valid for the current device size. If the device is resized then at least some conversions will become invalid. For example, suppose that I create a unit object as follows: `oneinch <- convertUnit(unit(1, "inches"), "native"`. Now if I resize the device, the unit object in oneinch no longer corresponds to a physical length of 1 inch.

Author(s)

Paul Murrell

See Also

`unit`

Examples

```
## A tautology
convertX(unit(1, "inches"), "inches")
## The physical units
convertX(unit(2.54, "cm"), "inches")
convertX(unit(25.4, "mm"), "inches")
convertX(unit(72.27, "points"), "inches")
convertX(unit(1/12*72.27, "picas"), "inches")
convertX(unit(72, "bigpts"), "inches")
convertX(unit(1157/1238*72.27, "dida"), "inches")
convertX(unit(1/12*1157/1238*72.27, "cicero"), "inches")
convertX(unit(65536*72.27, "scaledpts"), "inches")
convertX(unit(1/2.54, "inches"), "cm")
convertX(unit(1/25.4, "inches"), "mm")
convertX(unit(1/72.27, "inches"), "points")
convertX(unit(1/(1/12*72.27), "inches"), "picas")
convertX(unit(1/72, "inches"), "bigpts")
convertX(unit(1/(1157/1238*72.27), "inches"), "dida")
convertX(unit(1/(1/12*1157/1238*72.27), "inches"), "cicero")
```

```
convertX(unit(1/(65536*72.27), "inches"), "scaledpts")

pushViewport(viewport(width=unit(1, "inches"),
                      height=unit(2, "inches"),
                      xscale=c(0, 1),
                      yscale=c(1, 3)))
  ## Location versus dimension
  convertY(unit(2, "native"), "inches")
  convertHeight(unit(2, "native"), "inches")
  ## From "x" to "y" (the conversion is via "inches")
  convertUnit(unit(1, "native"), "native",
              axisFrom="x", axisTo="y")
  ## Convert several values at once
  convertX(unit(c(0.5, 2.54), c("npc", "cm")),
               c("inches", "native"))
popViewport()
## Convert a complex unit
convertX(unit(1, "strwidth", "Hello"), "native")
```

`grid.copy` *Make a Copy of a Grid Graphical Object*

Description

This function is redundant and will disappear in future versions.

Usage

```
grid.copy(grob)
```

Arguments

`grob`	A grob object.

Value

A copy of the grob object.

Author(s)

Paul Murrell

See Also

`grid.grob`.

grid.curve *Draw a Curve Between Locations*

Description

These functions create and draw a curve from one location to another.

Usage

```
grid.curve(...)
curveGrob(x1, y1, x2, y2, default.units = "npc",
          curvature = 1, angle = 90, ncp = 1, shape = 0.5,
          square = TRUE, squareShape = 1,
          inflect = FALSE, arrow = NULL, open = TRUE,
          debug = FALSE,
          name = NULL, gp = gpar(), vp = NULL)
arcCurvature(theta)
```

Arguments

x1	A numeric vector or unit object specifying the x-location of the start point.
y1	A numeric vector or unit object specifying the y-location of the start point.
x2	A numeric vector or unit object specifying the x-location of the end point.
y2	A numeric vector or unit object specifying the y-location of the end point.
default.units	A string indicating the default units to use if x1, y1, x2 or y2 are only given as numeric values.
curvature	A numeric value giving the amount of curvature. Negative values produce left-hand curves, positive values produce right-hand curves, and zero produces a straight line.
angle	A numeric value between 0 and 180, giving an amount to skew the control points of the curve. Values less than 90 skew the curve towards the start point and values greater than 90 skew the curve towards the end point.
ncp	The number of control points used to draw the curve. More control points creates a smoother curve.
shape	A numeric vector of values between -1 and 1, which control the shape of the curve relative to its control points. See grid.xspline for more details.
square	A logical value that controls whether control points for the curve are created city-block fashion or obliquely. When ncp is 1 and angle is 90, this is typically TRUE, otherwise this should probably be set to FALSE (see Examples below).
squareShape	A shape value to control the behaviour of the curve relative to any additional control point that is inserted if square is TRUE.
inflect	A logical value specifying whether the curve should be cut in half and inverted (see Examples below).
arrow	A list describing arrow heads to place at either end of the curve, as produced by the arrow function.

open	A logical value indicating whether to close the curve (connect the start and end points).
debug	A logical value indicating whether debugging information should be drawn.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).
...	Arguments to be passed to curveGrob.
theta	An angle (in degrees).

Details

Both functions create a curve grob (a graphical object describing an curve), but only grid.curve draws the curve.

The arcCurvature function can be used to calculate a curvature such that control points are generated on an arc corresponding to angle theta. This is typically used in conjunction with a large ncp to produce a curve corresponding to the desired arc.

Value

A grob object.

See Also

Grid, viewport, grid.xspline, arrow

Examples

```
curveTest <- function(i, j, ...) {
  pushViewport(viewport(layout.pos.col=j, layout.pos.row=i))
  do.call("grid.curve", c(list(x1=.25, y1=.25, x2=.75, y2=.75),
          if (is.null(...)) NULL else list(...)))
  grid.text(sub("list", "", deparse(substitute(list(...)))),
            y=unit(1, "npc"))
  popViewport()
}
# grid.newpage()
pushViewport(plotViewport(c(0, 0, 1, 0),
                          layout=grid.layout(2, 1, heights=c(2, 1))))
pushViewport(viewport(layout.pos.row=1,
                      layout=grid.layout(3, 3, respect=TRUE)))
curveTest(1, 1, NULL)
curveTest(1, 2, inflect=TRUE)
curveTest(1, 3, angle=135)
curveTest(2, 1, arrow=arrow())
curveTest(2, 2, ncp=8)
curveTest(2, 3, shape=0)
curveTest(3, 1, curvature=-1)
curveTest(3, 2, square=FALSE)
curveTest(3, 3, debug=TRUE)
popViewport()
pushViewport(viewport(layout.pos.row=2,
                      layout=grid.layout(3, 3)))
```

```
curveTest(1, 1, NULL)
curveTest(1, 2, inflect=TRUE)
curveTest(1, 3, angle=135)
curveTest(2, 1, arrow=arrow())
curveTest(2, 2, ncp=8)
curveTest(2, 3, shape=0)
curveTest(3, 1, curvature=-1)
curveTest(3, 2, square=FALSE)
curveTest(3, 3, debug=TRUE)
popViewport(2)
```

`grid.display.list` *Control the Grid Display List*

Description

Turn the Grid display list on or off.

Usage

```
grid.display.list(on=TRUE)
engine.display.list(on=TRUE)
```

Arguments

`on`	A logical value to indicate whether the display list should be on or off.

Details

All drawing and viewport-setting operations are (by default) recorded in the Grid display list. This allows redrawing to occur following an editing operation.

This display list could get very large so it may be useful to turn it off in some cases; this will of course disable redrawing.

All graphics output is also recorded on the main display list of the R graphics engine (by default). This supports redrawing following a device resize and allows copying between devices.

Turning off this display list means that grid will redraw from its own display list for device resizes and copies. This will be slower than using the graphics engine display list.

Value

None.

WARNING

Turning the display list on causes the display list to be erased!

Turning off both the grid display list and the graphics engine display list will result in no redrawing whatsoever.

Author(s)

Paul Murrell

grid.draw *Draw a grid grob*

Description

Produces graphical output from a graphical object.

Usage

```
grid.draw(x, recording=TRUE)
```

Arguments

x	An object of class "grob" or NULL.
recording	A logical value to indicate whether the drawing operation should be recorded on the Grid display list.

Details

This is a generic function with methods for grob and gTree objects.

The grob and gTree methods automatically push any viewports in a vp slot and automatically apply any gpar settings in a gp slot. In addition, the gTree method pushes and ups any viewports in a childrenvp slot and automatically calls grid.draw for any grobs in a children slot.

The methods for grob and gTree call the generic hook functions preDrawDetails, drawDetails, and postDrawDetails to allow classes derived from grob or gTree to perform additional viewport pushing/popping and produce additional output beyond the default behaviour for grobs and gTrees.

Value

None.

Author(s)

Paul Murrell

See Also

grob.

Examples

```
grid.newpage()
## Create a graphical object, but don't draw it
l <- linesGrob()
## Draw it
grid.draw(l)
```

grid.edit *Edit the Description of a Grid Graphical Object*

Description

Changes the value of one of the slots of a grob and redraws the grob.

Usage

```
grid.edit(gPath, ..., strict = FALSE, grep = FALSE,
          global = FALSE, allDevices = FALSE, redraw = TRUE)

grid.gedit(..., grep = TRUE, global = TRUE)

editGrob(grob, gPath = NULL, ..., strict = FALSE, grep = FALSE,
         global = FALSE)
```

Arguments

grob	A grob object.
...	Zero or more named arguments specifying new slot values.
gPath	A gPath object. For grid.edit this specifies a grob on the display list. For editGrob this specifies a descendant of the specified grob.
strict	A boolean indicating whether the gPath must be matched exactly.
grep	A boolean indicating whether the gPath should be treated as a regular expression. Values are recycled across elements of the gPath (e.g., c(TRUE, FALSE) means that every odd element of the gPath will be treated as a regular expression).
global	A boolean indicating whether the function should affect just the first match of the gPath, or whether all matches should be affected.
allDevices	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
redraw	A logical value to indicate whether to redraw the grob.

Details

editGrob copies the specified grob and returns a modified grob.

grid.edit destructively modifies a grob on the display list. If redraw is TRUE it then redraws everything to reflect the change.

Both functions call editDetails to allow a grob to perform custom actions and validDetails to check that the modified grob is still coherent.

grid.gedit (g for global) is just a convenience wrapper for grid.edit with different defaults.

Value

editGrob returns a grob object; grid.edit returns NULL.

Author(s)

Paul Murrell

See Also

grob, getGrob, addGrob, removeGrob.

Examples

```
grid.newpage()
grid.xaxis(name = "xa", vp = viewport(width=.5, height=.5))
grid.edit("xa", gp = gpar(col="red"))
# won't work because no ticks (at is NULL)
try(grid.edit(gPath("xa", "ticks"), gp = gpar(col="green")))
grid.edit("xa", at = 1:4/5)
# Now it should work
try(grid.edit(gPath("xa", "ticks"), gp = gpar(col="green")))
```

---

grid.frame *Create a Frame for Packing Objects*

Description

These functions, together with grid.pack, grid.place, packGrob, and placeGrob are part of a GUI-builder-like interface to constructing graphical images. The idea is that you create a frame with this function then use grid.pack or whatever to pack/place objects into the frame.

Usage

```
grid.frame(layout=NULL, name=NULL, gp=gpar(), vp=NULL, draw=TRUE)
frameGrob(layout=NULL, name=NULL, gp=gpar(), vp=NULL)
```

Arguments

layout	A Grid layout, or NULL. This can be used to initialise the frame with a number of rows and columns, with initial widths and heights, etc.
name	A character identifier.
vp	An object of class viewport, or NULL.
gp	An object of class gpar; typically the output from a call to the function gpar.
draw	Should the frame be drawn.

Details

Both functions create a frame grob (a graphical object describing a frame), but only grid.frame() draws the frame (and then only if draw is TRUE). Nothing will actually be drawn, but it will put the frame on the display list, which means that the output will be dynamically updated as objects are packed into the frame. Possibly useful for debugging.

Value

A frame grob. grid.frame() returns the value invisibly.

Author(s)

Paul Murrell

See Also

grid.pack

Examples

```
grid.newpage()
grid.frame(name="gf", draw=TRUE)
grid.pack("gf", rectGrob(gp=gpar(fill="grey")), width=unit(1, "null"))
grid.pack("gf", textGrob("hi there"), side="right")
```

---

grid.get *Get a Grid Graphical Object*

Description

Retrieve a grob or a descendant of a grob.

Usage

```
grid.get(gPath, strict = FALSE, grep = FALSE, global = FALSE,
         allDevices = FALSE)

grid.gget <- function(..., grep = TRUE, global = TRUE)

getGrob(gTree, gPath, strict = FALSE, grep = FALSE, global = FALSE)
```

Arguments

gTree	A gTree object.
gPath	A gPath object. For grid.get this specifyies a grob on the display list. For getGrob this specifies a descendant of the specified gTree.
strict	A boolean indicating whether the gPath must be matched exactly.
grep	A boolean indicating whether the gPath should be treated as a regular expression. Values are recycled across elements of the gPath (e.g., c(TRUE, FALSE) means that every odd element of the gPath will be treated as a regular expression).
global	A boolean indicating whether the function should affect just the first match of the gPath, or whether all matches should be affected.
allDevices	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.

Details

grid.gget (g for global) is just a convenience wrapper for grid.get with different defaults.

Value

A grob object.

Author(s)

Paul Murrell

See Also

grob, getGrob, addGrob, removeGrob.

Examples

```
grid.xaxis(name="xa")
grid.get("xa")
grid.get(gPath("xa", "ticks"))

grid.draw(gTree(name="gt", children=gList(xaxisGrob(name="axis"))))
grid.get(gPath("gt", "axis", "ticks"))
```

---

grid.grab *Grab the current grid output*

Description

Creates a gTree object from the current grid display list or from a scene generated by user-specified code.

Usage

```
grid.grab(warn = 2, wrap = FALSE, ...)
grid.grabExpr(expr, warn = 2, wrap = FALSE, ...)
```

Arguments

expr	An expression to be evaluated. Typically, some calls to grid drawing functions.
warn	An integer specifying the amount of warnings to emit. 0 means no warnings, 1 means warn when it is certain that the grab will not faithfully represent the original scene. 2 means warn if there's any possibility that the grab will not faithfully represent the original scene.
wrap	A logical indicating how the output should be captured. If TRUE, each non-grob element on the display list is captured by wrapping it in a grob.
...	arguments passed to gTree, for example, a name and/or class for the gTree that is created.

Details

There are four ways to capture grid output as a gTree.

There are two functions for capturing output: use `grid.grab` to capture an existing drawing and `grid.grabExpr` to capture the output from an expression (without drawing anything).

For each of these functions, the output can be captured in two ways. One way tries to be clever and make a gTree with a childrenvp slot containing all viewports on the display list (including those that are popped) and every grob on the display list as a child of the new gTree; each child has a vpPath in the vp slot so that it is drawn in the appropriate viewport. In other words, the gTree contains all elements on the display list, but in a slightly altered form.

The other way, `wrap=TRUE`, is to create a grob for every element on the display list (and make all of those grobs children of the gTree).

The first approach creates a more compact and elegant gTree, which is more flexible to work with, but is not guaranteed to faithfully replicate all possible grid output. The second approach is more brute force, and harder to work with, but should always faithfully replicate the original output.

Value

A gTree object.

See Also

gTree

Examples

```
pushViewport(viewport(w=.5, h=.5))
grid.rect()
grid.points(stats::runif(10), stats::runif(10))
popViewport()
grab <- grid.grab()
grid.newpage()
grid.draw(grab)
```

---

grid.grill *Draw a Grill*

Description

This function draws a grill within a Grid viewport.

Usage

```
grid.grill(h = unit(seq(0.25, 0.75, 0.25), "npc"),
           v = unit(seq(0.25, 0.75, 0.25), "npc"),
           default.units = "npc", gp=gpar(col = "grey"), vp = NULL)
```

Arguments

h	A numeric vector or unit object indicating the horizontal location of the vertical grill lines.
v	A numeric vector or unit object indicating the vertical location of the horizontal grill lines.
default.units	A string indicating the default units to use if h or v are only given as numeric vectors.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object.

Value

None.

Author(s)

Paul Murrell

See Also

Grid, viewport.

---

grid.grob *Create a Grid Graphical Object*

Description

These functions create grid graphical objects.

Usage

```
grid.grob(list.struct, cl = NULL, draw = TRUE)
grob(..., name = NULL, gp = NULL, vp = NULL, cl = NULL)
gTree(..., name = NULL, gp = NULL, vp = NULL, children = NULL,
      childrenvp = NULL, cl = NULL)
grobTree(..., name = NULL, gp = NULL, vp = NULL,
         childrenvp = NULL, cl = NULL)
childNames(gTree)
gList(...)
```

Arguments

...	For grob and gTree, the named slots describing important features of the graphical object. For gList and grobTree, a series of grob objects.
list.struct	A list (preferably with each element named).
name	A character identifier for the grob. Used to find the grob on the display list and/or as a child of another grob.

children	A gList object.
childrenvp	A viewport object (or NULL).
gp	A gpar object, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A viewport object (or NULL).
cl	A string giving the class attribute for the list.struct
draw	A logical value to indicate whether to produce graphical output.
gTree	A gTree object.

Details

These functions can be used to create a basic grob, gTree, or gList object, or a new class derived from one of these.

A grid graphical object (grob) is a description of a graphical item. These basic classes provide default behaviour for validating, drawing, and modifying graphical objects. Both call the function validDetails to check that the object returned is coherent.

A gTree can have other grobs as children; when a gTree is drawn, it draws all of its children. Before drawing its children, a gTree pushes its childrenvp slot and then navigates back up (calls upViewport) so that the children can specify their location within the childrenvp via a vpPath.

Grob names need not be unique in general, but all children of a gTree must have different names. A grob name can be any string, though it is not advisable to use the gPath separator (currently ::) in grob names.

The function childNames returns the names of the grobs which are children of a gTree.

All grid primitives (grid.lines, grid.rect, ...) and some higher-level grid components (e.g., grid.xaxis and grid.yaxis) are derived from these classes.

grobTree is just a convenient wrapper for gTree when the only components of the gTree are grobs (so all unnamed arguments become children of the gTree).

grid.grob is deprecated.

Value

A grob object.

Author(s)

Paul Murrell

See Also

grid.draw, grid.edit, grid.get.

grid.layout *Create a Grid Layout*

Description

This function returns a Grid layout, which describes a subdivision of a rectangular region.

Usage

```
grid.layout(nrow = 1, ncol = 1,
        widths = unit(rep(1, ncol), "null"),
        heights = unit(rep(1, nrow), "null"),
        default.units = "null", respect = FALSE,
        just="centre")
```

Arguments

nrow	An integer describing the number of rows in the layout.
ncol	An integer describing the number of columns in the layout.
widths	A numeric vector or unit object describing the widths of the columns in the layout.
heights	A numeric vector or unit object describing the heights of the rows in the layout.
default.units	A string indicating the default units to use if widths or heights are only given as numeric vectors.
respect	A logical value or a numeric matrix. If a logical, this indicates whether row heights and column widths should respect each other. If a matrix, non-zero values indicate that the corresponding row and column should be respected (see examples below).
just	A string vector indicating how the layout should be justified if it is not the same size as its parent viewport. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible values are: "left", "right", "centre", "center", "bottom", and "top". NOTE that in this context, "left", for example, means align the left edge of the left-most layout column with the left edge of the parent viewport.

Details

The unit objects given for the widths and heights of a layout may use a special units that only has meaning for layouts. This is the "null" unit, which indicates what relative fraction of the available width/height the column/row occupies. See the reference for a better description of relative widths and heights in layouts.

Value

A Grid layout object.

WARNING

This function must NOT be confused with the base R graphics function `layout`. In particular, do not use `layout` in combination with Grid graphics. The documentation for `layout` may provide some useful information and this function should behave identically in comparable situations. The `grid.layout` function has *added* the ability to specify a broader range of units for row heights and column widths, and allows for nested layouts (see `viewport`).

Author(s)

Paul Murrell

References

Murrell, P. R. (1999), Layouts: A Mechanism for Arranging Plots on a Page, *Journal of Computational and Graphical Statistics*, **8**, 121–134.

See Also

Grid, `grid.show.layout`, `viewport`, `layout`

Examples

```
## A variety of layouts (some a bit mid-bending ...)
layout.torture()
## Demonstration of layout justification
grid.newpage()
testlay <- function(just="centre") {
  pushViewport(viewport(layout=grid.layout(1, 1, widths=unit(1, "inches"),
                           heights=unit(0.25, "npc"),
                           just=just)))
  pushViewport(viewport(layout.pos.col=1, layout.pos.row=1))
  grid.rect()
  grid.text(paste(just, collapse="-"))
  popViewport(2)
}
testlay()
testlay(c("left", "top"))
testlay(c("right", "top"))
testlay(c("right", "bottom"))
testlay(c("left", "bottom"))
testlay(c("left"))
testlay(c("right"))
testlay(c("bottom"))
testlay(c("top"))
```

`grid.lines` *Draw Lines in a Grid Viewport*

Description

These functions create and draw a series of lines.

Usage

```
grid.lines(x = unit(c(0, 1), "npc"),
           y = unit(c(0, 1), "npc"),
           default.units = "npc",
           arrow = NULL, name = NULL,
           gp=gpar(), draw = TRUE, vp = NULL)
linesGrob(x = unit(c(0, 1), "npc"),
           y = unit(c(0, 1), "npc"),
           default.units = "npc",
           arrow = NULL, name = NULL,
           gp=gpar(), vp = NULL)
grid.polyline(...)
polylineGrob(x = unit(c(0, 1), "npc"),
             y = unit(c(0, 1), "npc"),
             id=NULL, id.lengths=NULL,
             default.units = "npc",
             arrow = NULL, name = NULL,
             gp=gpar(), vp = NULL)
```

Arguments

x	A numeric vector or unit object specifying x-values.
y	A numeric vector or unit object specifying y-values.
default.units	A string indicating the default units to use if x or y are only given as numeric vectors.
arrow	A list describing arrow heads to place at either end of the line, as produced by the arrow function.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).
id	A numeric vector used to separate locations in x and y into multiple lines. All locations with the same id belong to the same line.
id.lengths	A numeric vector used to separate locations in x and y into multiple lines. Specifies consecutive blocks of locations which make up separate lines.
...	Arguments passed to polylineGrob.

Details

The first two functions create a lines grob (a graphical object describing lines), and grid.lines draws the lines (if draw is TRUE).

The second two functions create or draw a polyline grob, which is just like a lines grob, except that there can be multiple distinct lines drawn.

Value

A lines grob or a polyline grob. grid.lines returns a lines grob invisibly.

Author(s)

Paul Murrell

See Also

Grid, `viewport`, `arrow`

Examples

```
grid.lines()
# Using id (NOTE: locations are not in consecutive blocks)
grid.newpage()
grid.polyline(x=c((0:4)/10, rep(.5, 5), (10:6)/10, rep(.5, 5)),
              y=c(rep(.5, 5), (10:6/10), rep(.5, 5), (0:4)/10),
              id=rep(1:5, 4),
              gp=gpar(col=1:5, lwd=3))
# Using id.lengths
grid.newpage()
grid.polyline(x=outer(c(0, .5, 1, .5), 5:1/5),
              y=outer(c(.5, 1, .5, 0), 5:1/5),
              id.lengths=rep(4, 5),
              gp=gpar(col=1:5, lwd=3))
```

`grid.locator` *Capture a Mouse Click*

Description

Allows the user to click the mouse once within the current graphics device and returns the location of the mouse click within the current viewport, in the specified coordinate system.

Usage

```
grid.locator(unit = "native")
```

Arguments

`unit`	The coordinate system in which to return the location of the mouse click. See the `unit` function for valid coordinate systems.

Details

This function is modal (like the graphics package function `locator`) so the command line and graphics drawing is blocked until the use has clicked the mouse in the current device.

Value

A unit object representing the location of the mouse click within the current viewport, in the specified coordinate system.

If the user did not click mouse button 1, the function (invisibly) returns `NULL`.

Author(s)

Paul Murrell

See Also

`viewport`, `unit`, `locator` in package **graphics**, and for an application see `trellis.focus` and `panel.identify` in package **lattice**.

Examples

```
if (interactive()) {
  ## Need to write a more sophisticated unit as.character method
  unittrim <- function(unit) {
    sub("^([0-9]+|[0-9]+[.][0-9])[0-9]*", "\\1", as.character(unit))
  }
  do.click <- function(unit) {
    click.locn <- grid.locator(unit)
    grid.segments(unit.c(click.locn$x, unit(0, "npc")),
                  unit.c(unit(0, "npc"), click.locn$y),
                  click.locn$x, click.locn$y,
                  gp=gpar(lty="dashed", col="grey"))
    grid.points(click.locn$x, click.locn$y, pch=16, size=unit(1, "mm"))
    clickx <- unittrim(click.locn$x)
    clicky <- unittrim(click.locn$y)
    grid.text(paste("(", clickx, ", ", clicky, ")", sep=""),
              click.locn$x + unit(2, "mm"), click.locn$y,
              just="left")
  }
  do.click("inches")
  pushViewport(viewport(width=0.5, height=0.5,
                        xscale=c(0, 100), yscale=c(0, 10)))
  grid.rect()
  grid.xaxis()
  grid.yaxis()
  do.click("native")
  popViewport()
}
```

---

grid.ls *List the names of grobs or viewports*

Description

Returns a listing of the names of grobs or viewports.

This is a generic function with methods for grobs (including gTrees) and viewports (including vpTrees).

Usage

```
grid.ls(x=NULL, grobs=TRUE, viewports=FALSE, fullNames=FALSE,
        recursive=TRUE, print=TRUE, flatten=TRUE, ...)

nestedListing(x, gindent="  ", vpindent=gindent)
```

```
pathListing(x, gvpSep=" | ", gAlign=TRUE)
grobPathListing(x, ...)
```

Arguments

x	A grob or viewport or NULL. If NULL, the current grid display list is listed. For print functions, this should be the result of a call to grid.ls.
grobs	A logical value indicating whether to list grobs.
viewports	A logical value indicating whether to list viewports.
fullNames	A logical value indicating whether to embellish object names with information about the object type.
recursive	A logical value indicating whether recursive structures should also list their children.
print	A logical indicating whether to print the listing or a function that will print the listing.
flatten	A logical value indicating whether to flatten the listing. Otherwise a more complex hierarchical object is produced.
gindent	The indent used to show nesting in the output for grobs.
vpindent	The indent used to show nesting in the output for viewports.
gvpSep	The string used to separate viewport paths from grob paths.
gAlign	Logical indicating whether to align the left hand edge of all grob paths.
...	Arguments passed to the print function.

Details

If the argument x is NULL, the current contents of the grid display list are listed (both viewports and grobs). In other words, all objects representing the current scene are listed.

Otherwise, x should be a grob or a viewport.

The default behaviour of this function is to print information about the grobs in the current scene. It is also possible to add information about the viewports in the scene. By default, the listing is recursive, so all children of gTrees and all nested viewports are reported.

The format of the information can be controlled via the print argument, which can be given a function to perform the formatting. The nestedListing function produces a line per grob or viewport, with indenting used to show nesting. The pathListing function produces a line per grob or viewport, with viewport paths and grob paths used to show nesting. The grobPathListing is a simple derivation that only shows lines for grobs. The user can define new functions.

Value

The result of this function is either a "gridFlatListing" object (if flatten is TRUE) or a "gridListing" object.

The former is a simple (flat) list of vectors. This is convenient, for example, for working programmatically with the list of grob and viewport names, or for writing a new display function for the listing.

The latter is a more complex heiararchical object (list of lists), but it does contain more detailed information so may be of use for more advanced customisations.

Author(s)

Paul Murrell

See Also

grob viewport

Examples

```
# A gTree, called "parent", with childrenvp vpTree (vp2 within vp1)
# and child grob, called "child", with vp vpPath (down to vp2)
sampleGTree <- gTree(name="parent",
                     children=gList(grob(name="child", vp="vp1::vp2")),
                     childrenvp=vpTree(parent=viewport(name="vp1"),
                                       children=vpList(viewport(name="vp2"))))
grid.ls(sampleGTree)
# Show viewports too
grid.ls(sampleGTree, view=TRUE)
# Only show viewports
grid.ls(sampleGTree, view=TRUE, grob=FALSE)
# Alternate displays
# nested listing, custom indent
grid.ls(sampleGTree, view=TRUE, print=nestedListing, gindent="--")
# path listing
grid.ls(sampleGTree, view=TRUE, print=pathListing)
# path listing, without grobs aligned
grid.ls(sampleGTree, view=TRUE, print=pathListing, gAlign=FALSE)
# grob path listing
grid.ls(sampleGTree, view=TRUE, print=grobPathListing)
# path listing, grobs only
grid.ls(sampleGTree, print=pathListing)
# path listing, viewports only
grid.ls(sampleGTree, view=TRUE, grob=FALSE, print=pathListing)
# raw flat listing
str(grid.ls(sampleGTree, view=TRUE, print=FALSE))
```

---

grid.move.to *Move or Draw to a Specified Position*

Description

Grid has the notion of a current location. These functions sets that location.

Usage

```
grid.move.to(x = 0, y = 0, default.units = "npc", name = NULL,
             draw = TRUE, vp = NULL)

moveToGrob(x = 0, y = 0, default.units = "npc", name = NULL, vp = NULL)

grid.line.to(x = 1, y = 1, default.units = "npc",
             arrow = NULL, name = NULL,
             gp = gpar(), draw = TRUE, vp = NULL)
```

```
lineToGrob(x = 1, y = 1, default.units = "npc", arrow = NULL,
           name = NULL, gp = gpar(), vp = NULL)
```

Arguments

x	A numeric value or a unit object specifying an x-value.
y	A numeric value or a unit object specifying a y-value.
default.units	A string indicating the default units to use if x or y are only given as numeric values.
arrow	A list describing arrow heads to place at either end of the line, as produced by the arrow function.
name	A character identifier.
draw	A logical value indicating whether graphics output should be produced.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).

Details

Both functions create a move.to/line.to grob (a graphical object describing a move-to/line-to), but only grid.move.to/line.to() draws the move.to/line.to (and then only if draw is TRUE).

Value

A move.to/line.to grob. grid.move.to/line.to() returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport, arrow

Examples

```
grid.newpage()
grid.move.to(0.5, 0.5)
grid.line.to(1, 1)
grid.line.to(0.5, 0)
pushViewport(viewport(x=0, y=0, w=0.25, h=0.25, just=c("left", "bottom")))
grid.rect()
grid.grill()
grid.line.to(0.5, 0.5)
popViewport()
```

grid.newpage *Move to a New Page on a Grid Device*

Description

This function erases the current device or moves to a new page.

Usage

```
grid.newpage(recording = TRUE)
```

Arguments

recording	A logical value to indicate whether the new-page operation should be saved onto the Grid display list.

Details

The new page is painted with the fill colour (`gpar("fill")`), which is often transparent. For devices with a *canvas* colour (the on-screen devices `X11`, `windows` and `quartz`), the page is first painted with the canvas colour and then the background colour.

There is a hook called `"grid.newpage"` (see `setHook`) which is used in the testing code to annotate the new page. The hook function(s) are called with no argument. (If the value is a character string, `get` is called on it from within the **grid** name space.)

Value

None.

Author(s)

Paul Murrell

See Also

Grid

grid.null *Null Graphical Object*

Description

These functions create a NULL graphical object, which has zero width, zero height, and draw nothing. It can be used as a place-holder or as an invisible reference point for other drawing.

Usage

```
nullGrob(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
         default.units = "npc",
         name = NULL, vp = NULL)
grid.null(...)
```

Arguments

x	A numeric vector or unit object specifying x-location.
y	A numeric vector or unit object specifying y-location.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
vp	A Grid viewport object (or NULL).
...	Arguments passed to nullGrob().

Value

A null grob.

Author(s)

Paul Murrell

See Also

Grid, viewport

Examples

```
grid.newpage()
grid.null(name="ref")
grid.rect(height=grobHeight("ref"))
grid.segments(0, 0, grobX("ref", 0), grobY("ref", 0))
```

---

grid.pack *Pack an Object within a Frame*

Description

This functions, together with grid.frame and frameGrob are part of a GUI-builder-like interface to constructing graphical images. The idea is that you create a frame with grid.frame or frameGrob then use this functions to pack objects into the frame.

Usage

```
grid.pack(gPath, grob, redraw = TRUE, side = NULL,
          row = NULL, row.before = NULL, row.after = NULL,
          col = NULL, col.before = NULL, col.after = NULL,
          width = NULL, height = NULL,
          force.width = FALSE, force.height = FALSE, border = NULL,
          dynamic = FALSE)

packGrob(frame, grob, side = NULL,
          row = NULL, row.before = NULL, row.after = NULL,
          col = NULL, col.before = NULL, col.after = NULL,
```

```
            width = NULL, height = NULL,
            force.width = FALSE, force.height = FALSE, border = NULL,
            dynamic = FALSE)
```

Arguments

gPath	A gPath object, which specifies a frame on the display list.
frame	An object of class frame, typically the output from a call to grid.frame.
grob	An object of class grob. The object to be packed.
redraw	A boolean indicating whether the output should be updated.
side	One of "left", "top", "right", "bottom" to indicate which side to pack the object on.
row	Which row to add the object to. Must be between 1 and the-number-of-rows-currently-in-the-frame + 1, or NULL in which case the object occupies all rows.
row.before	Add the object to a new row just before this row.
row.after	Add the object to a new row just after this row.
col	Which col to add the object to. Must be between 1 and the-number-of-cols-currently-in-the-frame + 1, or NULL in which case the object occupies all cols.
col.before	Add the object to a new col just before this col.
col.after	Add the object to a new col just after this col.
width	Specifies the width of the column that the object is added to (rather than allowing the width to be taken from the object).
height	Specifies the height of the row that the object is added to (rather than allowing the height to be taken from the object).
force.width	A logical value indicating whether the width of the column that the grob is being packed into should be EITHER the width specified in the call to grid.pack OR the maximum of that width and the pre-existing width.
force.height	A logical value indicating whether the height of the column that the grob is being packed into should be EITHER the height specified in the call to grid.pack OR the maximum of that height and the pre-existing height.
border	A unit object of length 4 indicating the borders around the object.
dynamic	If the width/height is taken from the grob being packed, this boolean flag indicates whether the grobwidth/height unit refers directly to the grob, or uses a gPath to the grob. In the latter case, changes to the grob will trigger a recalculation of the width/height.

Details

packGrob modifies the given frame grob and returns the modified frame grob.

grid.pack destructively modifies a frame grob on the display list (and redraws the display list if redraw is TRUE).

These are (meant to be) very flexible functions. There are many different ways to specify where the new object is to be added relative to the objects already in the frame. The function checks that the specification is not self-contradictory.

NOTE that the width/height of the row/col that the object is added to is taken from the object itself unless the width/height is specified.

Value

packGrob returns a frame grob, but grid.pack returns NULL.

Author(s)

Paul Murrell

See Also

grid.frame, grid.place, grid.edit, and gPath.

---

grid.place *Place an Object within a Frame*

Description

These functions provide a simpler (and faster) alternative to the grid.pack() and packGrob functions. They can be used to place objects within the existing rows and columns of a frame layout. They do not provide the ability to add new rows and columns nor do they affect the heights and widths of the rows and columns.

Usage

```
grid.place(gPath, grob, row = 1, col = 1, redraw = TRUE)
placeGrob(frame, grob, row = NULL, col = NULL)
```

Arguments

gPath	A gPath object, which specifies a frame on the display list.
frame	An object of class frame, typically the output from a call to grid.frame.
grob	An object of class grob. The object to be placed.
row	Which row to add the object to. Must be between 1 and the-number-of-rows-currently-in-the-frame.
col	Which col to add the object to. Must be between 1 and the-number-of-cols-currently-in-the-frame.
redraw	A boolean indicating whether the output should be updated.

Details

placeGrob modifies the given frame grob and returns the modified frame grob.

grid.place destructively modifies a frame grob on the display list (and redraws the display list if redraw is TRUE).

Value

placeGrob returns a frame grob, but grid.place returns NULL.

Author(s)

Paul Murrell

See Also

grid.frame, grid.pack, grid.edit, and gPath.

---

grid.plot.and.legend *A Simple Plot and Legend Demo*

Description

This function is just a wrapper for a simple demonstration of how a basic plot and legend can be drawn from scratch using grid.

Usage

```
grid.plot.and.legend()
```

Author(s)

Paul Murrell

Examples

```
grid.plot.and.legend()
```

---

grid.points *Draw Data Symbols*

Description

These functions create and draw data symbols.

Usage

```
grid.points(x = stats::runif(10),
            y = stats::runif(10),
            pch = 1, size = unit(1, "char"),
            default.units = "native", name = NULL,
            gp = gpar(), draw = TRUE, vp = NULL)
pointsGrob(x = stats::runif(10),
           y = stats::runif(10),
           pch = 1, size = unit(1, "char"),
           default.units = "native", name = NULL,
           gp = gpar(), vp = NULL)
```

Arguments

x	A numeric vector or unit object specifying x-values.
y	A numeric vector or unit object specifying y-values.
pch	A numeric or character vector indicating what sort of plotting symbol to use. See points for the interpretation of these values.
size	A unit object specifying the size of the plotting symbols.
default.units	A string indicating the default units to use if x or y are only given as numeric vectors.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a points grob (a graphical object describing points), but only grid.points draws the points (and then only if draw is TRUE).

Value

A points grob. grid.points returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport

---

grid.polygon *Draw a Polygon*

Description

These functions create and draw a polygon. The final point will automatically be connected to the initial point.

Usage

```
grid.polygon(x=c(0, 0.5, 1, 0.5), y=c(0.5, 1, 0.5, 0),
             id=NULL, id.lengths=NULL,
             default.units="npc", name=NULL,
             gp=gpar(), draw=TRUE, vp=NULL)
polygonGrob(x=c(0, 0.5, 1, 0.5), y=c(0.5, 1, 0.5, 0),
             id=NULL, id.lengths=NULL,
             default.units="npc", name=NULL,
             gp=gpar(), vp=NULL)
```

Arguments

x	A numeric vector or unit object specifying x-locations.
y	A numeric vector or unit object specifying y-locations.
id	A numeric vector used to separate locations in x and y into multiple polygons. All locations with the same id belong to the same polygon.
id.lengths	A numeric vector used to separate locations in x and y into multiple polygons. Specifies consecutive blocks of locations which make up separate polygons.
default.units	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a polygon grob (a graphical object describing a polygon), but only grid.polygon draws the polygon (and then only if draw is TRUE).

Value

A grob object.

Author(s)

Paul Murrell

See Also

Grid, viewport

Examples

```
grid.polygon()
# Using id (NOTE: locations are not in consecutive blocks)
grid.newpage()
grid.polygon(x=c((0:4)/10, rep(.5, 5), (10:6)/10, rep(.5, 5)),
             y=c(rep(.5, 5), (10:6/10), rep(.5, 5), (0:4)/10),
             id=rep(1:5, 4),
             gp=gpar(fill=1:5))
# Using id.lengths
grid.newpage()
grid.polygon(x=outer(c(0, .5, 1, .5), 5:1/5),
             y=outer(c(.5, 1, .5, 0), 5:1/5),
             id.lengths=rep(4, 5),
             gp=gpar(fill=1:5))
```

grid.pretty *Generate a Sensible Set of Breakpoints*

Description

Produces a pretty set of breakpoints within the range given.

Usage

```
grid.pretty(range)
```

Arguments

range	A numeric vector

Value

A numeric vector of breakpoints.

Author(s)

Paul Murrell

grid.prompt *Prompt before New Page*

Description

This function can be used to control whether the user is prompted before starting a new page of output.

Usage

```
grid.prompt(ask)
```

Arguments

ask	a logical value. If TRUE, the user is prompted before a new page of output is started.

Details

As from R 2.7.0 this is deprecated in favour of devAskNewPage as a single setting affects both the base and grid graphics systems.

The default value when a device is opened is taken from the setting of options("device.ask.default").

Value

The current prompt setting *before* any new setting is applied.

Author(s)

Paul Murrell

See Also

`grid.newpage`

`grid.record` *Encapsulate calculations and drawing*

Description

Evaluates an expression that includes both calculations and drawing that depends on the calculations so that both the calculations and the drawing will be rerun when the scene is redrawn (e.g., device resize or editing).

Intended *only* for expert use.

Usage

```
recordGrob(expr, list, name=NULL, gp=NULL, vp=NULL)
grid.record(expr, list, name=NULL, gp=NULL, vp=NULL)
```

Arguments

`expr`	object of mode `expression` or `call` or an unevaluated expression.
`list`	a list defining the environment in which `expr` is to be evaluated.
`name`	A character identifier.
`gp`	An object of class `gpar`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`vp`	A Grid viewport object (or NULL).

Details

A grob is created of special class `"recordedGrob"` (and drawn, in the case of `grid.record`). The `drawDetails` method for this class evaluates the expression with the list as the evaluation environment (and the grid Namespace as the parent of that environment).

Note

This function *must* be used instead of the function `recordGraphics`; all of the dire warnings about using `recordGraphics` responsibly also apply here.

Author(s)

Paul Murrell

See Also

`recordGraphics`

Examples

```
grid.record({
              w <- convertWidth(unit(1, "inches"), "npc")
                      grid.rect(width=w)
            },
            list())
```

`grid.rect` *Draw rectangles*

Description

These functions create and draw rectangles.

Usage

```
grid.rect(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          width = unit(1, "npc"), height = unit(1, "npc"),
          just = "centre", hjust = NULL, vjust = NULL,
          default.units = "npc", name = NULL,
          gp=gpar(), draw = TRUE, vp = NULL)
rectGrob(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
         width = unit(1, "npc"), height = unit(1, "npc"),
         just = "centre", hjust = NULL, vjust = NULL,
         default.units = "npc", name = NULL,
         gp=gpar(), vp = NULL)
```

Arguments

`x`	A numeric vector or unit object specifying x-location.
`y`	A numeric vector or unit object specifying y-location.
`width`	A numeric vector or unit object specifying width.
`height`	A numeric vector or unit object specifying height.
`just`	The justification of the rectangle relative to its (x, y) location. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible string values are: `"left"`, `"right"`, `"centre"`, `"center"`, `"bottom"`, and `"top"`. For numeric values, 0 means left alignment and 1 means right alignment.
`hjust`	A numeric vector specifying horizontal justification. If specified, overrides the `just` setting.
`vjust`	A numeric vector specifying vertical justification. If specified, overrides the `just` setting.
`default.units`	A string indicating the default units to use if `x`, `y`, `width`, or `height` are only given as numeric vectors.
`name`	A character identifier.
`gp`	An object of class `gpar`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`draw`	A logical value indicating whether graphics output should be produced.
`vp`	A Grid viewport object (or NULL).

Details

Both functions create a rect grob (a graphical object describing rectangles), but only `grid.rect` draws the rectangles (and then only if `draw` is `TRUE`).

Value

A rect grob. `grid.rect` returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, `viewport`

`grid.refresh` *Refresh the current grid scene*

Description

Replays the current grid display list.

Usage

```
grid.refresh()
```

Author(s)

Paul Murrell

`grid.remove` *Remove a Grid Graphical Object*

Description

Remove a grob from a gTree or a descendant of a gTree.

Usage

```
grid.remove(gPath, warn = TRUE, strict  =  FALSE, grep = FALSE,
            global = FALSE, allDevices = FALSE, redraw = TRUE)

grid.gremove <- function(..., grep = TRUE, global = TRUE)

removeGrob(gTree, gPath, strict = FALSE, grep = FALSE,
           global = FALSE, warn = TRUE)
```

Arguments

gTree	A gTree object.
gPath	A gPath object. For grid.remove this specifies a gTree on the display list. For removeGrob this specifies a descendant of the specified gTree.
strict	A boolean indicating whether the gPath must be matched exactly.
grep	A boolean indicating whether the gPath should be treated as a regular expression. Values are recycled across elements of the gPath (e.g., c(TRUE, FALSE) means that every odd element of the gPath will be treated as a regular expression).
global	A boolean indicating whether the function should affect just the first match of the gPath, or whether all matches should be affected.
allDevices	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
warn	A logical to indicate whether failing to find the specified grob should trigger an error.
redraw	A logical value to indicate whether to redraw the grob.

Details

removeGrob copies the specified grob and returns a modified grob.

grid.remove destructively modifies a grob on the display list. If redraw is TRUE it then redraws everything to reflect the change.

grid.gremove (g for global) is just a convenience wrapper for grid.remove with different defaults.

Value

removeGrob returns a grob object; grid.remove returns NULL.

Author(s)

Paul Murrell

See Also

grob, getGrob, removeGrob, removeGrob.

---

grid.segments *Draw Line Segments*

Description

These functions create and draw line segments.

Usage

```
grid.segments(x0 = unit(0, "npc"), y0 = unit(0, "npc"),
              x1 = unit(1, "npc"), y1 = unit(1, "npc"),
              default.units = "npc",
              arrow = NULL,
              name = NULL, gp = gpar(), draw = TRUE, vp = NULL)
segmentsGrob(x0 = unit(0, "npc"), y0 = unit(0, "npc"),
              x1 = unit(1, "npc"), y1 = unit(1, "npc"),
              default.units = "npc",
              arrow = NULL, name = NULL, gp = gpar(), vp = NULL)
```

Arguments

x0	Numeric indicating the starting x-values of the line segments.
y0	Numeric indicating the starting y-values of the line segments.
x1	Numeric indicating the stopping x-values of the line segments.
y1	Numeric indicating the stopping y-values of the line segments.
default.units	A string.
arrow	A list describing arrow heads to place at either end of the line segments, as produced by the arrow function.
name	A character identifier.
gp	An object of class gpar.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a segments grob (a graphical object describing segments), but only grid.segments draws the segments (and then only if draw is TRUE).

Value

A segments grob. grid.segments returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, viewport, arrow

grid.set *Set a Grid Graphical Object*

Description

Replace a grob or a descendant of a grob.

Usage

```
grid.set(gPath, newGrob, strict = FALSE, grep = FALSE,
         redraw = TRUE)

setGrob(gTree, gPath, newGrob, strict = FALSE, grep = FALSE)
```

Arguments

gTree	A gTree object.
gPath	A gPath object. For grid.set this specifyies a grob on the display list. For setGrob this specifies a descendant of the specified gTree.
newGrob	A grob object.
strict	A boolean indicating whether the gPath must be matched exactly.
grep	A boolean indicating whether the gPath should be treated as a regular expression. Values are recycled across elements of the gPath (e.g., c(TRUE, FALSE) means that every odd element of the gPath will be treated as a regular expression).
redraw	A logical value to indicate whether to redraw the grob.

Details

setGrob copies the specified grob and returns a modified grob.

grid.set destructively replaces a grob on the display list. If redraw is TRUE it then redraws everything to reflect the change.

These functions should not normally be called by the user.

Value

setGrob returns a grob object; grid.set returns NULL.

Author(s)

Paul Murrell

See Also

grid.grob.

`grid.show.layout` *Draw a Diagram of a Grid Layout*

Description

This function uses Grid graphics to draw a diagram of a Grid layout.

Usage

```
grid.show.layout(l, newpage=TRUE, bg = "light grey",
             cell.border = "blue", cell.fill = "light blue",
             cell.label = TRUE, label.col = "blue",
             unit.col = "red", vp = NULL)
```

Arguments

`l`	A Grid layout object.
`newpage`	A logical value indicating whether to move on to a new page before drawing the diagram.
`bg`	The colour used for the background.
`cell.border`	The colour used to draw the borders of the cells in the layout.
`cell.fill`	The colour used to fill the cells in the layout.
`cell.label`	A logical indicating whether the layout cells should be labelled.
`label.col`	The colour used for layout cell labels.
`unit.col`	The colour used for labelling the widths/heights of columns/rows.
`vp`	A Grid viewport object (or NULL).

Details

A viewport is created within `vp` to provide a margin for annotation, and the layout is drawn within that new viewport. The margin is filled with light grey, the new viewport is filled with white and framed with a black border, and the layout regions are filled with light blue and framed with a blue border. The diagram is annotated with the widths and heights (including units) of the columns and rows of the layout using red text. (All colours are defaults and may be customised via function arguments.)

Value

None.

Author(s)

Paul Murrell

See Also

Grid, `viewport`, `grid.layout`

Examples

```
## Diagram of a simple layout
grid.show.layout(grid.layout(4,2,
                     heights=unit(rep(1, 4),
                                  c("lines", "lines", "lines", "null")),
                     widths=unit(c(1, 1), "inches")))
```

---

grid.show.viewport *Draw a Diagram of a Grid Viewport*

Description

This function uses Grid graphics to draw a diagram of a Grid viewport.

Usage

```
grid.show.viewport(v, parent.layout = NULL, newpage = TRUE,
                   border.fill="light grey",
                   vp.col="blue", vp.fill="light blue",
                   scale.col="red",
                   vp = NULL)
```

Arguments

v	A Grid viewport object.
parent.layout	A grid layout object. If this is not NULL and the viewport given in v has its location specified relative to the layout, then the diagram shows the layout and which cells v occupies within the layout.
newpage	A logical value to indicate whether to move to a new page before drawing the diagram.
border.fill	Colour to fill the border margin.
vp.col	Colour for the border of the viewport region.
vp.fill	Colour to fill the viewport region.
scale.col	Colour to draw the viewport axes.
vp	A Grid viewport object (or NULL).

Details

A viewport is created within vp to provide a margin for annotation, and the diagram is drawn within that new viewport. By default, the margin is filled with light grey, the new viewport is filled with white and framed with a black border, and the viewport region is filled with light blue and framed with a blue border. The diagram is annotated with the width and height (including units) of the viewport, the (x, y) location of the viewport, and the x- and y-scales of the viewport, using red lines and text.

Value

None.

Author(s)

Paul Murrell

See Also

Grid, viewport

Examples

```
## Diagram of a sample viewport
grid.show.viewport(viewport(x=0.6, y=0.6,
                            w=unit(1, "inches"), h=unit(1, "inches")))
grid.show.viewport(viewport(layout.pos.row=2, layout.pos.col=2:3),
                   grid.layout(3, 4))
```

---

grid.text *Draw Text*

Description

These functions create and draw text and plotmath expressions.

Usage

```
grid.text(label, x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          just = "centre", hjust = NULL, vjust = NULL, rot = 0,
          check.overlap = FALSE, default.units = "npc",
          name = NULL, gp = gpar(), draw = TRUE, vp = NULL)

textGrob(label, x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          just = "centre", hjust = NULL, vjust = NULL, rot = 0,
          check.overlap = FALSE, default.units = "npc",
          name = NULL, gp = gpar(), vp = NULL)
```

Arguments

label	A character or expression vector. Other objects are coerced by as.graphicsAnnot.
x	A numeric vector or unit object specifying x-values.
y	A numeric vector or unit object specifying y-values.
just	The justification of the text relative to its (x, y) location. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible string values are: "left", "right", "centre", "center", "bottom", and "top". For numeric values, 0 means left alignment and 1 means right alignment.
hjust	A numeric vector specifying horizontal justification. If specified, overrides the just setting.
vjust	A numeric vector specifying vertical justification. If specified, overrides the just setting.
rot	The angle to rotate the text.

`check.overlap`
: A logical value to indicate whether to check for and omit overlapping text.

`default.units`
: A string indicating the default units to use if `x` or `y` are only given as numeric vectors.

`name`
: A character identifier.

`gp`
: An object of class `gpar`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.

`draw`
: A logical value indicating whether graphics output should be produced.

`vp`
: A Grid viewport object (or NULL).

Details

Both functions create a text grob (a graphical object describing text), but only `grid.text` draws the text (and then only if `draw` is `TRUE`).

If the `label` argument is an expression, the output is formatted as a mathematical annotation, as for base graphics text.

Value

A text grob. `grid.text` returns the value invisibly.

Author(s)

Paul Murrell

See Also

Grid, `viewport`

Examples

```
grid.newpage()
x <- stats::runif(20)
y <- stats::runif(20)
rot <- stats::runif(20, 0, 360)
grid.text("SOMETHING NICE AND BIG", x=x, y=y, rot=rot,
          gp=gpar(fontsize=20, col="grey"))
grid.text("SOMETHING NICE AND BIG", x=x, y=y, rot=rot,
          gp=gpar(fontsize=20), check=TRUE)
grid.newpage()
draw.text <- function(just, i, j) {
  grid.text("ABCD", x=x[j], y=y[i], just=just)
  grid.text(deparse(substitute(just)), x=x[j], y=y[i] + unit(2, "lines"),
            gp=gpar(col="grey", fontsize=8))
}
x <- unit(1:4/5, "npc")
y <- unit(1:4/5, "npc")
grid.grill(h=y, v=x, gp=gpar(col="grey"))
draw.text(c("bottom"), 1, 1)
draw.text(c("left", "bottom"), 2, 1)
draw.text(c("right", "bottom"), 3, 1)
draw.text(c("centre", "bottom"), 4, 1)
draw.text(c("centre"), 1, 2)
```

```
draw.text(c("left", "centre"), 2, 2)
draw.text(c("right", "centre"), 3, 2)
draw.text(c("centre", "centre"), 4, 2)
draw.text(c("top"), 1, 3)
draw.text(c("left", "top"), 2, 3)
draw.text(c("right", "top"), 3, 3)
draw.text(c("centre", "top"), 4, 3)
draw.text(c(), 1, 4)
draw.text(c("left"), 2, 4)
draw.text(c("right"), 3, 4)
draw.text(c("centre"), 4, 4)
```

`grid.xaxis` *Draw an X-Axis*

Description

These functions create and draw an x-axis.

Usage

```
grid.xaxis(at = NULL, label = TRUE, main = TRUE,
           edits = NULL, name = NULL,
           gp = gpar(), draw = TRUE, vp = NULL)

xaxisGrob(at = NULL, label = TRUE, main = TRUE,
          edits = NULL, name = NULL,
          gp = gpar(), vp = NULL)
```

Arguments

`at`	A numeric vector of x-value locations for the tick marks.
`label`	A logical value indicating whether to draw the labels on the tick marks, or an expression or character vector which specify the labels to use. If not logical, must be the same length as the `at` argument.
`main`	A logical value indicating whether to draw the axis at the bottom (`TRUE`) or at the top (`FALSE`) of the viewport.
`edits`	A gEdit or gEditList containing edit operations to apply (to the children of the axis) when the axis is first created and during redrawing whenever `at` is `NULL`.
`name`	A character identifier.
`gp`	An object of class `gpar`, typically the output from a call to the function `gpar`. This is basically a list of graphical parameter settings.
`draw`	A logical value indicating whether graphics output should be produced.
`vp`	A Grid viewport obect (or `NULL`).

Details

Both functions create an xaxis grob (a graphical object describing an xaxis), but only `grid.xaxis` draws the xaxis (and then only if `draw` is `TRUE`).

Value

An xaxis grob. `grid.xaxis` returns the value invisibly.

Children

If the `at` slot of an xaxis grob is not `NULL` then the xaxis will have the following children:

major representing the line at the base of the tick marks.

ticks representing the tick marks.

labels representing the tick labels.

If the `at` slot is `NULL` then there are no children and ticks are drawn based on the current viewport scale.

Author(s)

Paul Murrell

See Also

Grid, `viewport`, `grid.yaxis`

`grid.xspline` *Draw an Xspline*

Description

These functions create and draw an xspline, a curve drawn relative to control points.

Usage

```
grid.xspline(...)
xsplineGrob(x = c(0, 0.5, 1, 0.5), y = c(0.5, 1, 0.5, 0),
            id = NULL, id.lengths = NULL,
            default.units = "npc",
            shape = 0, open = TRUE, arrow = NULL, repEnds = TRUE,
            name = NULL, gp = gpar(), vp = NULL)
```

Arguments

`x`	A numeric vector or unit object specifying x-locations of spline control points.
`y`	A numeric vector or unit object specifying y-locations of spline control points.
`id`	A numeric vector used to separate locations in `x` and `y` into multiple xsplines. All locations with the same `id` belong to the same xspline.
`id.lengths`	A numeric vector used to separate locations in `x` and `y` into multiple xspline. Specifies consecutive blocks of locations which make up separate xsplines.
`default.units`	A string indicating the default units to use if `x` or `y` are only given as numeric vectors.

shape	A numeric vector of values between -1 and 1, which control the shape of the spline relative to the control points.
open	A logical value indicating whether the spline is a line or a closed shape.
arrow	A list describing arrow heads to place at either end of the xspline, as produced by the arrow function.
repEnds	A logical value indicating whether the first and last control points should be replicated for drawing the curve (see Details below).
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	A Grid viewport object (or NULL).
...	Arguments to be passed to xsplineGrob.

Details

Both functions create an xspline grob (a graphical object describing an xspline), but only grid.xspline draws the xspline.

An xspline is a line drawn relative to control points. For each control point, the line may pass through (interpolate) the control point or it may only approach (approximate) the control point; the behaviour is determined by a shape parameter for each control point.

If the shape parameter is greater than zero, the spline approximates the control points (and is very similar to a cubic B-spline when the shape is 1). If the shape parameter is less than zero, the spline interpolates the control points (and is very similar to a Catmull-Rom spline when the shape is -1). If the shape parameter is 0, the spline forms a sharp corner at that control point.

For open xsplines, the start and end control points must have a shape of 0 (and non-zero values are silently converted to zero without warning).

For open xsplines, by default the start and end control points are actually replicated before the curve is drawn. A curve is drawn between (interpolating or approximating) the second and third of each set of four control points, so this default behaviour ensures that the resulting curve starts at the first control point you have specified and ends at the last control point. The default behaviour can be turned off via the repEnds argument, in which case the curve that is drawn starts (approximately) at the second control point and ends (approximately) at the first and second-to-last control point.

The repEnds argument is ignored for closed xsplines.

Missing values are not allowed for x and y (i.e., it is not valid for a control point to be missing).

For closed xsplines, a curve is automatically drawn between the final control point and the initial control point.

Value

A grob object.

References

Blanc, C. and Schlick, C. (1995), "X-splines : A Spline Model Designed for the End User", in *Proceedings of SIGGRAPH 95*, pp. 377–386. http://dept-info.labri.fr/~schlick/DOC/sig1.html

See Also

Grid, viewport, arrow.

xspline.

Examples

```
x <- c(0.25, 0.25, 0.75, 0.75)
y <- c(0.25, 0.75, 0.75, 0.25)

xsplineTest <- function(s, i, j, open) {
  pushViewport(viewport(layout.pos.col=j, layout.pos.row=i))
  grid.points(x, y, default.units="npc", pch=16, size=unit(2, "mm"))
  grid.xspline(x, y, shape=s, open=open, gp=gpar(fill="grey"))
  grid.text(s, gp=gpar(col="grey"),
            x=unit(x, "npc") + unit(c(-1, -1, 1, 1), "mm"),
            y=unit(y, "npc") + unit(c(-1, 1, 1, -1), "mm"),
            hjust=c(1, 1, 0, 0),
            vjust=c(1, 0, 0, 1))
  popViewport()
}

pushViewport(viewport(width=.5, x=0, just="left",
                      layout=grid.layout(3, 3, respect=TRUE)))
pushViewport(viewport(layout.pos.row=1))
grid.text("Open Splines", y=1, just="bottom")
popViewport()
xsplineTest(c(0, -1, -1, 0), 1, 1, TRUE)
xsplineTest(c(0, -1,  0, 0), 1, 2, TRUE)
xsplineTest(c(0, -1,  1, 0), 1, 3, TRUE)
xsplineTest(c(0,  0, -1, 0), 2, 1, TRUE)
xsplineTest(c(0,  0,  0, 0), 2, 2, TRUE)
xsplineTest(c(0,  0,  1, 0), 2, 3, TRUE)
xsplineTest(c(0,  1, -1, 0), 3, 1, TRUE)
xsplineTest(c(0,  1,  0, 0), 3, 2, TRUE)
xsplineTest(c(0,  1,  1, 0), 3, 3, TRUE)
popViewport()
pushViewport(viewport(width=.5, x=1, just="right",
                      layout=grid.layout(3, 3, respect=TRUE)))
pushViewport(viewport(layout.pos.row=1))
grid.text("Closed Splines", y=1, just="bottom")
popViewport()
xsplineTest(c(-1, -1, -1, -1), 1, 1, FALSE)
xsplineTest(c(-1, -1,  0, -1), 1, 2, FALSE)
xsplineTest(c(-1, -1,  1, -1), 1, 3, FALSE)
xsplineTest(c( 0,  0, -1,  0), 2, 1, FALSE)
xsplineTest(c( 0,  0,  0,  0), 2, 2, FALSE)
xsplineTest(c( 0,  0,  1,  0), 2, 3, FALSE)
xsplineTest(c( 1,  1, -1,  1), 3, 1, FALSE)
xsplineTest(c( 1,  1,  0,  1), 3, 2, FALSE)
xsplineTest(c( 1,  1,  1,  1), 3, 3, FALSE)
popViewport()
```

grid.yaxis *Draw a Y-Axis*

Description

These functions create and draw a y-axis.

Usage

```
grid.yaxis(at = NULL, label = TRUE, main = TRUE,
           edits = NULL, name = NULL,
           gp = gpar(), draw = TRUE, vp = NULL)

yaxisGrob(at = NULL, label = TRUE, main = TRUE,
          edits = NULL, name = NULL,
          gp = gpar(), vp = NULL)
```

Arguments

at	A numeric vector of y-value locations for the tick marks.
label	A logical value indicating whether to draw the labels on the tick marks, or an expression or character vector which specify the labels to use. If not logical, must be the same length as the at argument.
main	A logical value indicating whether to draw the axis at the left (TRUE) or at the right (FALSE) of the viewport.
edits	A gEdit or gEditList containing edit operations to apply (to the children of the axis) when the axis is first created and during redrawing whenever at is NULL.
name	A character identifier.
gp	An object of class gpar, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL).

Details

Both functions create a yaxis grob (a graphical object describing a yaxis), but only grid.yaxis draws the yaxis (and then only if draw is TRUE).

Value

A yaxis grob. grid.yaxis returns the value invisibly.

Children

If the at slot of an xaxis grob is not NULL then the xaxis will have the following children:

major representing the line at the base of the tick marks.

ticks representing the tick marks.

labels representing the tick labels.

If the at slot is NULL then there are no children and ticks are drawn based on the current viewport scale.

Author(s)

Paul Murrell

See Also

Grid, `viewport`, `grid.xaxis`

`grobName` *Generate a Name for a Grob*

Description

This function generates a unique (within-session) name for a grob, based on the grob's class.

Usage

```
grobName(grob = NULL, prefix = "GRID")
```

Arguments

`grob`	A grob object or `NULL`.
`prefix`	The prefix part of the name.

Value

A character string of the form `prefix.class(grob).index`

Author(s)

Paul Murrell

`grobWidth` *Create a Unit Describing the Width of a Grob*

Description

These functions create a unit object describing the width or height of a grob. They are generic.

Usage

```
grobWidth(x)
grobHeight(x)
```

Arguments

`x`	A grob object.

Value

A unit object.

Author(s)

Paul Murrell

See Also

`unit` and `stringWidth`

`grobX` *Create a Unit Describing a Grob Boundary Location*

Description

These functions create a unit object describing a location somewhere on the boundary of a grob. They are generic.

Usage

```
grobX(x, theta)
grobY(x, theta)
```

Arguments

`x`	A grob, or gList, or gTree, or gPath.
`theta`	An angle indicating where the location is on the grob boundary. Can be one of `"east"`, `"north"`, `"west"`, or `"south"`, which correspond to angles 0, 90, 180, and 270, respectively.

Details

The angle is anti-clockwise with zero corresponding to a line with an origin centred between the extreme points of the shape, and pointing at 3 o'clock.

If the grob describes a single shape, the boundary value should correspond to the exact edge of the shape.

If the grob describes multiple shapes, the boundary value will either correspond to the edge of a bounding box around all of the shapes described by the grob (for multiple rectangles, circles, xsplines, or text), or to a convex hull around all vertices of all shapes described by the grob (for multiple polygons, points, lines, polylines, and segments).

Points grobs are currently a special case because the convex hull is based on the data symbol *locations* and does not take into account the extent of the data symbols themselves.

The extents of any arrow heads are currently *not* taken into account.

Value

A unit object.

Author(s)

Paul Murrell

See Also

`unit` and `grobWidth`

plotViewport *Create a Viewport with a Standard Plot Layout*

Description

This is a convenience function for producing a viewport with the common S-style plot layout – i.e., a central plot region surrounded by margins given in terms of a number of lines of text.

Usage

```
plotViewport(margins=c(5.1, 4.1, 4.1, 2.1), ...)
```

Arguments

margins	A numeric vector interpreted in the same way as `par(mar)` in base graphics.
...	All other arguments will be passed to a call to the `viewport()` function.

Value

A grid viewport object.

Author(s)

Paul Murrell

See Also

`viewport` and `dataViewport`.

pop.viewport *Pop a Viewport off the Grid Viewport Stack*

Description

Grid maintains a viewport stack — a list of nested drawing contexts.

This function makes the parent of the specified viewport the new default viewport.

Usage

```
pop.viewport(n=1, recording=TRUE)
```

Arguments

n	An integer giving the number of viewports to pop. Defaults to 1.
recording	A logical value to indicate whether the set-viewport operation should be recorded on the Grid display list.

Value

None.

Warning

This function has been deprecated. Please use `popViewport` instead.

Author(s)

Paul Murrell

See Also

`push.viewport`.

`push.viewport` *Push a Viewport onto the Grid Viewport Stack*

Description

Grid maintains a viewport stack — a list of nested drawing contexts.

This function makes the specified viewport the default viewport and makes its parent the previous default viewport (i.e., nests the specified context within the previous default context).

Usage

```
push.viewport(..., recording=TRUE)
```

Arguments

`...`	One or more objects of class `"viewport"`, or `NULL`.
`recording`	A logical value to indicate whether the set-viewport operation should be recorded on the Grid display list.

Value

None.

Warning

This function has been deprecated. Please use `pushViewport` instead.

Author(s)

Paul Murrell

See Also

`pop.viewport`.

`Querying the Viewport Tree`
Get the Current Grid Viewport (Tree)

Description

`current.viewport()` returns the viewport that Grid is going to draw into.

`current.vpTree` returns the entire Grid viewport tree.

`current.vpPath` returns the viewport path to the current viewport.

`current.transform` returns the transformation matrix for the current viewport.

Usage

```
current.viewport(vp=NULL)
current.vpTree(all=TRUE)
current.vpPath()
current.transform()
```

Arguments

`vp`	A Grid viewport object. Use of this argument has been deprecated.
`all`	A logical value indicating whether the entire viewport tree should be returned.

Details

If `all` is `FALSE` then `current.vpTree` only returns the subtree below the current viewport.

Value

A Grid viewport object from `current.viewport` or `current.vpTree`.

`current.transform` returns a 4x4 transformation matrix.

The viewport path returned by `current.vpPath` is `NULL` if the current viewport is the `ROOT` viewport

Author(s)

Paul Murrell

See Also

`viewport`

Examples

```
grid.newpage()
pushViewport(viewport(width=0.8, height=0.8, name="A"))
pushViewport(viewport(x=0.1, width=0.3, height=0.6,
  just="left", name="B"))
upViewport(1)
pushViewport(viewport(x=0.5, width=0.4, height=0.8,
  just="left", name="C"))
```

```
pushViewport(viewport(width=0.8, height=0.8, name="D"))
current.vpPath()
upViewport(1)
current.vpPath()
current.vpTree()
current.viewport()
current.vpTree(all=FALSE)
popViewport(0)
```

`stringWidth` *Create a Unit Describing the Width of a String*

Description

These functions create a unit object describing the width or height of a string.

Usage

```
stringWidth(string)
stringHeight(string)
```

Arguments

`string`	A character vector.

Value

A unit object.

Author(s)

Paul Murrell

See Also

`unit` and `grobWidth`

`unit` *Function to Create a Unit Object*

Description

This function creates a unit object — a vector of unit values. A unit value is typically just a single numeric value with an associated unit.

Usage

```
unit(x, units, data=NULL)
```

Arguments

`x`	A numeric vector.
`units`	A character vector specifying the units for the corresponding numeric values.
`data`	This argument is used to supply extra information for special `unit` types.

Details

Unit objects allow the user to specify locations and dimensions in a large number of different coordinate systems. All drawing occurs relative to a viewport and the `units` specifies what coordinate system to use within that viewport.

Possible `units` (coordinate systems) are:

`"npc"` Normalised Parent Coordinates (the default). The origin of the viewport is (0, 0) and the viewport has a width and height of 1 unit. For example, (0.5, 0.5) is the centre of the viewport.

`"cm"` Centimetres.

`"inches"` Inches. 1 in = 2.54 cm.

`"mm"` Millimetres. 10 mm = 1 cm.

`"points"` Points. 72.27 pt = 1 in.

`"picas"` Picas. 1 pc = 12 pt.

`"bigpts"` Big Points. 72 bp = 1 in.

`"dida"` Dida. 1157 dd = 1238 pt.

`"cicero"` Cicero. 1 cc = 12 dd.

`"scaledpts"` Scaled Points. 65536 sp = 1 pt.

`"lines"` Lines of text. Locations and dimensions are in terms of multiples of the default text size of the viewport (as specified by the viewport's `fontsize` and `lineheight`).

`"char"` Multiples of nominal font height of the viewport (as specified by the viewport's `fontsize`).

`"native"` Locations and dimensions are relative to the viewport's `xscale` and `yscale`.

`"snpc"` Square Normalised Parent Coordinates. Same as Normalised Parent Coordinates, except gives the same answer for horizontal and vertical locations/dimensions. It uses the *lesser* of npc-width and npc-height. This is useful for making things which are a proportion of the viewport, but have to be square (or have a fixed aspect ratio).

`"strwidth"` Multiples of the width of the string specified in the `data` argument. The font size is determined by the pointsize of the viewport.

`"strheight"` Multiples of the height of the string specified in the `data` argument. The font size is determined by the pointsize of the viewport.

`"grobwidth"` Multiples of the width of the grob specified in the `data` argument.

`"grobheight"` Multiples of the height of the grob specified in the `data` argument.

A special `units` value of `"null"` is also allowed, but only makes sense when used in specifying widths of columns or heights of rows in grid layouts (see `grid.layout`).

The `data` argument must be a list when the `unit.length()` is greater than 1. For example, `unit(rep(1, 3), c("npc", "strwidth", "inches"), data=list(NULL, "my string", NULL))`.

It is possible to subset unit objects in the normal way (e.g., `unit(1:5, "npc")[2:4]`), but a special function `unit.c` is provided for combining unit objects.

Certain arithmetic and summary operations are defined for unit objects. In particular, it is possible to add and subtract unit objects (e.g., `unit(1, "npc") - unit(1, "inches")`), and to specify the minimum or maximum of a list of unit objects (e.g., `min(unit(0.5, "npc"), unit(1, "inches"))`).

Value

An object of class `"unit"`.

WARNING

There is a special function `unit.c` for concatenating several unit objects.

The `c` function will not give the right answer.

There used to be `"mylines"`, `"mychar"`, `"mystrwidth"`, `"mystrheight"` units. These will still be accepted, but work exactly the same as `"lines"`, `"char"`, `"strwidth"`, `"strheight"`.

Author(s)

Paul Murrell

See Also

`unit.c`

Examples

```
unit(1, "npc")
unit(1:3/4, "npc")
unit(1:3/4, "npc") + unit(1, "inches")
min(unit(0.5, "npc"), unit(1, "inches"))
unit.c(unit(0.5, "npc"), unit(2, "inches") + unit(1:3/4, "npc"),
       unit(1, "strwidth", "hi there"))
```

`unit.c` *Combine Unit Objects*

Description

This function produces a new unit object by combining the unit objects specified as arguments.

Usage

```
unit.c(...)
```

Arguments

`...`	An arbitrary number of unit objects.

Value

An object of class `unit`.

Author(s)

Paul Murrell

See Also

unit.

`unit.length` *Length of a Unit Object*

Description

The length of a unit object is defined as the number of unit values in the unit object.

This function has been deprecated in favour of a unit method for the generic `length` function.

Usage

```
unit.length(unit)
```

Arguments

`unit`	A unit object.

Value

An integer value.

Author(s)

Paul Murrell

See Also

unit

Examples

```
length(unit(1:3, "npc"))
length(unit(1:3, "npc") + unit(1, "inches"))
length(max(unit(1:3, "npc") + unit(1, "inches")))
length(max(unit(1:3, "npc") + unit(1, "strwidth", "a"))*4)
length(unit(1:3, "npc") + unit(1, "strwidth", "a")*4)
```

unit.pmin *Parallel Unit Minima and Maxima*

Description

Returns a unit object whose i'th value is the minimum (or maximum) of the i'th values of the arguments.

Usage

```
unit.pmin(...)
unit.pmax(...)
```

Arguments

...	One or more unit objects.

Details

The length of the result is the maximum of the lengths of the arguments; shorter arguments are recycled in the usual manner.

Value

A unit object.

Author(s)

Paul Murrell

Examples

```
max(unit(1:3, "cm"), unit(0.5, "npc"))
unit.pmax(unit(1:3, "cm"), unit(0.5, "npc"))
```

unit.rep *Replicate Elements of Unit Objects*

Description

Replicates the units according to the values given in `times` and `length.out`.

This function has been deprecated in favour of a unit method for the generic `rep` function.

Usage

```
unit.rep(x, ...)
```

Arguments

x	An object of class `"unit"`.
...	arguments to be passed to `rep` such as `times` and `length.out`.

Value

An object of class "unit".

Author(s)

Paul Murrell

See Also

rep

Examples

```
rep(unit(1:3, "npc"), 3)
rep(unit(1:3, "npc"), 1:3)
rep(unit(1:3, "npc") + unit(1, "inches"), 3)
rep(max(unit(1:3, "npc") + unit(1, "inches")), 3)
rep(max(unit(1:3, "npc") + unit(1, "strwidth", "a"))*4, 3)
rep(unit(1:3, "npc") + unit(1, "strwidth", "a")*4, 3)
```

---

validDetails *Customising grid grob Validation*

Description

This generic hook function is called whenever a grid grob is created or edited via grob, gTree, grid.edit or editGrob. This provides an opportunity for customising the validation of a new class derived from grob (or gTree).

Usage

```
validDetails(x)
```

Arguments

x	A grid grob.

Details

This function is called by grob, gTree, grid.edit and editGrob. A method should be written for classes derived from grob or gTree to validate the values of slots specific to the new class. (e.g., see grid:::validDetails.axis).

Note that the standard slots for grobs and gTrees are automatically validated (e.g., vp, gp slots for grobs and, in addition, children, and childrenvp slots for gTrees) so only slots specific to a new class need to be addressed.

Value

The function MUST return the validated grob.

Author(s)

Paul Murrell

See Also

grid.edit

---

vpPath *Concatenate Viewport Names*

Description

This function can be used to generate a viewport path for use in downViewport or seekViewport.

A viewport path is a list of nested viewport names.

Usage

```
vpPath(...)
```

Arguments

...	Character values which are viewport names.

Details

Viewport names must only be unique amongst viewports which share the same parent in the viewport tree.

This function can be used to generate a specification for a viewport that includes the viewport's parent's name (and the name of its parent and so on).

For interactive use, it is possible to directly specify a path, but it is strongly recommended that this function is used otherwise in case the path separator is changed in future versions of grid.

Value

A vpPath object.

See Also

viewport, pushViewport, popViewport, downViewport, seekViewport, upViewport

Examples

```
vpPath("vp1", "vp2")
```

widthDetails *Width and Height of a grid grob*

Description

These generic functions are used to determine the size of grid grobs.

Usage

```
widthDetails(x)
heightDetails(x)
```

Arguments

x	A grid grob.

Details

These functions are called in the calculation of "grobwidth" and "grobheight" units. Methods should be written for classes derived from grob or gTree where the size of the grob can be determined (see, for example grid:::widthDetails.frame).

Value

A unit object.

Author(s)

Paul Murrell

See Also

absolute.size.

Working with Viewports *Maintaining and Navigating the Grid Viewport Tree*

Description

Grid maintains a tree of viewports — nested drawing contexts.

These functions provide ways to add or remove viewports and to navigate amongst viewports in the tree.

Usage

```
pushViewport(..., recording=TRUE)
popViewport(n, recording=TRUE)
downViewport(name, strict=FALSE, recording=TRUE)
seekViewport(name, recording=TRUE)
upViewport(n, recording=TRUE)
```

Arguments

`...`	One or more objects of class `"viewport"`.
`n`	An integer value indicating how many viewports to pop or navigate up. The special value `0` indicates to pop or navigate viewports right up to the root viewport.
`name`	A character value to identify a viewport in the tree.
`strict`	A boolean indicating whether the vpPath must be matched exactly.
`recording`	A logical value to indicate whether the viewport operation should be recorded on the Grid display list.

Details

Objects created by the `viewport()` function are only descriptions of a drawing context. A viewport object must be pushed onto the viewport tree before it has any effect on drawing.

The viewport tree always has a single root viewport (created by the system) which corresponds to the entire device (and default graphical parameter settings). Viewports may be added to the tree using `pushViewport()` and removed from the tree using `popViewport()`.

There is only ever one current viewport, which is the current position within the viewport tree. All drawing and viewport operations are relative to the current viewport. When a viewport is pushed it becomes the current viewport. When a viewport is popped, the parent viewport becomes the current viewport. Use `upViewport` to navigate to the parent of the current viewport, without removing the current viewport from the viewport tree. Use `downViewport` to navigate to a viewport further down the viewport tree and `seekViewport` to navigate to a viewport anywhere else in the tree.

If a viewport is pushed and it has the same `name` as a viewport at the same level in the tree, then it replaces the existing viewport in the tree.

Value

`downViewport` returns the number of viewports it went down.

This can be useful for returning to your starting point by doing something like `depth <- downViewport()` then `upViewport(depth)`.

Author(s)

Paul Murrell

See Also

`viewport` and `vpPath`.

Examples

```
# push the same viewport several times
grid.newpage()
vp <- viewport(width=0.5, height=0.5)
pushViewport(vp)
grid.rect(gp=gpar(col="blue"))
grid.text("Quarter of the device",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="blue"))
pushViewport(vp)
grid.rect(gp=gpar(col="red"))
grid.text("Quarter of the parent viewport",
```

```
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="red"))
popViewport(2)
# push several viewports then navigate amongst them
grid.newpage()
grid.rect(gp=gpar(col="grey"))
grid.text("Top-level viewport",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="grey"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(width=0.8, height=0.7, name="A"))
grid.rect(gp=gpar(col="blue"))
grid.text("1. Push Viewport A",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="blue"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(x=0.1, width=0.3, height=0.6,
  just="left", name="B"))
grid.rect(gp=gpar(col="red"))
grid.text("2. Push Viewport B (in A)",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="red"))
if (interactive()) Sys.sleep(1.0)
upViewport(1)
grid.text("3. Up from B to A",
  y=unit(1, "npc") - unit(2, "lines"), gp=gpar(col="blue"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(x=0.5, width=0.4, height=0.8,
  just="left", name="C"))
grid.rect(gp=gpar(col="green"))
grid.text("4. Push Viewport C (in A)",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="green"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(width=0.8, height=0.6, name="D"))
grid.rect()
grid.text("5. Push Viewport D (in C)",
  y=unit(1, "npc") - unit(1, "lines"))
if (interactive()) Sys.sleep(1.0)
upViewport(0)
grid.text("6. Up from D to top-level",
  y=unit(1, "npc") - unit(2, "lines"), gp=gpar(col="grey"))
if (interactive()) Sys.sleep(1.0)
downViewport("D")
grid.text("7. Down from top-level to D",
  y=unit(1, "npc") - unit(2, "lines"))
if (interactive()) Sys.sleep(1.0)
seekViewport("B")
grid.text("8. Seek from D to B",
  y=unit(1, "npc") - unit(2, "lines"), gp=gpar(col="red"))
pushViewport(viewport(width=0.9, height=0.5, name="A"))
grid.rect()
grid.text("9. Push Viewport A (in B)",
  y=unit(1, "npc") - unit(1, "lines"))
if (interactive()) Sys.sleep(1.0)
seekViewport("A")
grid.text("10. Seek from B to A (in ROOT)",
  y=unit(1, "npc") - unit(3, "lines"), gp=gpar(col="blue"))
if (interactive()) Sys.sleep(1.0)
seekViewport(vpPath("B", "A"))
grid.text("11. Seek from\nA (in ROOT)\nto A (in B)")
popViewport(0)
```

`xDetails` *Boundary of a grid grob*

Description

These generic functions are used to determine a location on the boundary of a grid grob.

Usage

```
xDetails(x, theta)
yDetails(x, theta)
```

Arguments

`x`	A grid grob.
`theta`	A numeric angle, in degrees, measured anti-clockwise from the 3 o'clock *or* one of the following character strings: `"north"`, `"east"`, `"west"`, `"south"`.

Details

The location on the grob boundary is determined by taking a line from the centre of the grob at the angle `theta` and intersecting it with the convex hull of the grob (for the basic grob primitives, the centre is determined as half way between the minimum and maximum values in x and y directions).

These functions are called in the calculation of `"grobx"` and `"groby"` units as produced by the `grobX` and `grobY` functions. Methods should be written for classes derived from grob or gTree where the boundary of the grob can be determined.

Value

A unit object.

Author(s)

Paul Murrell

See Also

`grobX`, `grobY`.

Chapter 6

The `methods` package

`methods-package` *Formal Methods and Classes*

Description

Formally defined methods and classes for R objects, plus other programming tools, as described in the references.

Details

This package provides the 'S4' or 'S version 4' approach to methods and classes in a functional language.

See the documentation entries `Classes`, `Methods`, and `GenericFunctions` for general discussion of these topics, at a fairly technical level. Links from those pages, and the documentation of `setClass` and `setMethod` cover the main programming tools needed.

For a complete list of functions and classes, use `library(help="methods")`.

Author(s)

R Development Core Team

Maintainer: R Core Team ⟨R-core@r-project.org⟩

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

.BasicFunsList *List of Builtin and Special Functions*

Description

A named list providing instructions for turning builtin and special functions into generic functions.

Functions in R that are defined as `.Primitive(<name>)` are not suitable for formal methods, because they lack the basic reflectance property. You can't find the argument list for these functions by examining the function object itself.

Future versions of R may fix this by attaching a formal argument list to the corresponding function. While generally the names of arguments are not checked by the internal code implementing the function, the number of arguments frequently is.

In any case, some definition of a formal argument list is needed if users are to define methods for these functions. In particular, if methods are to be merged from multiple packages, the different sets of methods need to agree on the formal arguments.

In the absence of reflectance, this list provides the relevant information via a dummy function associated with each of the known specials for which methods are allowed.

At the same, the list flags those specials for which methods are meaningless (e.g., `for`) or just a very bad idea (e.g., `.Primitive`).

A generic function created via `setMethod`, for example, for one of these special functions will have the argument list from `.BasicFunsList`. If no entry exists, the argument list `(x, ...)` is assumed.

as *Force an Object to Belong to a Class*

Description

These functions manage the relations that allow coercing an object to a given class.

Usage

```
as(object, Class, strict=TRUE, ext)

as(object, Class) <- value

setAs(from, to, def, replace, where = topenv(parent.frame()))
```

Arguments

`object`	any R object.
`Class`	the name of the class to which `object` should be coerced.

`strict` logical flag. If `TRUE`, the returned object must be strictly from the target class (unless that class is a virtual class, in which case the object will be from the closest actual class, in particular the original object, if that class extends the virtual class directly).

If `strict = FALSE`, any simple extension of the target class will be returned, without further change. A simple extension is, roughly, one that just adds slots to an existing class.

`value` The value to use to modify `object` (see the discussion below). You should supply an object with class `Class`; some coercion is done, but you're unwise to rely on it.

`from, to` The classes between which the coerce methods `def` and `replace` perform coercion.

`def` function of one argument. It will get an object from class `from` and had better return an object of class `to`. The convention is that the name of the argument is `from`; if another argument name is used, `setAs` will attempt to substitute `from`.

`replace` if supplied, the function to use as a replacement method, when `as` is used on the left of an assignment. Should be a function of two arguments, `from, value`, although `setAs` will attempt to substitute if the arguments differ.

`where` the position or environment in which to store the resulting methods. For most applications, it is recommended to omit this argument and to include the call to `setAs` in source code that is evaluated at the top level; that is, either in an R session by something equivalent to a call to `source`, or as part of the R source code for a package.

`ext` the optional object defining how `Class` is extended by the class of the object (as returned by `possibleExtends`). This argument is used internally (to provide essential information for non-public classes), but you are unlikely to want to use it directly.

Summary of Functions

`as`: Returns the version of this object coerced to be the given `Class`. When used in the replacement form on the left of an assignment, the portion of the object corresponding to `Class` is replaced by `value`.

The operation of `as()` in either form depends on the definition of coerce methods. Methods are defined automatically when the two classes are related by inheritance; that is, when one of the classes is a subclass of the other. See the section on inheritance below for details.

Coerce methods are also predefined for basic classes (including all the types of vectors, functions and a few others). See `showMethods(coerce)` for a list of these.

Beyond these two sources of methods, further methods are defined by calls to the `setAs` function.

`setAs`: Define methods for coercing an object of class `from` to be of class `to`; the `def` argument provides for direct coercing and the `replace` argument, if included, provides for replacement. See the "How" section below for details.

`coerce`, `coerce<-`: Coerce `from` to be of the same class as `to`.

These functions should not be called explicitly. The function `setAs` creates methods for them for the `as` function to use.

Inheritance and Coercion

Objects from one class can turn into objects from another class either automatically or by an explicit call to the `as` function. Automatic conversion is special, and comes from the designer of one class of objects asserting that this class extends another class. The most common case is that one or more class names are supplied in the `contains=` argument to `setClass`, in which case the new class extends each of the earlier classes (in the usual terminology, the earlier classes are *superclasses* of the new class and it is a *subclass* of each of them).

This form of inheritance is called *simple* inheritance in R. See `setClass` for details. Inheritance can also be defined explicitly by a call to `setIs`. The two versions have slightly different implications for coerce methods. Simple inheritance implies that inherited slots behave identically in the subclass and the superclass. Whenever two classes are related by simple inheritance, corresponding coerce methods are defined for both direct and replacement use of `as`. In the case of simple inheritance, these methods do the obvious computation: they extract or replace the slots in the object that correspond to those in the superclass definition.

The implicitly defined coerce methods may be overridden by a call to `setAs`; note, however, that the implicit methods are defined for each subclass-superclass pair, so that you must override each of these explicitly, not rely on inheritance.

When inheritance is defined by a call to `setIs`, the coerce methods are provided explicitly, not generated automatically. Inheritance will apply (to the `from` argument, as described in the section below). You could also supply methods via `setAs` for non-inherited relationships, and now these also can be inherited.

For further on the distinction between simple and explicit inheritance, see `setIs`.

How Functions 'as' and 'setAs' Work

The function `as` turns `object` into an object of class `Class`. In doing so, it applies a "coerce method", using S4 classes and methods, but in a somewhat special way. Coerce methods are methods for the function `coerce` or, in the replacement case the function `` `coerce<-` ``. These functions have two arguments in method signatures, `from` and `to`, corresponding to the class of the object and the desired coerce class. These functions must not be called directly, but are used to store tables of methods for the use of `as`, directly and for replacements. In this section we will describe the direct case, but except where noted the replacement case works the same way, using `` `coerce<-` `` and the `replace` argument to `setAs`, rather than `coerce` and the `def` argument.

Assuming the `object` is not already of the desired class, `as` first looks for a method in the table of methods for the function `coerce` for the signature `c(from = class(object), to = Class)`, in the same way method selection would do its initial lookup. To be precise, this means the table of both direct and inherited methods, but inheritance is used specially in this case (see below).

If no method is found, `as` looks for one. First, if either `Class` or `class(object)` is a superclass of the other, the class definition will contain the information needed to construct a coerce method. In the usual case that the subclass contains the superclass (i.e., has all its slots), the method is constructed either by extracting or replacing the inherited slots. Non-simple extensions (the result of a call to `setIs`) will usually contain explicit methods, though possibly not for replacement.

If no subclass/superclass relationship provides a method, `as` looks for an inherited method, but applying, inheritance for the argument `from` only, not for the argument `to` (if you think about it, you'll probably agree that you wouldn't want the result to be from some class other than the `Class` specified). Thus, `selectMethod("coerce", sig, useInherited= c(from=TRUE, to= FALSE))` replicates the method selection used by `as()`.

In nearly all cases the method found in this way will be cached in the table of coerce methods (the exception being subclass relationships with a test, which are legal but discouraged). So the detailed

calculations should be done only on the first occurrence of a coerce from `class(object)` to `Class`.

With this explanation as background, the function `setAs` does a fairly obvious computation: It constructs and sets a method for the function `coerce` with signature `c(from, to)`, using the `def` argument to define the body of the method. The function supplied as `def` can have one argument (interpreted as an object to be coerced) or two arguments (the `from` object and the `to` class). Either way, `setAs` constructs a function of two arguments, with the second defaulting to the name of the `to` class. The method will be called from `as` with the object as the `from` argument and no `to` argument, with the default for this argument being the name of the intended `to` class, so the method can use this information in messages.

The direct version of the `as` function also has a `strict=` argument that defaults to `TRUE`. Calls during the evaluation of methods for other functions will set this argument to `FALSE`. The distinction is relevant when the object being coerced is from a simple subclass of the `to` class; if `strict=FALSE` in this case, nothing need be done. For most user-written coerce methods, when the two classes have no subclass/superclass, the `strict=` argument is irrelevant.

The `replace` argument to `setAs` provides a method for `'coerce<-'`. As with all replacement methods, the last argument of the method must have the name `value` for the object on the right of the assignment. As with the `coerce` method, the first two arguments are `from, to`; there is no `strict=` option for the replace case.

The function `coerce` exists as a repository for such methods, to be selected as described above by the `as` function. Actually dispatching the methods using `standardGeneric` could produce incorrect inherited methods, by using inheritance on the `to` argument; as mentioned, this is not the logic used for `as`. To prevent selecting and caching invalid methods, calls to `coerce` are currently mapped into calls to `as`, with a warning message.

Basic Coercion Methods

Methods are pre-defined for coercing any object to one of the basic datatypes. For example, `as(x, "numeric")` uses the existing `as.numeric` function. These built-in methods can be listed by `showMethods("coerce")`.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

If you think of using `try(as(x, cl))`, consider `canCoerce(x, cl)` instead.

Examples

```
## using the definition of class "track" from Classes


setAs("track", "numeric", function(from) from@y)

t1 <- new("track", x=1:20, y=(1:20)^2)

as(t1, "numeric")
```

```
## The next example shows:
##  1. A virtual class to define setAs for several classes at once.
##  2. as() using inherited information

setClass("ca", representation(a = "character", id = "numeric"))

setClass("cb", representation(b = "character", id = "numeric"))

setClass("id")
setIs("ca", "id")
setIs("cb", "id")

setAs("id", "numeric", function(from) from@id)

CA <- new("ca", a = "A", id = 1)
CB <- new("cb", b = "B", id = 2)

setAs("cb", "ca", function(from, to )new(to, a=from@b, id = from@id))

as(CB, "numeric")
```

BasicClasses *Classes Corresponding to Basic Data Types*

Description

Formal classes exist corresponding to the basic R object types, allowing these types to be used in method signatures, as slots in class definitions, and to be extended by new classes.

Usage

```
### The following are all basic vector classes.
### They can appear as class names in method signatures,
### in calls to as(), is(), and new().
"character"
"complex"
"double"
"expression"
"integer"
"list"
"logical"
"numeric"
"single"
"raw"

### the class
"vector"
### is a virtual class, extended by all the above

### the class
"S4"
```

```
### is an object type for S4 objects that do not extend
### any of the basic vector classes.  It is a virtual class.

### The following are additional basic classes
"NULL"     #  NULL objects
"function" #  function objects, including primitives
"externalptr" # raw external pointers for use in C code

"ANY"  # virtual classes used by the methods package itself
"VIRTUAL"
"missing"
```

Objects from the Classes

Objects can be created by calls of the form `new(Class, ...)`, where `Class` is the quoted class name, and the remaining arguments if any are objects to be interpreted as vectors of this class. Multiple arguments will be concatenated.

The class `"expression"` is slightly odd, in that the ... arguments will *not* be evaluated; therefore, don't enclose them in a call to `quote()`.

Extends

Class `"vector"`, directly.

Methods

coerce Methods are defined to coerce arbitrary objects to these classes, by calling the corresponding basic function, for example, `as(x, "numeric")` calls `as.numeric(x)`.

`callGeneric` *Call the Current Generic Function from a Method*

Description

A call to `callGeneric` can only appear inside a method definition. It then results in a call to the current generic function. The value of that call is the value of `callGeneric`. While it can be called from any method, it is useful and typically used in methods for group generic functions.

Usage

```
callGeneric(...)
```

Arguments

`...`	Optionally, the arguments to the function in its next call. If no arguments are included in the call to `callGeneric`, the effect is to call the function with the current arguments. See the detailed description for what this really means.

Details

The name and package of the current generic function is stored in the environment of the method definition object. This name is looked up and the corresponding function called.

The statement that passing no arguments to `callGeneric` causes the generic function to be called with the current arguments is more precisely as follows. Arguments that were missing in the current call are still missing (remember that `"missing"` is a valid class in a method signature). For a formal argument, say `x`, that appears in the original call, there is a corresponding argument in the generated call equivalent to `x = x`. In effect, this means that the generic function sees the same actual arguments, but arguments are evaluated only once.

Using `callGeneric` with no arguments is prone to creating infinite recursion, unless one of the arguments in the signature has been modified in the current method so that a different method is selected.

Value

The value returned by the new call.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`GroupGenericFunctions` for other information about group generic functions; Methods for the general behavior of method dispatch

Examples

```
## the method for group generic function Ops
## for signature( e1="structure", e2="vector")
function (e1, e2)
{
    value <- callGeneric(e1@.Data, e2)
    if (length(value) == length(e1)) {
        e1@.Data <- value
        e1
    }
    else value
}

## For more examples
## Not run:
showMethods("Ops", includeDefs = TRUE)
## End(Not run)
```

`callNextMethod` *Call an Inherited Method*

Description

A call to `callNextMethod` can only appear inside a method definition. It then results in a call to the first inherited method after the current method, with the arguments to the current method passed down to the next method. The value of that method call is the value of `callNextMethod`.

Usage

```
callNextMethod(...)
```

Arguments

`...`	Optionally, the arguments to the function in its next call (but note that the dispatch is as in the detailed description below; the arguments have no effect on selecting the next method.) If no arguments are included in the call to `callNextMethod`, the effect is to call the method with the current arguments. See the detailed description for what this really means. Calling with no arguments is often the natural way to use `callNextMethod`; see the examples.

Details

The 'next' method (i.e., the first inherited method) is defined to be that method which *would* have been called if the current method did not exist. This is more-or-less literally what happens: The current method (to be precise, the method with signature given by the `defined` slot of the method from which `callNextMethod` is called) is deleted from a copy of the methods for the current generic, and `selectMethod` is called to find the next method (the result is cached in a special object, so the search only typically happens once per session per combination of argument classes).

Note that the preceding definition means that the next method is defined uniquely when `setMethod` inserts the method containing the `callNextMethod` call, given the definitions of the classes in the signature. The choice does not depend on the path that gets us to that method (for example, through inheritance or from another `callNextMethod` call). This definition was not enforced in versions of R prior to 2.3.0, where the method was selected based on the target signature, and so could vary depending on the actual arguments.

It is also legal, and often useful, for the method called by `callNextMethod` to itself have a call to `callNextMethod`. This generally works as you would expect, but for completeness be aware that it is possible to have ambiguous inheritance in the S structure, in the sense that the same two classes can appear as superclasses *in the opposite order* in two other class definitions. In this case the effect of a nested instance of `callNextMethod` is not well defined. Such inconsistent class hierarchies are both rare and nearly always the result of bad design, but they are possible, and currently undetected.

The statement that the method is called with the current arguments is more precisely as follows. Arguments that were missing in the current call are still missing (remember that `"missing"` is a valid class in a method signature). For a formal argument, say `x`, that appears in the original call, there is a corresponding argument in the next method call equivalent to `x = x`. In effect, this means that the next method sees the same actual arguments, but arguments are evaluated only once.

Value

The value returned by the selected method.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`callGeneric` to call the generic function with the current dispatch rules (typically for a group generic function); Methods for the general behavior of method dispatch

Examples

```
## some class definitions with simple inheritance
setClass("B0" , representation(b0 = "numeric"))

setClass("B1", representation(b1 = "character"), contains = "B0")

setClass("B2", representation(b2 = "logical"), contains = "B1")

## and a rather silly function to illustrate callNextMethod

f <- function(x) class(x)

setMethod("f", "B0", function(x) c(x@b0^2, callNextMethod()))
setMethod("f", "B1", function(x) c(paste(x@b1,":"), callNextMethod()))
setMethod("f", "B2", function(x) c(x@b2, callNextMethod()))

b1 <- new("B1", b0 = 2, b1 = "Testing")

b2 <- new("B2", b2 = FALSE, b1 = "More testing", b0 = 10)

f(b2)
stopifnot(identical(f(b2), c(b2@b2, paste(b2@b1,":"), b2@b0^2, "B2")))

f(b1)

## a sneakier method: the *changed* x is used:
setMethod("f", "B2", function(x) {x@b0 <- 111; c(x@b2, callNextMethod())})
f(b2)
stopifnot(identical(f(b2), c(b2@b2, paste(b2@b1,":"), 111^2, "B2")))
```

canCoerce *Can an Object be Coerced to a Certain S4 Class?*

Description

Test if an object can be coerced to a given S4 class. Maybe useful inside `if()` to ensure that calling `as(object, Class)` will find a method.

Usage

```
canCoerce(object, Class)
```

Arguments

`object`	any R object, typically of a formal S4 class.
`Class`	an S4 class (see `isClass`).

Value

a scalar logical, `TRUE` if there is a `coerce` method (as defined by `setAs`, e.g.) for the signature `(from = class(object), to = Class)`.

See Also

`as`, `setAs`, `selectMethod`, `setClass`,

Examples

```
m <- matrix(pi, 2,3)
canCoerce(m, "numeric") # TRUE
canCoerce(m, "array")   # TRUE
```

cbind2 *Combine two Objects by Columns or Rows*

Description

Combine two matrix-like R objects by columns (`cbind2`) or rows (`rbind2`). These are (S4) generic functions with default methods.

Usage

```
cbind2(x, y)
rbind2(x, y)
```

Arguments

`x`	any R object, typically matrix-like.
`y`	any R object, typically similar to `x`, or missing completely.

Details

The main use of `cbind2` (`rbind2`) is to be called by `cbind()` (`rbind()`) **if** these are activated. This allows `cbind` (`rbind`) to work for formally classed (aka 'S4') objects by providing S4 methods for these objects. Currently, a call
`methods:::bind_activation(TRUE)`
is needed to install a `cbind2`-calling version of `cbind` (into the `base` name space) and the same for `rbind`.
`methods:::bind_activation(FALSE)` reverts to the previous internal version of `cbind` which does not build on `cbind2`, see the examples.

Value

A matrix (or matrix like object) combining the columns (or rows) of `x` and `y`.

Methods

x = "ANY", y = "ANY" the default method using R's internal code.

x = "ANY", y = "missing" the default method for one argument using R's internal code.

See Also

`cbind`, `rbind`.

Examples

```
cbind2(1:3, 4)
m <- matrix(3:8, 2,3, dimnames=list(c("a","b"), LETTERS[1:3]))
cbind2(1:2, m) # keeps dimnames from m

### Note: Use the following activation if you want cbind() to work
### ----  on S4 objects -- be careful otherwise!

methods:::bind_activation(on = TRUE)
trace("cbind2")
cbind(a=1:3)# no call to cbind2()
cbind(a=1:3, four=4, 7:9)# calling cbind2() twice
untrace("cbind2")



## The following fails currently,
## since cbind() works recursively from the tail:
try( cbind(m, a=1, b=3) )

## turn off the `special cbind()' :
methods:::bind_activation(FALSE)
```

`Classes` *Class Definitions*

Description

Class definitions are objects that contain the formal definition of a class of R objects, usually referred to as an S4 class, to distinguish them from the informal S3 classes. This document gives an overview of S4 classes; for details of the class representation objects, see help for the class `classRepresentation`.

Metadata Information

When a class is defined, an object is stored that contains the information about that class. The object, known as the *metadata* defining the class, is not stored under the name of the class (to allow programmers to write generating functions of that name), but under a specially constructed name. To examine the class definition, call `getClass`. The information in the metadata object includes:

Slots: The data contained in an object from an S4 class is defined by the *slots* in the class definition.

Each slot in an object is a component of the object; like components (that is, elements) of a list, these may be extracted and set, using the function `slot()` or more often the operator `"@"`. However, they differ from list components in important ways. First, slots can only be referred to by name, not by position, and there is no partial matching of names as with list elements.

All the objects from a particular class have the same set of slot names; specifically, the slot names that are contained in the class definition. Each slot in each object always is an object of the class specified for this slot in the definition of the current class. The word "is" corresponds to the R function of the same name (`is`), meaning that the class of the object in the slot must be the same as the class specified in the definition, or some class that extends the one in the definition (a *subclass*).

One class name is special, `.Data`. This stands for the 'data part' of the object. Any class that contains one of the basic object types in R, has implicitly a corresponding `.Data` slot of that type, allowing computations to extract or replace the data part while leaving other slots unchanged. The `.Data` slot also determines the type of the object; if `x` has a `.Data` slot, the type of the slot is the type of the object (that is, the value of `typeof(x)`). Extending a basic type this way allows objects to use old-style code for the corresponding type as well as S4 methods. Any basic type can be used for `.Data`, with the exception of a few that do not behave like ordinary objects; namely, `"NULL"`, environments, and external pointers.

There is one additional use of the data part, which is also an exception to the correspondence with the object's type. The exception arises from the special treatment of `matrix` and `array` "classes" in R. Matrix and array objects are managed internally and recognized without regard to any class attribute; therefore, they can be used as the data part of a new class. In this case, the object type for the new class depends on the type of the data in the matrix or array.

If the new class does not have a data part as described above, the type of objects from the new class is `"S4"`.

Superclasses: The definition of a class includes the *superclasses* —the classes that this class extends. A class `Fancy`, say, extends a class `Simple` if an object from the `Fancy` class has all the capabilities of the `Simple` class (and probably some more as well). In particular, and very usefully, any method defined to work for a `Simple` object can be applied to a `Fancy` object as well.

This relationship is expressed equivalently by saying that `Simple` is a superclass of `Fancy`, or that `Fancy` is a subclass of `Simple`.

The direct superclasses of a class are those superclasses explicitly defined. Direct superclasses can be defined in three ways. Most commonly, the superclasses are listed in the `contains=` argument in the call to `setClass` that creates the subclass. In this case the subclass will contain all the slots of the superclass, and the relation between the class is called *simple*, as it in fact is. Superclasses can also be defined explicitly by a call to `setIs`; in this case, the relation requires methods to be specified to go from subclass to superclass. Thirdly, a class union is a superclass of all the members of the union. In this case too the relation is simple, but notice that the relation is defined when the superclass is created, not when the subclass is created as with the `contains=` mechanism.

The definition of a superclass will also potentially contain its own direct superclasses. These are considered (and shown) as superclasses at distance 2 from the original class; their direct superclasses are at distance 3, and so on. All these are legitimate superclasses for purposes such as method selection.

When superclasses are defined by including the names of superclasses in the `contains=` argument to `setClass`, an object from the class will have all the slots defined for its own class *and* all the slots defined for all its superclasses as well.

The information about the relation between a class and a particular superclass is encoded as an object of class `SClassExtension`. A list of such objects for the superclasses (and sometimes for the subclasses) is included in the metadata object defining the class. If you need to compute with these objects (for example, to compare the distances), call the function `extends` with argument `fullInfo=TRUE`.

Prototype: The objects from a class, typically created by a call to `new` or by assigning another object from the class, are defined by the *prototype* object for the class and by additional arguments in the call to `new`, which are passed to a method for that class for the function `initialize`.

Each class definition contains a prototype object for the class. This must have values for all the slots defined by the class definition. By default, these are the prototypes of all the slot classes, if those are not virtual classes. However, the definition of the class can specify any valid object for any of the slots.

Virtual classes; Basic classes

Classes exist for which there are no actual objects, the *virtual* classes, in fact a very important programming tool. They are used to group together ordinary classes that want to share some programming behavior, without necessarily restricting how the behavior is implemented. Virtual class definitions may if you want include slots (to provide some common behavior without fully defining the object—see the class `traceable` for an example).

A simple and useful form of virtual class is the *class union*, a virtual class that is defined in a call to `setClassUnion` by listing one or more of subclasses (classes that extend the class union). Class unions can include as subclasses basic data types (whose definition is otherwise sealed).

There are a number of 'basic' classes, corresponding to the ordinary kinds of data occurring in R. For example, `"numeric"` is a class corresponding to numeric vectors. The other vector basic classes are `"logical"`, `"integer"`, `"complex"`, `"character"`, `"raw"`, `"list"` and `"expression"`. The prototypes for the vector classes are vectors of length 0 of the corresponding type. Notice that basic classes are unusual in that the prototype object is from the class itself.

In addition to the vector classes there are also basic classes corresponding to objects in the language, such as `"function"` and `"call"`. These classes are subclasses of the virtual class `"language"`. Finally, there are basic classes for specialized objects, such as `"environment"`

and `"externalptr"`. The vector and language basic classes can be used as slots or as superclasses for any other class definitions. The classes corresponding to other object types can be used as slots but not always as superclasses, since many of them do not follow the functional behavior of the language; in particular, they are not copied and so cannot have attributes or slots defined locally.

S3 Classes

Earlier, informal classes of objects (usually referred to as 'S3' classes) are used by many R functions. It is natural to consider including them as the class for a slot in a formal class, or even as a class to be extended by the new class. This isn't prohibited but there are some disadvantages, and if you do want to include S3 classes, they should be declared by including them in a call to `setOldClass`. Here are some considerations:

- Using S3 classes somewhat defeats the purpose of defining a formal class: An important advantage to your users is that a formal class provides guarantees of what the object contains (minimally, the classes of the slots and therfore what data they contain; optionally, any other requirements imposed by a validity method).

  But there is no guarantee whatever about the data in an object from an S3 class. It's entirely up to the functions that create or modify such objects. If you want to provide guarantees to your users, you will need a valdity method that explicitly checks the contents of S3-class objects.
- To get the minimal guarantee (that the object in a slot has, or extends, the class for the slot) you should ensure that the S3 classes are known to *be* S3 classes, possibly with S inheritance. To do this, include a call to `setOldClass` for the S3 classes used.

  Otherwise, the S3 class is undefined (and the code used by `setClass` will issue a warning). Slot assignments, for example, will not then check for possible errors.
- Current versions of R (beginning with 2.8.0) try to accommodate S4 classes that extend S3 classes, that is, those including an S3 class in the `contains=` argument to `setClass`. Specifically, objects from such classes will contain the S3 class as a slot, and some S3 computations will recognize the S3 class, including method dispatch and the function `inherits`. See `S3Class` for details. The S3 classes *must* have been registered by a call to `setOldClass`. The basic caution remains true however: There is no guarantee that all S3 computations will be compatible, and some are known not to be.
- These caveats apply to S3 classes; that is, objects with a class assigned by some R function but without a formal class definition. In contrast, the built-in vector types (`numeric`, `list`, etc.) are generally fine as slots or for `contains=` classes. These object types don't have formal slots, but the base code in the system essentially forces them to contain the type of data they claim to have.

  Objects with a "class" of `matrix` or `array` are somewhat in between. They do not have an explicit S3 class, but do have one or two attributes. There is no general problem in having these as slots, but because there is no guarantee of a dimnames slot, they don't work as formal classes. The `ts` class, although also ancient in the S language, is implemented in R essentially as an S3 class, with the implications noted above—not suitable for a `contains=` argument—but with a few S4 methods defined. See the documentation for class `structure` for more details.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

Chambers, John M. and Hastie, Trevor J. eds (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole (Appendix A for S3 classes.)

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (Out of print.) (The description of vectors, matrix, array and time-series objects.)

See Also

Methods for analogous discussion of methods, setClass for details of specifying class definitions, is, as, new, slot

---

classRepresentation-class
Class Objects

Description

These are the objects that hold the definition of classes of objects. They are constructed and stored as meta-data by calls to the function setClass. Don't manipulate them directly, except perhaps to look at individual slots.

Details

Class definitions are stored as metadata in various packages. Additional metadata supplies information on inheritance (the result of calls to setIs). Inheritance information implied by the class definition itself (because the class contains one or more other classes) is also constructed automatically.

When a class is to be used in an R session, this information is assembled to complete the class definition. The completion is a second object of class "classRepresentation", cached for the session or until something happens to change the information. A call to getClass returns the completed definition of a class; a call to getClassDef returns the stored definition (uncompleted).

In particular, completion fills in the upward- and downward-pointing inheritance information for the class, in slots contains and subclasses respectively. It's in principle important to note that this information can depend on which packages are installed, since these may define additional subclasses or superclasses.

Slots

slots: A named list of the slots in this class; the elements of the list are the classes to which the slots must belong (or extend), and the names of the list gives the corresponding slot names.

contains: A named list of the classes this class 'contains'; the elements of the list are objects of SClassExtension. The list may be only the direct extensions or all the currently known extensions (see the details).

virtual: Logical flag, set to TRUE if this is a virtual class.

prototype: The object that represents the standard prototype for this class; i.e., the data and slots returned by a call to new for this class with no special arguments. Don't mess with the prototype object directly.

validity: Optionally, a function to be used to test the validity of objects from this class. See validObject.

access: Access control information. Not currently used.

className: The character string name of the class.

package: The character string name of the package to which the class belongs. Nearly always the package on which the metadata for the class is stored, but in operations such as constructing inheritance information, the internal package name rules.

subclasses: A named list of the classes known to extend this class'; the elements of the list are objects of class `SClassExtension`. The list is currently only filled in when completing the class definition (see the details).

versionKey: Object of class `"externalptr"`; eventually will perhaps hold some versioning information, but not currently used.

sealed: Object of class `"logical"`; is this class sealed? If so, no modifications are allowed.

See Also

See function `setClass` to supply the information in the class definition. See Classes for a more basic discussion of class information.

`Documentation` *Using and Creating On-line Documentation for Classes and Methods*

Description

Special documentation can be supplied to describe the classes and methods that are created by the software in the methods package. Techniques to access this documentation and to create it in R help files are described here.

Getting documentation on classes and methods

You can ask for on-line help for class definitions, for specific methods for a generic function, and for general discussion of methods for a generic function. These requests use the `?` operator (see `help` for a general description of the operator). Of course, you are at the mercy of the implementer as to whether there *is* any documentation on the corresponding topics.

Documentation on a class uses the argument `class` on the left of the `?`, and the name of the class on the right; for example,

```
class ?  genericFunction
```

to ask for documentation on the class `"genericFunction"`.

When you want documentation for the methods defined for a particular function, you can ask either for a general discussion of the methods or for documentation of a particular method (that is, the method that would be selected for a particular set of actual arguments).

Overall methods documentation is requested by calling the `?` operator with `methods` as the left-side argument and the name of the function as the right-side argument. For example,

```
methods ?  initialize
```

asks for documentation on the methods for the `initialize` function.

Asking for documentation on a particular method is done by giving a function call expression as the right-hand argument to the `"?"` operator. There are two forms, depending on whether you prefer to give the class names for the arguments or expressions that you intend to use in the actual call.

If you planned to evaluate a function call, say `myFun(x, sqrt(wt))` and wanted to find out something about the method that would be used for this call, put the call on the right of the `"?"` operator:

```
?myFun(x, sqrt(wt))
```

A method will be selected, as it would be for the call itself, and documentation for that method will be requested. If `myFun` is not a generic function, ordinary documentation for the function will be requested.

If you know the actual classes for which you would like method documentation, you can supply these explicitly in place of the argument expressions. In the example above, if you want method documentation for the first argument having class `"maybeNumber"` and the second `"logical"`, call the `"?"` operator, this time with a left-side argument `method`, and with a function call on the right using the class names as arguments:

```
method ?  myFun("maybeNumber", "logical")
```

Once again, a method will be selected, this time corresponding to the specified classes, and method documentation will be requested. This version only works with generic functions.

The two forms each have advantages. The version with actual arguments doesn't require you to figure out (or guess at) the classes of the arguments. On the other hand, evaluating the arguments may take some time, depending on the example. The version with class names does require you to pick classes, but it's otherwise unambiguous. It has a subtler advantage, in that the classes supplied may be virtual classes, in which case no actual argument will have specifically this class. The class `"maybeNumber"`, for example, might be a class union (see the example for `setClassUnion`).

In either form, methods will be selected as they would be in actual computation, including use of inheritance and group generic functions. See `selectMethod` for the details, since it is the function used to find the appropriate method.

Writing Documentation for Methods

The on-line documentation for methods and classes uses some extensions to the R documentation format to implement the requests for class and method documentation described above. See the document *Writing R Extensions* for the available markup commands (you should have consulted this document already if you are at the stage of documenting your software).

In addition to the specific markup commands to be described, you can create an initial, overall file with a skeleton of documentation for the methods defined for a particular generic function:

```
promptMethods("myFun")
```

will create a file, 'myFun-methods.Rd' with a skeleton of documentation for the methods defined for function `myFun`. The output from `promptMethods` is suitable if you want to describe all or most of the methods for the function in one file, separate from the documentation of the generic function itself. Once the file has been filled in and moved to the 'man' subdirectory of your source package, requests for methods documentation will use that file, both for specific methods documentation as described above, and for overall documentation requested by

```
methods ?  myFun
```

You are not required to use `promptMethods`, and if you do, you may not want to use the entire file created:

- If you want to document the methods in the file containing the documentation for the generic function itself, you can cut-and-paste to move the `\alias` lines and the `Methods` section from the file created by `promptMethods` to the existing file.
- On the other hand, if these are auxiliary methods, and you only want to document the added or modified software, you should strip out all but the relevant `\alias` lines for the methods of interest, and remove all but the corresponding `\item` entries in the `Methods` section. Note that in this case you will usually remove the first `\alias` line as well, since that is the marker for general methods documentation on this function (in the example, `\alias{myfun-methods}`).

If you simply want to direct documentation for one or more methods to a particular R documentation file, insert the appropriate alias.

`dotsMethods` *The Use of "..." in Method Signatures*

Description

The "..." argument in R functions is treated specially, in that it matches zero, one or more actual arguments (and so, objects). A mechanism has been added to R to allow "..." as the signature of a generic function. Methods defined for such functions will be selected and called when *all* the arguments matching "..." are from the specified class or from some subclass of that class.

Using "..." in a Signature

Beginning with version 2.8.0 of R, S4 methods can be dispatched (selected and called) corresponding to the special argument "...". Currently, "..." cannot be mixed with other formal arguments: either the signature of the generic function is "..." only, or it does not contain "...". (This restriction may be lifted in a future version.)

Given a suitable generic function, methods are specified in the usual way by a call to `setMethod`. The method definition must be written expecting all the arguments corresponding to "..." to be from the class specified in the method's signature, or from a class that extends that class (i.e., a subclass of that class).

Typically the methods will pass "..." down to another function or will create a list of the arguments and iterate over that. See the examples below.

When you have a computation that is suitable for more than one existing class, a convenient approach may be to define a union of these classes by a call to `setClassUnion`. See the example below.

Method Selection and Dispatch for "..."

See Methods for a general discussion. The following assumes you have read the "Method Selection and Dispatch" section of that documentation.

A method selecting on "..." is specified by a single class in the call to `setMethod`. If all the actual arguments corresponding to "..." have this class, the corresponding method is selected directly.

Otherwise, the class of each argument and that class' superclasses are computed, beginning with the first "..." argument. For the first argument, eligible methods are those for any of the classes. For each succeeding argument that introduces a class not considered previously, the eligible methods are further restricted to those matching the argument's class or superclasses. If no further eligible classes exist, the iteration breaks out and the default method, if any, is selected.

At the end of the iteration, one or more methods may be eligible. If more than one, the selection looks for the method with the least distance to the actual arguments. For each argument, any inherited method corresponds to a distance, available from the `contains` slot of the class definition. Since the same class can arise for more than one argument, there may be several distances associated with it. Combining them is inevitably arbitrary: the current computation uses the minimum distance. Thus, for example, if a method matched one argument directly, one as first generation superclass and another as a second generation superclass, the distances are 0, 1 and 2. The current selection computation would use distance 0 for this method. In particular, this selection criterion tends to use a method that matches exactly one or more of the arguments' class.

As with ordinary method selection, there may be multiple methods with the same distance. A warning message is issued and one of the methods is chosen (the first encountered, which in this case is rather arbitrary).

Notice that, while the computation examines all arguments, the essential cost of dispatch goes up with the number of *distinct* classes among the arguments, likely to be much smaller than the number of arguments when the latter is large.

Implementation Details

Methods dispatching on "..." were introduced in version 2.8.0 of R. The initial implementation of the corresponding selection and dispatch is in an R function, for flexibility while the new mechanism is being studied. In this implementation, a local version of `setGeneric` is inserted in the generic function's environment. The local version selects a method according to the criteria above and calls that method, from the environment of the generic function. This is slightly different from the action taken by the C implementation when "..." is not involved. Aside from the extra computing time required, the method is evaluated in a true function call, as opposed to the special context constructed by the C version (which cannot be exactly replicated in R code.) However, situations in which different computational results would be obtained have not been encountered so far, and seem very unlikely.

Methods dispatching on arguments other than "..." are *cached* by storing the inherited method in the table of all methods, where it will be found on the next selection with the same combination of classes in the actual arguments (but not used for inheritance searches). Methods based on "..." are also cached, but not found quite as immediately. As noted, the selected method depends only on the set of classes that occur in the "..." arguments. Each of these classes can appear one or more times, so many combinations of actual argument classes will give rise to the same effective signature. The selection computation first computes and sorts the distinct classes encountered. This gives a label that will be cached in the table of all methods, avoiding any further search for inherited classes after the first occurrence. A call to `showMethods` will expose such inherited methods.

The intention is that the "..." features will be added to the standard C code when enough experience with them has been obtained. It is possible that at the same time, combinations of "..." with other arguments in signatures may be supported.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

For the general discussion of methods, see Methods and links from there.

Examples

```
cc <- function(...)c(...)

setGeneric("cc")

setMethod("cc", "character", function(...)paste(...))

setClassUnion("Number", c("numeric", "complex"))

setMethod("cc", "Number", function(...) sum(...))
```

```
setClass("cdate", contains = "character", representation(date = "Date"))

setClass("vdate", contains = "vector", representation(date = "Date"))

cd1 = new("cdate", "abcdef", date = Sys.Date())

cd2 = new("vdate", "abcdef", date = Sys.Date())

stopifnot(identical(cc(letters, character(), cd1), paste(letters, character(), cd1))) # t

stopifnot(identical(cc(letters, character(), cd2), c(letters, character(), cd2))) # the d

stopifnot(identical(cc(1:10, 1+1i), sum(1:10, 1+1i))) # the "Number" method

stopifnot(identical(cc(1:10, 1+1i, TRUE), c(1:10, 1+1i, TRUE))) # the default

stopifnot(identical(cc(), c())) # no arguments implies the default method

setGeneric("numMax", function(...)standardGeneric("numMax"))

setMethod("numMax", "numeric", function(...)max(...)) # won't work for complex data
setMethod("numMax", "Number", function(...) paste(...)) # should not be selected w/o comp

stopifnot(identical(numMax(1:10, pi, 1+1i), paste(1:10, pi, 1+1i)))
stopifnot(identical(numMax(1:10, pi, 1), max(1:10, pi, 1)))

try(numMax(1:10, pi, TRUE)) # should be an error:  no default method

## A generic version of paste(), dispatching on the "..." argument:
setGeneric("paste", signature = "...")

setMethod("paste", "Number", function(..., sep, collapse) c(...))

stopifnot(identical(paste(1:10, pi, 1), c(1:10, pi, 1)))
```

`environment-class` *Class "environment"*

Description

A formal class for R environments.

Objects from the Class

Objects can be created by calls of the form `new("environment", ...)`. The arguments in ..., if any, should be named and will be assigned to the newly created environment.

Methods

coerce `signature(from = "ANY", to = "environment")`: calls `as.environment`.

initialize `signature(object = "environment")`: Implements the assignments in the new environment. Note that the `object` argument is ignored; a new environment is *always* created, since environments are not protected by copying.

See Also

`new.env`

`findClass` *Computations with Classes*

Description

Functions to find and manipulate class definitions.

Usage

```
removeClass(Class, where)

isClass(Class, formal=TRUE, where)

getClasses(where, inherits = missing(where))

findClass(Class, where, unique = "")

resetClass(Class, classDef, where)

sealClass(Class, where)
```

Arguments

`Class`	character string name for the class. The functions will usually take a class definition instead of the string. To restrict the class to those defined in a particular package, set the `packageSlot` of the character string.
`where`	The environment in which to modify or remove the definition. Defaults to the top-level environment of the calling function (the global environment for ordinary computations, but the environment or name space of a package in the source for a package). When searching for class definitions, `where` defines where to do the search, and the default is to search from the top-level environment or name space of the caller to this function.
`unique`	if `findClass` expects a unique location for the class, `unique` is a character string explaining the purpose of the search (and is used in warning and error messages). By default, multiple locations are possible and the function always returns a list.
`inherits`	in a call to `getClasses`, should the value returned include all parent environments of `where`, or that environment only? Defaults to `TRUE` if `where` is omitted, and to `FALSE` otherwise.
`formal`	Should a formal definition be required?
`classDef`	For `removeClass`, the optional class definition (but usually it's better for `Class` to be the class definition, and to omit `classDef`).

Details

These are the functions that test and manipulate formal class definitions. Brief documentation is provided below. See the references for an introduction and for more details.

removeClass: Remove the definition of this class, from the environment `where` if this argument is supplied; if not, `removeClass` will search for a definition, starting in the top-level environment of the call to `removeClass`, and remove the (first) definition found.

isClass: Is this the name of a formally defined class? (Argument `formal` is for compatibility and is ignored.)

getClasses: The names of all the classes formally defined on `where`. If called with no argument, all the classes visible from the calling function (if called from the top-level, all the classes in any of the environments on the search list). The `inherits` argument can be used to search a particular environment and all its parents, but usually the default setting is what you want.

findClass: The list of environments or positions on the search list in which a class definition of `Class` is found. If `where` is supplied, this is an environment (or name space) from which the search takes place; otherwise the top-level environment of the caller is used. If `unique` is supplied as a character string, `findClass` returns a single environment or position. By default, it always returns a list. The calling function should select, say, the first element as a position or environment for functions such as `get`.

If `unique` is supplied as a character string, `findClass` will warn if there is more than one definition visible (using the string to identify the purpose of the call), and will generate an error if no definition can be found.

resetClass: Reset the internal definition of a class. Causes the complete definition of the class to be re-computed, from the representation and superclasses specified in the original call to `setClass`.

This function is called when aspects of the class definition are changed. You would need to call it explicitly if you changed the definition of a class that this class extends (but doing that in the middle of a session is living dangerously, since it may invalidate existing objects).

sealClass: Seal the current definition of the specified class, to prevent further changes. It is possible to seal a class in the call to `setClass`, but sometimes further changes have to be made (e.g., by calls to `setIs`). If so, call `sealClass` after all the relevant changes have been made.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`setClassUnion`, `Methods`, `makeClassRepresentation`

`findMethods` *Descriptions of the Methods Defined for a Generic Function*

Description

These functions convert the methods defined in a table for a generic function (as used for selection of methods) into a list, for study or display, or provide other information summarizing the methods.

The function `findMethods` returns a list of the method definitions currently existing for generic function `f`, limited to the methods defined in environment `where` if that argument is supplied and possibly limited to those including one or more of the specified `classes` in the method signature.

The function `findMethodSignatures` returns a character matrix whose rows are the class names from the signature of the corresponding methods; it operates either from a list returned by `findMethods`, or by computing such a list itself, given the same arguments as `findMethods` .

The function `hasMethods` returns `TRUE` or `FALSE` according to whether there is a non-empty table of methods for function `f` in the environment or search position `where` (or anywhere on the search list if `where` is missing).

The function `getMethods` is an older alternative to `findMethods` , returning information in the form of an object of class `MethodsList`, previously used for method dispatch. It is not recommended, since the object returned is usually less convenient than the value of `findMethods`.

Usage

```
findMethods(f, where, classes = character(), inherited = FALSE)
findMethodSignatures(..., target = TRUE, methods = )

hasMethods(f, where, package)

getMethods(f, where)
```

Arguments

`f`	A generic function or the character-string name of one.
`where`	Optionally, an environment or position on the search list to look for methods metadata. If `where` is missing, `findMethods` uses the current table of methods in the generic function itself, and `hasMethods` looks for metadata anywhere in the search list.
`classes`	If supplied, only methods whose signatures contain at least one of the supplied classes will be included in the value returned.
`inherited`	Logical flag; if `TRUE`, the table of all methods, inherited or defined directly, will be used; otherwise, only the methods explicitly defined. Option `TRUE` is meaningful only if `where` is missing.
`...`	In the call to `findMethodSignatures`, any arguments that might be given to `findMethods`.
`target`	Optional flag to `findMethodSignatures`; if `TRUE`, the signatures used are the target signatures (the classes for which the method will be selected); if `FALSE`, they will be the signatures are defined. The difference is only meaningful if `inherited` is `TRUE`.

`methods` In the call to `findMethodSignatures`, an optional list of methods, presumably returned by a previous call to `findMethods`. If missing, that function will be call with the ... arguments.

`package` In a call to `hasMethods`, the package name for the generic function (e.g., `"base"` for primitives). If missing this will be inferred either from the `"package"` attribute of the function name, if any, or from the package slot of the generic function. See details below.

Details

The functions obtain a table of the defined methods, either from the generic function or from the stored metadata object in the environment specified by `where`. The information in the table is converted as described above to produce the returned value.

Note that the list returned may contain the primitive function itself, when the generic corresponds to a primitive. (Basically, because primitive functions are abnormal R objects, which cannot currently be extended as method definitions.) Computations that use the returned list to derive other information need to take account of this possibility. See the implementation of `findMethodSignatures` for an example.

Note that `hasMethods`, but not the other functions, can be used even if no generic function of this name is currently found. In this case `package` must either be supplied as an argument or included as an attribute of `f`, since the package name is part of the identification of the methods tables.

These functions should be used in searching for or summarizing methods, rather than using information about how the metadata for methods is stored.

See Also

`showMethods`, `MethodsList-class`

Examples

```
mm <-  findMethods("Ops")
findMethodSignatures(methods = mm)
```

`fixPre1.8` *Fix Objects Saved from R Versions Previous to 1.8*

Description

Beginning with R version 1.8.0, the class of an object contains the identification of the package in which the class is defined. The function `fixPre1.8` fixes and re-assigns objects missing that information (typically because they were loaded from a file saved with a previous version of R.)

Usage

```
fixPre1.8(names, where)
```

Arguments

`names` Character vector of the names of all the objects to be fixed and re-assigned.

`where` The environment from which to look for the objects, and for class definitions. Defaults to the top environment of the call to `fixPre1.8`, the global environment if the function is used interactively.

Details

The named object will be saved where it was found. Its class attribute will be changed to the full form required by R 1.8; otherwise, the contents of the object should be unchanged.

Objects will be fixed and re-assigned only if all the following conditions hold:

1. The named object exists.
2. It is from a defined class (not a basic datatype which has no actual class attribute).
3. The object appears to be from an earlier version of R.
4. The class is currently defined.
5. The object is consistent with the current class definition.

If any condition except the second fails, a warning message is generated.

Note that `fixPre1.8` currently fixes *only* the change in class attributes. In particular, it will not fix binary versions of packages installed with earlier versions of R if these use incompatible features. Such packages must be re-installed from source, which is the wise approach always when major version changes occur in R.

Value

The names of all the objects that were in fact re-assigned.

---

genericFunction-class

Generic Function Objects

Description

Generic functions (objects from or extending class `genericFunction`) are extended function objects, containing information used in creating and dispatching methods for this function. They also identify the package associated with the function and its methods.

Objects from the Class

Generic functions are created and assigned by `setGeneric` or `setGroupGeneric` and, indirectly, by `setMethod`.

As you might expect `setGeneric` and `setGroupGeneric` create objects of class `"genericFunction"` and `"groupGenericFunction"` respectively.

Slots

`.Data`: Object of class `"function"`, the function definition of the generic, usually created automatically as a call to `standardGeneric`.

`generic`: Object of class `"character"`, the name of the generic function.

`package`: Object of class `"character"`, the name of the package to which the function definition belongs (and *not* necessarily where the generic function is stored). If the package is not specified explicitly in the call to `setGeneric`, it is usually the package on which the corresponding non-generic function exists.

`group`: Object of class `"list"`, the group or groups to which this generic function belongs. Empty by default.

valueClass: Object of class "character"; if not an empty character vector, identifies one or more classes. It is asserted that all methods for this function return objects from these class (or from classes that extend them).

signature: Object of class "character", the vector of formal argument names that can appear in the signature of methods for this generic function. By default, it is all the formal arguments, except for .... Order matters for efficiency: the most commonly used arguments in specifying methods should come first.

default: Object of class "OptionalMethods", the default method for this function. Generated automatically and used to initialize method dispatch.

skeleton: Object of class "call", a slot used internally in method dispatch. Don't expect to use it directly.

Extends

Class "function", from data part.
Class "OptionalMethods", by class "function".
Class "PossibleMethod", by class "function".

Methods

Generic function objects are used in the creation and dispatch of formal methods; information from the object is used to create methods list objects and to merge or update the existing methods for this generic.

GenericFunctions *Tools for Managing Generic Functions*

Description

The functions documented here manage collections of methods associated with a generic function, as well as providing information about the generic functions themselves.

Usage

```
isGeneric(f, where, fdef, getName = FALSE)
isGroup(f, where, fdef)
removeGeneric(f, where)

dumpMethod(f, signature, file, where, def)
findFunction(f, generic = TRUE, where = topenv(parent.frame()))
dumpMethods(f, file, signature, methods, where)
signature(...)

removeMethods(f, where = topenv(parent.frame()), all = missing(where))
setReplaceMethod(f, ..., where = topenv(parent.frame()))

getGenerics(where, searchForm = FALSE)
```

Arguments

f	The character string naming the function.
where	The environment, name space, or search-list position from which to search for objects. By default, start at the top-level environment of the calling function, typically the global environment (i.e., use the search list), or the name space of a package from which the call came. It is important to supply this argument when calling any of these functions indirectly. With package name spaces, the default is likely to be wrong in such calls.
signature	The class signature of the relevant method. A signature is a named or unnamed vector of character strings. If named, the names must be formal argument names for the generic function. If signature is unnamed, the default is to use the first length(signature) formal arguments of the function.
file	The file or connection on which to dump method definitions.
def	The function object defining the method; if omitted, the current method definition corresponding to the signature.
...	Named or unnamed arguments to form a signature.
generic	In testing or finding functions, should generic functions be included. Supply as FALSE to get only non-generic functions.
fdef	Optional, the generic function definition. Usually omitted in calls to isGeneric
getName	If TRUE, isGeneric returns the name of the generic. By default, it returns TRUE.
methods	The methods object containing the methods to be dumped. By default, the methods defined for this generic (optionally on the specified where location).
all	in removeMethods, logical indicating if all (default) or only the first method found should be removed.
searchForm	In getGenerics, if TRUE, the package slot of the returned result is in the form used by search(), otherwise as the simple package name (e.g, "package:base" vs "base").

Summary of Functions

isGeneric: Is there a function named f, and if so, is it a generic?

The getName argument allows a function to find the name from a function definition. If it is TRUE then the name of the generic is returned, or FALSE if this is not a generic function definition.

The behavior of isGeneric and getGeneric for primitive functions is slightly different. These functions don't exist as formal function objects (for efficiency and historical reasons), regardless of whether methods have been defined for them. A call to isGeneric tells you whether methods have been defined for this primitive function, anywhere in the current search list, or in the specified position where. In contrast, a call to getGeneric will return what the generic for that function would be, even if no methods have been currently defined for it.

removeGeneric, removeMethods: Remove all the methods for the generic function of this name. In addition, removeGeneric removes the function itself; removeMethods restores the non-generic function which was the default method. If there was no default method, removeMethods leaves a generic function with no methods.

standardGeneric: Dispatches a method from the current function call for the generic function `f`. It is an error to call `standardGeneric` anywhere except in the body of the corresponding generic function.

Note that `standardGeneric` is a primitive function in the **base** package for efficiency reasons, but rather documented here where it belongs naturally.

dumpMethod: Dump the method for this generic function and signature.

findFunction: return a list of either the positions on the search list, or the current top-level environment, on which a function object for `name` exists. The returned value is *always* a list, use the first element to access the first visible version of the function. See the example.

NOTE: Use this rather than `find` with `mode="function"`, which is not as meaningful, and has a few subtle bugs from its use of regular expressions. Also, `findFunction` works correctly in the code for a package when attaching the package via a call to `library`.

dumpMethods: Dump all the methods for this generic.

signature: Returns a named list of classes to be matched to arguments of a generic function.

getGenerics: returns the names of the generic functions that have methods defined on `where`; this argument can be an environment or an index into the search list. By default, the whole search list is used. (`allGenerics()` is a *deprecated* alias for `getGenerics`.)

The methods definitions are stored with package qualifiers; for example, methods for function `"initialize"` might refer to two different functions of that name, on different packages. The package names corresponding to the method list object are contained in the slot `package` of the returned object. The form of the returned name can be plain (e.g., `"base"`), or in the form used in the search list (`"package:base"`) according to the value of `searchForm`

Details

setGeneric: If there is already a non-generic function of this name, it will be used to define the generic unless `def` is supplied, and the current function will become the default method for the generic.

If `def` is supplied, this defines the generic function, and no default method will exist (often a good feature, if the function should only be available for a meaningful subset of all objects).

Arguments `group` and `valueClass` are retained for consistency with S-Plus, but are currently not used.

isGeneric: If the `fdef` argument is supplied, take this as the definition of the generic, and test whether it is really a generic, with `f` as the name of the generic. (This argument is not available in S-Plus.)

removeGeneric: If `where` supplied, just remove the version on this element of the search list; otherwise, removes the first version encountered.

standardGeneric: Generic functions should usually have a call to `standardGeneric` as their entire body. They can, however, do any other computations as well.

The usual `setGeneric` (directly or through calling `setMethod`) creates a function with a call to `standardGeneric`.

dumpMethod: The resulting source file will recreate the method.

findFunction: If `generic` is `FALSE`, ignore generic functions.

dumpMethods: If `signature` is supplied only the methods matching this initial signature are dumped. (This feature is not found in S-Plus: don't use it if you want compatibility.)

signature: The advantage of using `signature` is to provide a check on which arguments you meant, as well as clearer documentation in your method specification. In addition, `signature` checks that each of the elements is a single character string.

removeMethods: Returns TRUE if f was a generic function, FALSE (silently) otherwise.

If there is a default method, the function will be re-assigned as a simple function with this definition. Otherwise, the generic function remains but with no methods (so any call to it will generate an error). In either case, a following call to setMethod will consistently re-establish the same generic function as before.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

getMethod (also for selectMethod), setGeneric, setClass, showMethods

Examples

```
require(stats) # for lm

## get the function "myFun" -- throw an error if 0 or > 1 versions visible:
findFuncStrict <- function(fName) {
  allF <- findFunction(fName)
  if(length(allF) == 0)
    stop("No versions of ",fName," visible")
  else if(length(allF) > 1)
    stop(fName," is ambiguous: ", length(allF), " versions")
  else
    get(fName, allF[[1]])
}

try(findFuncStrict("myFun"))# Error: no version
lm <- function(x) x+1
try(findFuncStrict("lm"))#    Error: 2 versions
findFuncStrict("findFuncStrict")# just 1 version
rm(lm)



## method dumping -----------------------------------

setClass("A", representation(a="numeric"))
setMethod("plot", "A", function(x,y,...){ cat("A meth\n") })
dumpMethod("plot","A", file="")
## Not run:
setMethod("plot", "A",
function (x, y, ...)
{
    cat("AAAAA\n")
}
)
## End(Not run)
tmp <- tempfile()
dumpMethod("plot","A", file=tmp)
## now remove, and see if we can parse the dump
stopifnot(removeMethod("plot", "A"))
```

```
source(tmp)
stopifnot(is(getMethod("plot", "A"), "MethodDefinition"))

## same with dumpMethods() :
setClass("B", contains="A")
setMethod("plot", "B", function(x,y,...){ cat("B ...\n") })
dumpMethods("plot", file=tmp)
stopifnot(removeMethod("plot", "A"),
          removeMethod("plot", "B"))
source(tmp)
stopifnot(is(getMethod("plot", "A"), "MethodDefinition"),
          is(getMethod("plot", "B"), "MethodDefinition"))
```

`getClass` *Get Class Definition*

Description

Get the definition of a class.

Usage

```
getClass(Class, .Force = FALSE, where)
getClassDef(Class, where, package)
```

Arguments

`Class`	the character-string name of the class.
`.Force`	if `TRUE`, return `NULL` if the class is undefined; otherwise, an undefined class results in an error.
`where`	environment from which to begin the search for the definition; by default, start at the top-level (global) environment and proceed through the search list.
`package`	the name of the package asserted to hold the definition. Supplied instead of `where`, with the distinction that the package need not be currently attached.

Details

A call to `getClass` returns the complete definition of the class supplied as a string, including all slots, etc. in classes that this class extends. A call to `getClassDef` returns the definition of the class from the environment `where`, unadorned. It's usually `getClass` you want.

If you really want to know whether a class is formally defined, call `isClass`.

Value

The object defining the class. This is an object of class `classRepresentation`. However, *do not* deal with the contents of the object directly unless you are very sure you know what you're doing. Even then, it is nearly always better practice to use functions such as `setClass` and `setIs`. Messing up a class object will cause great confusion.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

Classes, `setClass`, `isClass`.

Examples

```
getClass("numeric") ## a built in class

cld <- getClass("thisIsAnUndefinedClass", .Force = TRUE)
cld ## a NULL prototype
## If you are really curious:
utils::str(cld)
## Whereas these generate errors:
try(getClass("thisIsAnUndefinedClass"))
try(getClassDef("thisIsAnUndefinedClass"))
```

`getMethod` *Get or Test for the Definition of a Method*

Description

Functions to look for a method corresponding to a given generic function and signature. The functions `getMethod` and `selectMethod` return the method; the functions `existsMethod` and `hasMethod` test for its existence. In both cases the first function only gets direct definitions and the second uses inheritance. In all cases, the search is in the generic function itself or in the package/environment specified by argument `where`.

The function `findMethod` returns the package(s) in the search list (or in the packages specified by the `where` argument) that contain a method for this function and signature.

Usage

```
getMethod(f, signature=character(), where, optional = FALSE,
          mlist, fdef)

existsMethod(f, signature = character(), where)

findMethod(f, signature, where)

selectMethod(f, signature, optional = FALSE, useInherited =,
             mlist = , fdef = , verbose = )

hasMethod(f, signature=character(), where)
```

Arguments

f	A generic function or the character-string name of one.
signature	the signature of classes to match to the arguments of f. See the details below.
where	The position or environment in which to look for the method(s): by default, the table of methods defined in the generic function itself is used.
optional	If the selection in selectMethod does find a valid method an error is generated, unless this argument is TRUE. In that case, the value returned is NULL if no method matches.
mlist, fdef, useInherited, verbose	Optional arguments to getMethod and selectMethod. Avoid these: some will work and others will not, and none of them is required for normal use of the functions.

Details

The signature argument specifies classes, corresponding to formal arguments of the generic function; to be precise, to the signature slot of the generic function object. The argument may be a vector of strings identifying classes, and may be named or not. Names, if supplied, match the names of those formal arguments included in the signature of the generic. That signature is normally all the arguments except .... However, generic functions can be specified with only a subset of the arguments permitted, or with the signature taking the arguments in a different order.

It's a good idea to name the arguments in the signature to avoid confusion, if you're dealing with a generic that does something special with its signature. In any case, the elements of the signature are matched to the formal signature by the same rules used in matching arguments in function calls (see match.call).

The strings in the signature may be class names, "missing" or "ANY". See Methods for the meaning of these in method selection. Arguments not supplied in the signature implicitly correspond to class "ANY"; in particular, giving an empty signature means to look for the default method.

A call to getMethod returns the method for a particular function and signature. As with other get functions, argument where controls where the function looks (by default anywhere in the search list) and argument optional controls whether the function returns NULL or generates an error if the method is not found. The search for the method makes no use of inheritance.

The function selectMethod also looks for a method given the function and signature, but makes full use of the method dispatch mechanism; i.e., inherited methods and group generics are taken into account just as they would be in dispatching a method for the corresponding signature, with the one exception that conditional inheritance is not used. Like getMethod, selectMethod returns NULL or generates an error if the method is not found, depending on the argument optional.

The functions existsMethod and hasMethod return TRUE or FALSE according to whether a method is found, the first corresponding to getMethod (no inheritance) and the second to selectMethod.

Value

The call to selectMethod or getMethod returns the selected method, if one is found. (This class extends function, so you can use the result directly as a function if that is what you want.) Otherwise an error is thrown if optional is FALSE and NULL is returned if optional is TRUE.

The returned method object is a MethodDefinition object, *except* that the default method for a primitive function is required to be the primitive itself. Note therefore that the only reliable test that the search failed is is.null().

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`Methods` for the details of method selection; `GenericFunctions` for other functions manipulating methods and generic function objects; `MethodDefinition` for the class that represents method definitions.

Examples

```
setGeneric("testFun", function(x)standardGeneric("testFun"))
setMethod("testFun", "numeric", function(x)x+1)
hasMethod("testFun", "numeric")
## Not run: [1] TRUE
hasMethod("testFun", "integer") #inherited
## Not run: [1] TRUE
existsMethod("testFun", "integer")
## Not run: [1] FALSE
hasMethod("testFun") # default method
## Not run: [1] FALSE
hasMethod("testFun", "ANY")
## Not run: [1] FALSE
```

---

getPackageName *The Name associated with a Given Package*

Description

The functions below produce the package associated with a particular environment or position on the search list, or of the package containing a particular function. They are primarily used to support computations that need to differentiate objects on multiple packages.

Usage

```
getPackageName(where, create = TRUE)
setPackageName(pkg, env)

packageSlot(object)
packageSlot(object) <- value
```

Arguments

`where`	the environment or position on the search list associated with the desired package.
`object`	object providing a character string name, plus the package in which this object is to be found.
`value`	the name of the package.

create	flag, should a package name be created if none can be inferred? If TRUE and no non-empty package name is found, the current date and time are used as a package name, and a warning is issued. The created name is stored in the environment if that environment is not locked.
pkg, env	make the string in pkg the internal package name for all computations that set class and method definitions in environment env.

Details

Package names are normally installed during loading of the package, by the INSTALL script or by the library function. (Currently, the name is stored as the object .packageName but don't trust this for the future.)

Value

packageName returns the character-string name of the package (without the extraneous "package:" found in the search list).

packageSlot returns or sets the package name slot (currently an attribute, not a formal slot, but this may change someday).

setPackageName can be used to establish a package name in an environment that would otherwise not have one. This allows you to create classes and/or methods in an arbitrary environment, but it is usually preferable to create packages by the standard R programming tools (package.skeleton, etc.)

See Also

search

Examples

```
## all the following usually return "base"
getPackageName(length(search()))
getPackageName(baseenv())
getPackageName(asNamespace("base"))
getPackageName("package:base")
```

---

hasArg *Look for an Argument in the Call*

Description

Returns TRUE if name corresponds to an argument in the call, either a formal argument to the function, or a component of ..., and FALSE otherwise.

Usage

```
hasArg(name)
```

Arguments

name	The unquoted name of a potential argument.

Details

The expression `hasArg(x)`, for example, is similar to `!missing(x)`, with two exceptions. First, `hasArg` will look for an argument named `x` in the call if `x` is not a formal argument to the calling function, but `...` is. Second, `hasArg` never generates an error if given a name as an argument, whereas `missing(x)` generates an error if `x` is not a formal argument.

Value

Always `TRUE` or `FALSE` as described above.

See Also

`missing`

Examples

```
ftest <- function(x1, ...) c(hasArg(x1), hasArg(y2))

ftest(1) ## c(TRUE, FALSE)
ftest(1, 2)  ## c(TRUE, FALSE)
ftest(y2=2)   ## c(FALSE, TRUE)
ftest(y=2)    ## c(FALSE, FALSE) (no partial matching)
ftest(y2 = 2, x=1)  ## c(TRUE, TRUE) partial match x1
```

`implicitGeneric` *Manage Implicit Versions of Generic Functions*

Description

Create or access implicit generic functions, used to enforce consistent generic versions of functions that are not currently generic. Function `implicitGeneric()` returns the implicit generic version, `setGenericImplicit()` turns a generic implicit, `prohibitGeneric()` prevents your function from being made generic, and `registerImplicitGenerics()` saves a set of implicit generic definitions in the cached table of the current session.

Usage

```
implicitGeneric(name, where, generic)
setGenericImplicit(name, where, restore = TRUE)
prohibitGeneric(name, where)
registerImplicitGenerics(what, where)
```

Arguments

`name`	Character string name of the function.
`where`	Package or environment in which to register the implicit generics. When using the functions from the top level of your own package source, this argument can usually be omitted (and should be).
`generic`	Optionally, the generic function definition to be cached, but usually omitted. See Details section.

restore	Should the non-generic version of the function be restored after the current.
what	For registerImplicitGenerics(), Optional table of the implicit generics to register, but nearly always omitted. See Details section.

Details

Multiple packages may define methods for the same function, using the version of a function stored in one package. All these methods should be marshaled and dispatched consistently when a user calls the function. For consistency, the generic version of the function must have a unique definition (the same arguments allowed in methods signatures, the same values for optional slots such as the value class, and the same standard or non-standard definition of the function itself).

If the original function is already an S4 generic, there is no problem. The implicit generic mechanism enforces consistency when the version in the package owning the function is *not* generic. If a call to setGeneric() attempts to turn a function in another package into a generic, the mechanism compares the proposed new generic function to the implicit generic version of that function. If the two agree, all is well. If not, and if the function belongs to another package, then the new generic will not be associated with that package. Instead, a warning is issued and a separate generic function is created, with its package slot set to the current package, not the one that owns the non-generic version of the function. The effect is that the new package can still define methods for this function, but it will not share the methods in other packages, since it is forcing a different definition of the generic function.

The right way to proceed in nearly all cases is to call setGeneric("foo"), giving *only* the name of the function; this will automatically use the implicit generic version. If you don't like that version, the best solution is to convince the owner of the other package to agree with you and to insert code to define the non-default properties of the function (even if the owner does not want foo() to be a generic by default).

For any function, the implicit generic form is a standard generic in which all formal arguments, except for ..., are allowed in the signature of methods. If that is the suitable generic for a function, no action is needed. If not, the best mechanism is to set up the generic in the code of the package owning the function, and to then call setGenericImplicit() to record the implicit generic and restore the non-generic version. See the example.

Note that the package can define methods for the implicit generic as well; when the implicit generic is made a real generic, those methods will be included.

Other than predefining methods, the usual reason for having a non-default implicit generic is to provide a non-default signature, and the usual reason for *that* is to allow lazy evaluation of some arguments. See the example. All arguments in the signature of a generic function must be evaluated at the time the function needs to select a method. (But those arguments can be missing, with or without a default expression being defined; you can always examine missing(x) even for arguments in the signature.)

If you want to completely prohibit anyone from turning your function into a generic, call prohibitGeneric().

Value

Function implicitGeneric() returns the implicit generic definition (and caches that definition the first time if it has to construct it).

The other functions exist for their side effect and return nothing useful.

See Also

setGeneric

Examples

```
### How we would make the function with() into a generic:

## Since the second argument, 'expr' is used literally, we want
## with() to only have "data" in the signature.

## Note that 'methods'-internal code now has already extended  with()
## to do the equivalent of the following
## Not run:
setGeneric("with", signature = "data")
## Now we could predefine methods for "with" if we wanted to.

## When ready, we store the generic as implicit, and restore the original
setGenericImplicit("with")

## (This example would only work if we "owned" function with(),
##  but it is in base.)## End(Not run)

implicitGeneric("with")
```

---

initialize-methods *Methods to Initialize New Objects from a Class*

Description

The arguments to function new to create an object from a particular class can be interpreted specially for that class, by the definition of a method for function initialize for the class. This documentation describes some existing methods, and also outlines how to write new ones.

Methods

.Object = "ANY" The default method for initialize takes either named or unnamed arguments. Argument names must be the names of slots in this class definition, and the corresponding arguments must be valid objects for the slot (that is, have the same class as specified for the slot, or some superclass of that class). If the object comes from a superclass, it is not coerced strictly, so normally it will retain its current class (specifically, as(object, Class, strict = FALSE)).

Unnamed arguments must be objects of this class, of one of its superclasses, or one of its subclasses (from the class, from a class this class extends, or from a class that extends this class). If the object is from a superclass, this normally defines some of the slots in the object. If the object is from a subclass, the new object is that argument, coerced to the current class.

Unnamed arguments are processed first, in the order they appear. Then named arguments are processed. Therefore, explicit values for slots always override any values inferred from superclass or subclass arguments.

.Object = "traceable" Objects of a class that extends traceable are used to implement debug tracing (see class traceable and trace).

The initialize method for these classes takes special arguments def, tracer, exit, at, print. The first of these is the object to use as the original definition (e.g., a function). The others correspond to the arguments to trace.

.Object = "environment" The `initialize` method for environments takes a named list of objects to be used to initialize the environment.

.Object = "signature" This is a method for internal use only. It takes an optional `functionDef` argument to provide a generic function with a `signature` slot to define the argument names. See Methods for details.

Writing Initialization Methods

Initialization methods provide a general mechanism corresponding to generator functions in other languages.

The arguments to `initialize` are `.Object` and `....` Nearly always, `initialize` is called from `new`, not directly. The `.Object` argument is then the prototype object from the class.

Two techniques are often appropriate for `initialize` methods: special argument names and `callNextMethod`.

You may want argument names that are more natural to your users than the (default) slot names. These will be the formal arguments to your method definition, in addition to `.Object` (always) and `...` (optionally). For example, the method for class `"traceable"` documented above would be created by a call to `setMethod` of the form:

```
setMethod("initialize", "traceable",
  function(.Object, def, tracer, exit, at, print) ...
)
```

In this example, no other arguments are meaningful, and the resulting method will throw an error if other names are supplied.

When your new class extends another class, you may want to call the initialize method for this superclass (either a special method or the default). For example, suppose you want to define a method for your class, with special argument `x`, but you also want users to be able to set slots specifically. If you want `x` to override the slot information, the beginning of your method definition might look something like this:

```
function(.Object, x, ...) {
  Object <- callNextMethod(.Object, ...)
  if(!missing(x)) { # do something with x
```

You could also choose to have the inherited method override, by first interpreting `x`, and then calling the next method.

`is` *Is an Object from a Class*

Description

Functions to test inheritance relationships between an object and a class (`is`) or between two classes (`extends`), and to establish such relationships (`setIs`, an explicit alternative to the `contains=` argument to `setClass`).

Usage

```
is(object, class2)

extends(class1, class2, maybe=TRUE, fullInfo = FALSE)

setIs(class1, class2, test=NULL, coerce=NULL, replace=NULL,
      by = character(), where = topenv(parent.frame()), classDef =,
      extensionObject = NULL, doComplete = TRUE)
```

Arguments

`object`	any R object.
`class1, class2`	the names of the classes between which `is` relations are to be examined defined, or (more efficiently) the class definition objects for the classes.
`maybe, fullInfo`	In a call to `extends`, `maybe` is the value returned if a relation is conditional. In a call with `class2` missing, `fullInfo` is a flag, which if `TRUE` causes a list of objects of class `classExtension` to be returned, rather than just the names of the classes.
`test, coerce, replace`	In a call to `setIs`, functions optionally supplied to test whether the relation is defined, to coerce the object to `class2`, and to alter the object so that `is(object, class2)` is identical to `value`. See the details section below. The remaining arguments are for internal use and/or usually omitted.
`extensionObject`	alternative to the `test, coerce, replace, by` arguments; an object from class `SClassExtension` describing the relation. (Used in internal calls.)
`doComplete`	when `TRUE`, the class definitions will be augmented with indirect relations as well. (Used in internal calls.)
`by`	In a call to `setIs`, the name of an intermediary class. Coercion will proceed by first coercing to this class and from there to the target class. (The intermediate coercions have to be valid.)
`where`	In a call to `setIs`, where to store the metadata defining the relationship. Default is the global environment for calls from the top level of the session or a source file evaluated there. When the call occurs in the top level of a file in the source of a package, the default will be the name space or environment of the package. Other uses are tricky and not usually a good idea, unless you really know what you are doing.
`classDef`	Optional class definition for `class` , required internally when `setIs` is called during the initial definition of the class by a call to `setClass`. *Don't* use this argument, unless you really know why you're doing so.

Summary of Functions

`is`: With two arguments, tests whether `object` can be treated as from `class2`.
With one argument, returns all the super-classes of this object's class.

extends: Does the first class extend the second class? The call returns `maybe` if the extension includes a test.

When called with one argument, the value is a vector of the superclasses of `class1`. If argument `fullInfo` is `TRUE`, the call returns a named list of objects of class `SClassExtension`; otherwise, just the names of the superclasses.

setIs: Defines `class1` to be an extension (subclass) of `class2`. If `class2` is an existing virtual class, such as a class union, then only the two classes need to be supplied in the call, if the implied inherited methods work for `class1`. See the details section below.

Alternatively, arguments `coerce` and `replace` should be supplied, defining methods to coerce to the superclass and to replace the part corresponding to the superclass. As discussed in the details and other sections below, this form is often less recommended than the corresponding call to `setAs`, to which it is an alternative.

Details

Arranging for a class to inherit from another class is a key tool in programming. In R, there are three basic techniques, the first two providing what is called "simple" inheritance, the preferred form:

1. By the `contains=` argument in a call to `setClass`. This is and should be the most common mechanism. It arranges that the new class contains all the structure of the existing class, and in particular all the slots with the same class specified. The resulting class extension is defined to be `simple`, with important implications for method definition (see the section on this topic below).
2. Making `class1` a subclass of a virtual class either by a call to `setClassUnion` to make the subclass a member of a new class union, or by a call to `setIs` to add a class to an existing class union or as a new subclass of an existing virtual class. In either case, the implication should be that methods defined for the class union or other superclass all work correctly for the subclass. This may depend on some similarity in the structure of the subclasses or simply indicate that the superclass methods are defined in terms of generic functions that apply to all the subclasses. These relationships are also generally simple.
3. Supplying `coerce` and `replace` arguments to `setAs`. R allows arbitrary inheritance relationships, using the same mechanism for defining coerce methods by a call to `setAs`. The difference between the two is simply that `setAs` will require a call to `as` for a conversion to take place, whereas after the call to `setIs`, objects will be automatically converted to the superclass.

   The automatic feature is the dangerous part, mainly because it results in the subclass potentially inheriting methods that do not work. See the section on inheritance below. If the two classes involved do not actually inherit a large collection of methods, as in the first example below, the danger may be relatively slight.

   If the superclass inherits methods where the subclass has only a default or remotely inherited method, problems are more likely. In this case, a general recommendation is to use the `setAs` mechanism instead, unless there is a strong counter reason. Otherwise, be prepared to override some of the methods inherited.

With this caution given, the rest of this section describes what happens when `coerce=` and `replace=` argument are supplied to `setIs`.

The `coerce` and `replace` arguments are functions that define how to coerce a `class1` object to `class2`, and how to replace the part of the subclass object that corresponds to `class2`. The first of these is a function of one argument (conventionally `from`) and the second of two arguments (`from`, `value`). For details, see the section on coerce functions below .

When `by` is specified, the coerce process first coerces to this class and then to `class2`. It's unlikely you would use the `by` argument directly, but it is used in defining cached information about classes.

The value returned (invisibly) by `setIs` is the revised class definition of `class1`.

Coerce, replace, and test functions

The `coerce` argument is a function that turns a `class1` object into a `class2` object. The `replace` argument is a function of two arguments that modifies a `class1` object (the first argument) to replace the part of it that corresponds to `class2` (supplied as `value`, the second argument). It then returns the modified object as the value of the call. In other words, it acts as a replacement method to implement the expression `as(object, class2) <- value`.

The easiest way to think of the `coerce` and `replace` functions is by thinking of the case that `class1` contains `class2` in the usual sense, by including the slots of the second class. (To repeat, in this situation you would not call `setIs`, but the analogy shows what happens when you do.)

The `coerce` function in this case would just make a `class2` object by extracting the corresponding slots from the `class1` object. The `replace` function would replace in the `class1` object the slots corresponding to `class2`, and return the modified object as its value.

For additional discussion of these functions, see the documentation of the `setAs` function. (Unfortunately, argument `def` to that function corresponds to argument `coerce` here.)

The inheritance relationship can also be conditional, if a function is supplied as the `test` argument. This should be a function of one argument that returns `TRUE` or `FALSE` according to whether the object supplied satisfies the relation `is(object, class2)`. Conditional relations between classes are slightly deprecated because they cannot be implemented as efficiently as ordinary relations and because they sometimes can lead to confusion (in thinking about what methods are dispatched for a particular function, for example). But they can correspond to distinctions such as two classes that have the same representation, but with only one of them obeying certain additional constraints.

Inherited methods

A method written for a particular signature (classes matched to one or more formal arguments to the function) naturally assumes that the objects corresponding to the arguments can be treated as coming from the corresponding classes. The objects will have all the slots and available methods for the classes.

The code that selects and dispatches the methods ensures that this assumption is correct. If the inheritance was "simple", that is, defined by one or more uses of the `contains=` argument in a call to `setClass`, no extra work is generally needed. Classes are inherited from the superclass, with the same definition.

When inheritance is defined by a general call to `setIs`, extra computations are required. This form of inheritance implies that the subclass does *not* just contain the slots of the superclass, but instead requires the explicit call to the coerce and/or replace method. To ensure correct computation, the inherited method is supplemented by calls to `as` before the body of the method is evaluated.

The calls to `as` generated in this case have the argument `strict = FALSE`, meaning that extra information can be left in the converted object, so long as it has all the appropriate slots. (It's this option that allows simple subclass objects to be used without any change.) When you are writing your coerce method, you may want to take advantage of that option.

Methods inherited through non-simple extensions can result in ambiguities or unexpected selections. If `class2` is a specialized class with just a few applicable methods, creating the inheritance relation may have little effect on the behavior of `class1`. But if `class2` is a class with many methods, you may find that you now inherit some undesirable methods for `class1`, in some cases,

fail to inherit expected methods. In the second example below, the non-simple inheritance from class "factor" might be assumed to inherit S3 methods via that class. But the S3 class is ambiguous, and in fact is "character" rather than "factor".

For some generic functions, methods inherited by non-simple extensions are either known to be invalid or sufficiently likely to be so that the generic function has been defined to exclude such inheritance. For example initialize methods must return an object of the target class; this is straightforward if the extension is simple, because no change is made to the argument object, but is essentially impossible. For this reason, the generic function insists on only simple extensions for inheritance. See the simpleInheritanceOnly argument to setGeneric for the mechanism. You can use this mechanism when defining new generic functions.

If you get into problems with functions that do allow non-simple inheritance, there are two basic choices. Either back off from the setIs call and settle for explicit coercing defined by a call to setAs; or, define explicit methods involving class1 to override the bad inherited methods. The first choice is the safer, when there are serious problems.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

Examples

```
## Two examples of setIs() with coerce= and replace= arguments
## The first one works fairly well, because neither class has many
## inherited methods do be disturbed by the new inheritance

## The second example does NOT work well, because the new superclass,
## "factor", causes methods to be inherited that should not be.

## First example:
## a class definition (see setClass for class "track")
setClass("trackCurve", contains = "track",
         representation( smooth = "numeric"))
## A class similar to "trackCurve", but with different structure
## allowing matrices for the "y" and "smooth" slots
setClass("trackMultiCurve",
         representation(x="numeric", y="matrix", smooth="matrix"),
         prototype = structure(list(), x=numeric(), y=matrix(0,0,0),

                               smooth= matrix(0,0,0)))
## Automatically convert an object from class "trackCurve" into
## "trackMultiCurve", by making the y, smooth slots into 1-column matrices
setIs("trackCurve",
      "trackMultiCurve",
      coerce = function(obj) {
        new("trackMultiCurve",
            x = obj@x,
            y = as.matrix(obj@y),
            smooth = as.matrix(obj@smooth))
      },
      replace = function(obj, value) {
        obj@y <- as.matrix(value@y)
        obj@x <- value@x
```

```
          obj@smooth <- as.matrix(value@smooth)
          obj})


## Second Example:
## A class that adds a slot to "character"
setClass("stringsDated", contains = "character", representation(stamp="POSIXt"))

## Convert automatically to a factor by explicit coerce
setIs("stringsDated", "factor",
   coerce = function(from){factor(from@.Data)},
   replace=function(from, value){
       from@.Data<- as.character(value); from})


ll <- sample(letters, 10, replace = TRUE)
 ld <- new("stringsDated", ll, stamp = Sys.time())

levels(as(ld, "factor"))

levels(ld) # will be NULL--see comment in section on inheritance above.

## In contrast, a class that simply extends "factor" has no such ambiguities

setClass("factorDated", contains = "factor", representation(stamp="POSIXt"))

fd = new("factorDated", factor(ll), stamp = Sys.time())

identical(levels(fd), levels(as(fd, "factor")))
```

`isSealedMethod` *Check for a Sealed Method or Class*

Description

These functions check for either a method or a class that has been *sealed* when it was defined, and which therefore cannot be re-defined.

Usage

```
isSealedMethod(f, signature, fdef, where)
isSealedClass(Class, where)
```

Arguments

`f`	The quoted name of the generic function.
`signature`	The class names in the method's signature, as they would be supplied to `setMethod`.
`fdef`	Optional, and usually omitted: the generic function definition for `f`.
`Class`	The quoted name of the class.

`where` where to search for the method or class definition. By default, searches from the top environment of the call to `isSealedMethod` or `isSealedClass`, typically the global environment or the name space of a package containing a call to one of the functions.

Details

In the R implementation of classes and methods, it is possible to seal the definition of either a class or a method. The basic classes (numeric and other types of vectors, matrix and array data) are sealed. So also are the methods for the primitive functions on those data types. The effect is that programmers cannot re-define the meaning of these basic data types and computations. More precisely, for primitive functions that depend on only one data argument, methods cannot be specified for basic classes. For functions (such as the arithmetic operators) that depend on two arguments, methods can be specified if *one* of those arguments is a basic class, but not if both are.

Programmers can seal other class and method definitions by using the `sealed` argument to `setClass` or `setMethod`.

Value

The functions return `FALSE` if the method or class is not sealed (including the case that it is not defined); `TRUE` if it is.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

Examples

```
## these are both TRUE
isSealedMethod("+", c("numeric", "character"))
isSealedClass("matrix")

setClass("track",
           representation(x="numeric", y="numeric"))
## but this is FALSE
isSealedClass("track")
## and so is this
isSealedClass("A Name for an undefined Class")
## and so are these, because only one of the two arguments is basic
isSealedMethod("+", c("track", "numeric"))
isSealedMethod("+", c("numeric", "track"))
```

---

language-class *Classes to Represent Unevaluated Language Objects*

Description

The virtual class `"language"` and the specific classes that extend it represent unevaluated objects, as produced for example by the parser or by functions such as `quote`.

Usage

```
### each of these classes corresponds to an unevaluated object
### in the S language.
### The class name can appear in method signatures,
### and in a few other contexts (such as some calls to as()).

"("
"<-"
"call"
"for"
"if"
"repeat"
"while"
"name"
"{"

### Each of the classes above extends the virtual class

"language"
```

Objects from the Class

`"language"` is a virtual class; no objects may be created from it.

Objects from the other classes can be generated by a call to `new(Class, ...)`, where `Class` is the quoted class name, and the ... arguments are either empty or a *single* object that is from this class (or an extension).

Methods

coerce `signature(from = "ANY", to = "call")`. A method exists for `as(object, "call")`, calling `as.call()`.

LinearMethodsList-class

Class "LinearMethodsList"

Description

A version of methods lists that has been 'linearized' for producing summary information. The actual objects from class `"MethodsList"` used for method dispatch are defined recursively over the arguments involved.

Objects from the Class

The function `linearizeMlist` converts an ordinary methods list object into the linearized form.

Slots

methods: Object of class "list", the method definitions.

arguments: Object of class "list", the corresponding formal arguments, namely as many of the arguments in the signature of the generic function as are active in the relevant method table.

classes: Object of class "list", the corresponding classes in the signatures.

generic: Object of class "genericFunction"; the generic function to which the methods correspond.

Future Note

The current version of linearizeMlist does not take advantage of the MethodDefinition class, and therefore does more work for less effect than it could. In particular, we may move to redefine both the function and the class to take advantage of the stored signatures. Don't write code depending precisely on the present form, although all the current information will be obtainable in the future.

See Also

Function linearizeMlist for the computation, and class MethodsList for the original, recursive form.

---

makeClassRepresentation

Create a Class Definition

Description

Constructs an object of class classRepresentation to describe a particular class. Mostly a utility function, but you can call it to create a class definition without assigning it, as setClass would do.

Usage

```
makeClassRepresentation(name, slots=list(), superClasses=character(),
                        prototype=NULL, package, validity, access,
                        version, sealed, virtual=NA, where)
```

Arguments

name	character string name for the class
slots	named list of slot classes as would be supplied to setClass, but *without* the unnamed arguments for superClasses if any.
superClasses	what classes does this class extend
prototype	an object providing the default data for the class, e.g, the result of a call to prototype.
package	The character string name for the package in which the class will be stored; see getPackageName.

validity	Optional validity method. See validObject, and the discussion of validity methods in the reference.
access	Access information. Not currently used.
version	Optional version key for version control. Currently generated, but not used.
sealed	Is the class sealed? See setClass.
virtual	Is this known to be a virtual class?
where	The environment from which to look for class definitions needed (e.g., for slots or superclasses). See the discussion of this argument under GenericFunctions.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

setClass

---

method.skeleton *Create a Skeleton File for a New Method*

Description

This function writes a source file containing a call to setMethod to define a method for the generic function and signature supplied. By default the method definition is in line in the call, but can be made an external (previously assigned) function.

Usage

```
method.skeleton(generic, signature, file, external = FALSE, where)
```

Arguments

generic	the character string name of the generic function, or the generic function itself. In the first case, the function need not currently be a generic, as it would not for the resulting call to setMethod.
signature	the method signature, as it would be given to setMethod
file	a character string name for the output file, or a writable connection. By default the generic function name and the classes in the signature are concatenated, with separating underscore characters. The file name should normally end in ".R".
external	flag to control whether the function definition for the method should be a separate external object assigned in the source file, or included in line in the call to setMethod. If supplied as a character string, this will be used as the name for the external function; by default the name concatenates the generic and signature names, with separating underscores.
where	The environment in which to look for the function; by default, the top-level environment of the call to method.skeleton.

Value

The `file` argument, invisibly, but the function is used for its side effect.

See Also

`setMethod`, `package.skeleton`

Examples

```
setClass("track", representation(x ="numeric", y="numeric"))
method.skeleton("show", "track")            ## writes show_track.R
method.skeleton("Ops", c("track", "track")) ## writes "Ops_track_track.R"

## write multiple method skeletons to one file
con <- file("./Math_track.R", "w")
method.skeleton("Math", "track", con)
method.skeleton("exp", "track", con)
method.skeleton("log", "track", con)
close(con)
```

MethodDefinition-class

Classes to Represent Method Definitions

Description

These classes extend the basic class `"function"` when functions are to be stored and used as method definitions.

Details

Method definition objects are functions with additional information defining how the function is being used as a method. The `target` slot is the class signature for which the method will be dispatched, and the `defined` slot the signature for which the method was originally specified (that is, the one that appeared in some call to `setMethod`).

Objects from the Class

The action of setting a method by a call to `setMethod` creates an object of this class. It's unwise to create them directly.

The class `"SealedMethodDefinition"` is created by a call to `setMethod` with argument `sealed = TRUE`. It has the same representation as `"MethodDefinition"`.

Slots

`.Data`: Object of class `"function"`; the data part of the definition.

`target`: Object of class `"signature"`; the signature for which the method was wanted.

`defined`: Object of class `"signature"`; the signature for which a method was found. If the method was inherited, this will not be identical to `target`.

`generic`: Object of class `"character"`; the function for which the method was created.

Extends

Class "function", from data part.
Class "PossibleMethod", directly.
Class "OptionalMethods", by class "function".

See Also

class MethodsList for the objects defining sets of methods associated with a particular generic function. The individual method definitions stored in these objects are from class MethodDefinition, or an extension. Class MethodWithNext for an extension used by callNextMethod.

Methods *General Information on Methods*

Description

This documentation section covers some general topics on how methods work and how the **methods** package interacts with the rest of R. The information is usually not needed to get started with methods and classes, but may be helpful for moderately ambitious projects, or when something doesn't work as expected.

The section "How Methods Work" describes the underlying mechanism; "Method Selection and Dispatch" provides more details on how class definitions determine which methods are used; "Generic Functions" discusses generic functions as objects. For additional information specifically about class definitions, see Classes.

How Methods Work

A generic function has associated with it a collection of other functions (the methods), all of which have the same formal arguments as the generic. See the "Generic Functions" section below for more on generic functions themselves.

Each R package will include methods metadata objects corresponding to each generic function for which methods have been defined in that package. When the package is loaded into an R session, the methods for each generic function are *cached*, that is, stored in the environment of the generic function along with the methods from previously loaded packages. This merged table of methods is used to dispatch or select methods from the generic, using class inheritance and possibly group generic functions (see GroupGenericFunctions) to find an applicable method. See the "Method Selection and Dispatch" section below. The caching computations ensure that only one version of each generic function is visible globally; although different attached packages may contain a copy of the generic function, these behave identically with respect to method selection. In contrast, it is possible for the same function name to refer to more than one generic function, when these have different package slots. In the latter case, R considers the functions unrelated: A generic function is defined by the combination of name and package. See the "Generic Functions" section below.

The methods for a generic are stored according to the corresponding signature in the call to setMethod that defined the method. The signature associates one class name with each of a subset of the formal arguments to the generic function. Which formal arguments are available, and the order in which they appear, are determined by the "signature" slot of the generic function itself. By default, the signature of the generic consists of all the formal arguments except ..., in the order they appear in the function definition.

Trailing arguments in the signature of the generic will be *inactive* if no method has yet been specified that included those arguments in its signature. Inactive arguments are not needed or used in labeling the cached methods. (The distinction does not change which methods are dispatched, but ignoring inactive arguments improves the efficiency of dispatch.)

All arguments in the signature of the generic function will be evaluated when the function is called, rather than using the traditional lazy evaluation rules of S. Therefore, it's important to *exclude* from the signature any arguments that need to be dealt with symbolically (such as the first argument to function `substitute`). Note that only actual arguments are evaluated, not default expressions. A missing argument enters into the method selection as class `"missing"`.

The cached methods are stored in an environment object. The names used for assignment are a concatenation of the class names for the active arguments in the method signature.

Method Selection and Dispatch

When a call to a generic function is evaluated, a method is selected corresponding to the classes of the actual arguments in the signature. First, the cached methods table is searched for an exact match; that is, a method stored under the signature defined by the string value of `class(x)` for each non-missing argument, and `"missing"` for each missing argument. If no method is found directly for the actual arguments in a call to a generic function, an attempt is made to match the available methods to the arguments by using the superclass information about the actual classes.

Each class definition may include a list of one or more *superclasses* of the new class. The simplest and most common specification is by the `contains=` argument in the call to `setClass`. Each class named in this argument is a superclass of the new class. The S language has two additional mechanisms for defining superclasses. A call to `setIs` can create an inheritance relationship that is not the simple one of containing the superclass representation in the new class. In this case, explicit methods are defined to relate the subclass and the superclass. Also, a call to `setClassUnion` creates a union class that is a superclass of each of the members of the union. All three mechanisms are treated equivalently for purposes of method selection: they define the *direct* superclasses of a particular class. For more details on the mechanisms, see `Classes`.

The direct superclasses themselves may have superclasses, defined by any of the same mechanisms, and similarly for further generations. Putting all this information together produces the full list of superclasses for this class. The superclass list is included in the definition of the class that is cached during the R session. Each element of the list describes the nature of the relationship (see `SClassExtension` for details). Included in the element is a `distance` slot giving a numeric distance between the two classes. The distance currently is the path length for the relationship: `1` for direct superclasses (regardless of which mechanism defined them), then `2` for the direct superclasses of those classes, and so on. In addition, any class implicitly has class `"ANY"` as a superclass. The distance to `"ANY"` is treated as larger than the distance to any actual class. The special class `"missing"` corresponding to missing arguments has only `"ANY"` as a superclass, while `"ANY"` has no superclasses.

The information about superclasses is summarized when a class definition is printed.

When a method is to be selected by inheritance, a search is made in the table for all methods directly corresponding to a combination of either the direct class or one of its superclasses, for each argument in the active signature. For an example, suppose there is only one argument in the signature and that the class of the corresponding object was `"dgeMatrix"` (from the `Matrix` package on CRAN). This class has two direct superclasses and through these 4 additional superclasses. Method selection finds all the methods in the table of directly specified methods labeled by one of these classes, or by `"ANY"`.

When there are multiple arguments in the signature, each argument will generate a similar list of inherited classes. The possible matches are now all the combinations of classes from each argument (think of the function `outer` generating an array of all possible combinations). The search now

finds all the methods matching any of this combination of classes. The computation of distances also has to combine distances for the individual arguments. There are many ways to combine the distances; the current implementation simply adds them. The result of the search is then a list of zero, one, or more methods, and a parallel vector of distances between the target signature and the available methods.

If the list has more than one matching method, only those corresponding to the minimum distance are considered. There may still be multiple best methods. The dispatch software considers this an ambiguous case and warns the user (only on the first call for this selection). The method occurring first in the list of superclasses is selected. By the mechanism of producing the extension information, this orders the direct superclasses by the order they appeared in the original call to `setClass`. Classes specified in `setIs` and `setClassUnion` calls, and by the superclasses of these classes. (Note that only the ordering of classes within a particular generation of superclasses counts, because only these will have the same distance). It is generally a very bad idea to count on any observed ordering, other than of the simple superclasses, since both circumstances and future changes to the computations could alter such orderings.

All this detail about selection is less important than the realization that having ambiguous method selection usually means that you need to be more specific about intentions. It is likely that some consideration other than the ordering of superclasses in the class definition is more important in determining which method *should* be selected, and the preference may well be different for different generic functions. Where ambiguities arise, the best approach is usually to provide a specific method for the subclass.

When the inherited method has been selected, the selection is cached in the generic function so that future calls with the same class will not require repeating the search. Cached inherited selections are not themselves used in future inheritance searches, since that could result in invalid selections. If you want inheritance computations to be done again (for example, because a newly loaded package has a more direct method than one that has already been used in this session), call `resetGeneric`. Because classes and methods involving them tend to come from the same package, the current implementation does not reset all generics every time a new package is loaded.

Besides being initiated through calls to the generic function, method selection can be done explicitly by calling the function `selectMethod`.

Once a method has been selected, the evaluator creates a new context in which a call to the method is evaluated. The context is initialized with the arguments from the call to the generic function. These arguments are not rematched. All the arguments in the signature of the generic will have been evaluated (including any that are currently inactive); arguments that are not in the signature will obey the usual lazy evaluation rules of the language. If an argument was missing in the call, its default expression if any will *not* have been evaluated, since method dispatch always uses class `missing` for such arguments.

A call to a generic function therefore has two contexts: one for the function and a second for the method. The argument objects will be copied to the second context, but not any local objects created in a nonstandard generic function. The other important distinction is that the parent ("enclosing") environment of the second context is the environment of the method as a function, so that all R programming techniques using such environments apply to method definitions as ordinary functions.

For further discussion of method selection and dispatch, see the first reference.

Generic Functions

In principle, a generic function could be any function that evaluates a call to `standardGeneric()`, the internal function that selects a method and evaluates a call to the selected method. In practice, generic functions are special objects that in addition to being from a subclass of class `"function"` also extend the class `genericFunction`. Such objects

have slots to define information needed to deal with their methods. They also have specialized environments, containing the tables used in method selection.

The slots `"generic"` and `"package"` in the object are the character string names of the generic function itself and of the package from which the function is defined. As with classes, generic functions are uniquely defined in R by the combination of the two names. There can be generic functions of the same name associated with different packages (although inevitably keeping such functions cleanly distinguished is not always easy). On the other hand, R will enforce that only one definition of a generic function can be associated with a particular combination of function and package name, in the current session or other active version of R.

Tables of methods for a particular generic function, in this sense, will often be spread over several other packages. The total set of methods for a given generic function may change during a session, as additional packages are loaded. Each table must be consistent in the signature assumed for the generic function.

R distinguishes *standard* and *nonstandard* generic functions, with the former having a function body that does nothing but dispatch a method. For the most part, the distinction is just one of simplicity: knowing that a generic function only dispatches a method call allows some efficiencies and also removes some uncertainties.

In most cases, the generic function is the visible function corresponding to that name, in the corresponding package. There are two exceptions, *implicit* generic functions and the special computations required to deal with R's *primitive* functions. Packages can contain a table of implicit generic versions of functions in the package, if the package wishes to leave a function non-generic but to constrain what the function would be like if it were generic. Such implicit generic functions are created during the installation of the package, essentially by defining the generic function and possibly methods for it, and then reverting the function to its non-generic form. (See implicitGeneric for how this is done.) The mechanism is mainly used for functions in the older packages in R, which may prefer to ignore S4 methods. Even in this case, the actual mechanism is only needed if something special has to be specified. All functions have a corresponding implicit generic version defined automatically (an implicit, implicit generic function one might say). This function is a standard generic with the same arguments as the non-generic function, with the non-generic version as the default (and only) method, and with the generic signature being all the formal arguments except `...`.

The implicit generic mechanism is needed only to override some aspect of the default definition. One reason to do so would be to remove some arguments from the signature. Arguments that may need to be interpreted literally, or for which the lazy evaluation mechanism of the language is needed, must *not* be included in the signature of the generic function, since all arguments in the signature will be evaluated in order to select a method. For example, the argument `expr` to the function `with` is treated literally and must therefore be excluded from the signature.

One would also need to define an implicit generic if the existing non-generic function were not suitable as the default method. Perhaps the function only applies to some classes of objects, and the package designer prefers to have no general default method. In the other direction, the package designer might have some ideas about suitable methods for some classes, if the function were generic. With reasonably modern packages, the simple approach in all these cases is just to define the function as a generic. The implicit generic mechanism is mainly attractive for older packages that do not want to require the methods package to be available.

Generic functions will also be defined but not obviously visible for functions implemented as *primitive* functions in the base package. Primitive functions look like ordinary functions when printed but are in fact not function objects but objects of two types interpreted by the R evaluator to call underlying C code directly. Since their entire justification is efficiency, R refuses to hide primitives behind a generic function object. Methods may be defined for most primitives, and corresponding metadata objects will be created to store them. Calls to the primitive still go directly to the C code,

which will sometimes check for applicable methods. The definition of "sometimes" is that methods must have been detected for the function in some package loaded in the session and `isS4(x)` is `TRUE` for the first argument (or for the second argument, in the case of binary operators). You can test whether methods have been detected by calling `isGeneric` for the relevant function and you can examine the generic function by calling `getGeneric`, whether or not methods have been detected. For more on generic functions, see the first reference and also section 2 of *R Internals*.

Method Definitions

All method definitions are stored as objects from the `MethodDefinition` class. Like the class of generic functions, this class extends ordinary R functions with some additional slots: `"generic"`, containing the name and package of the generic function, and two signature slots, `"defined"` and `"target"`, the first being the signature supplied when the method was defined by a call to `setMethod`. The `"target"` slot starts off equal to the `"defined"` slot. When an inherited method is cached after being selected, as described above, a copy is made with the appropriate `"target"` signature. Output from `showMethods`, for example, includes both signatures.

Method definitions are required to have the same formal arguments as the generic function, since the method dispatch mechanism does not rematch arguments, for reasons of both efficiency and consistency.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version: see section 10.6 for method selection and section 10.5 for generic functions).

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

For more specific information, see `setGeneric`, `setMethod`, and `setClass`.

For the use of ... in methods, see dotsMethods.

`MethodsList-class` *Class MethodsList, Representation of Methods for a Generic Function*

Description

Objects from this class are generated and revised by the definition of methods for a generic function.

Details

Suppose a function `f` has formal arguments `x` and `y`. The methods list object for that function has the object `as.name("x")` as its `argument` slot. An element of the methods named `"track"` is selected if the actual argument corresponding to `x` is an object of class `"track"`. If there is such an element, it can generally be either a function or another methods list object.

In the first case, the function defines the method to use for any call in which `x` is of class `"track"`. In the second case, the new methods list object defines the available methods depending on the remaining formal arguments, in this example, `y`.

Each method corresponds conceptually to a *signature*; that is a named list of classes, with names corresponding to some or all of the formal arguments. In the previous example, if selecting class `"track"` for `x`, finding that the selection was another methods list and then selecting class

"numeric" for y would produce a method associated with the signature x = "track", y = "numeric".

You can see the methods arranged by signature by calling the function showMethods, A methods list can be converted into an ordinary list with the methods arranged this way (in two different forms) by calling the functions listFromMlist and linearizeMlist.

Slots

argument: Object of class "name". The name of the argument being used for dispatch at this level.

methods: A named list of the methods (and method lists) defined *explicitly* for this argument. The names are the names of classes, and the corresponding element defines the method or methods to be used if the corresponding argument has that class. See the details below.

allMethods: A named list, contains all the directly defined methods from the methods slot, plus any inherited methods. Ignored when methods tables are used for dispatch (see Methods

Extends

Class "OptionalMethods", directly.

MethodWithNext-class

Class MethodWithNext

Description

Class of method definitions set up for callNextMethod

Objects from the Class

Objects from this class are generated as a side-effect of calls to callNextMethod.

Slots

.Data: Object of class "function"; the actual function definition.

nextMethod: Object of class "PossibleMethod" the method to use in response to a callNextMethod() call.

excluded: Object of class "list"; one or more signatures excluded in finding the next method.

target: Object of class "signature", from class "MethodDefinition"

defined: Object of class "signature", from class "MethodDefinition"

generic: Object of class "character"; the function for which the method was created.

Extends

Class "MethodDefinition", directly.
Class "function", from data part.
Class "PossibleMethod", by class "MethodDefinition".
Class "OptionalMethods", by class "MethodDefinition".

Methods

findNextMethod `signature(method = "MethodWithNext")`: used internally by method dispatch.

loadMethod `signature(method = "MethodWithNext")`: used internally by method dispatch.

show `signature(object = "MethodWithNext")`

See Also

`callNextMethod`, and class `MethodDefinition`.

`new` *Generate an Object from a Class*

Description

Given the name or the definition of a class, plus optionally data to be included in the object, `new` returns an object from that class.

Usage

```
new(Class, ...)

initialize(.Object, ...)
```

Arguments

`Class`	Either the name of a class (the usual case) or the object describing the class (e.g., the value returned by `getClass`).
`...`	Data to include in the new object. Named arguments correspond to slots in the class definition. Unnamed arguments must be objects from classes that this class extends.
`.Object`	An object: see the Details section.

Details

The function `new` begins by copying the prototype object from the class definition. Then information is inserted according to the `...` arguments, if any. As of version 2.4 of R, the type of the prototype object, and therefore of all objects returned by `new()`, is `"S4"` except for classes that extend one of the basic types, where the prototype has that basic type. User functions that depend on `typeof(object)` should be careful to handle `"S4"` as a possible type.

The interpretation of the `...` arguments can be specialized to particular classes, if an appropriate method has been defined for the generic function `"initialize"`. The `new` function calls `initialize` with the object generated from the prototype as the `.Object` argument to `initialize`.

By default, unnamed arguments in the `...` are interpreted as objects from a superclass, and named arguments are interpreted as objects to be assigned into the correspondingly named slots. Thus, explicit slots override inherited information for the same slot, regardless of the order in which the arguments appear.

The initialize methods do not have to have ... as their second argument (see the examples). Initialize methods are often written when the natural parameters describing the new object are not the names of the slots. If you do define such a method, note the implications for future subclasses of your class. If these have additional slots, and your initialize method has ... as a formal argument, then your method should pass such arguments along via callNextMethod. If your method does not have this argument, then either a subclass must have its own method or else the added slots must be specified by users in some way other than as arguments to new.

For examples of initialize methods, see initialize-methods for existing methods for classes "traceable" and "environment", among others.

Methods for initialize can be inherited only by simple inheritance, since it is a requirement that the method return an object from the target class. See the simpleInheritanceOnly argument to setGeneric and the discussion in setIs for the general concept.

Note that the basic vector classes, "numeric", etc. are implicitly defined, so one can use new for these classes.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

Classes for an overview of defining class, and setOldClass for the relation to S3 classes.

Examples

```
## using the definition of class "track" from Classes



## a new object with two slots specified
t1 <- new("track", x = seq_along(ydata), y = ydata)

# a new object including an object from a superclass, plus a slot
t2 <- new("trackCurve", t1, smooth = ysmooth)

### define a method for initialize, to ensure that new objects have
### equal-length x and y slots.

setMethod("initialize",
          "track",
          function(.Object, x = numeric(0), y = numeric(0)) {
            if(nargs() > 1) {
              if(length(x) != length(y))
                stop("specified x and y of different lengths")
              .Object@x <- x
              .Object@y <- y
            }
            .Object
          })

### the next example will cause an error (x will be numeric(0)),
### because we didn't build in defaults for x,
```

```
### although we could with a more elaborate method for initialize

try(new("track", y = sort(stats::rnorm(10))))

## a better way to implement the previous initialize method.
## Why?  By using callNextMethod to call the default initialize method
## we don't inhibit classes that extend "track" from using the general
## form of the new() function.  In the previous version, they could only
## use x and y as arguments to new, unless they wrote their own
## initialize method.

setMethod("initialize", "track", function(.Object, ...) {
    .Object <- callNextMethod()
    if(length(.Object@x) != length(.Object@y))
     stop("specified x and y of different lengths")
    .Object
  })
```

`ObjectsWithPackage-class`
A Vector of Object Names, with associated Package Names

Description

This class of objects is used to represent ordinary character string object names, extended with a `package` slot naming the package associated with each object.

Objects from the Class

The function `getGenerics` returns an object of this class.

Slots

`.Data`: Object of class `"character"`: the object names.

`package`: Object of class `"character"` the package names.

Extends

Class `"character"`, from data part.
Class `"vector"`, by class `"character"`.

See Also

`Methods` for general background.

promptClass *Generate a Shell for Documentation of a Formal Class*

Description

Assembles all relevant slot and method information for a class, with minimal markup for Rd processing; no QC facilities at present.

Usage

```
promptClass(clName, filename = NULL, type = "class",
            keywords = "classes", where = topenv(parent.frame()))
```

Arguments

clName	a character string naming the class to be documented.
filename	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is the topic name for the class documentation, followed by ".Rd". Can also be NA (see below).
type	the documentation type to be declared in the output file.
keywords	the keywords to include in the shell of the documentation. The keyword "classes" should be one of them.
where	where to look for the definition of the class and of methods that use it.

Details

The class definition is found on the search list. Using that definition, information about classes extended and slots is determined.

In addition, the currently available generics with methods for this class are found (using getGenerics). Note that these methods need not be in the same environment as the class definition; in particular, this part of the output may depend on which packages are currently in the search list.

As with other prompt-style functions, unless filename is NA, the documentation shell is written to a file, and a message about this is given. The file will need editing to give information about the *meaning* of the class. The output of promptClass can only contain information from the metadata about the formal definition and how it is used.

If filename is NA, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to cat(unlist(x), file = filename, sep = "\n"), where x is the list-style representation.

Value

If filename is NA, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

Author(s)

VJ Carey ⟨stvjc@channing.harvard.edu⟩ and John Chambers

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`prompt` for documentation of functions, `promptMethods` for documentation of method definitions.

For processing of the edited documentation, either use `R CMD Rdconv`, or include the edited file in the 'man' subdirectory of a package.

Examples

```
## Not run:
> promptClass("track")
A shell of class documentation has been written to the
file "track-class.Rd".
## End(Not run)
```

---

promptMethods *Generate a Shell for Documentation of Formal Methods*

Description

Generates a shell of documentation for the methods of a generic function.

Usage

```
promptMethods(f, filename = NULL, methods)
```

Arguments

`f`	a character string naming the generic function whose methods are to be documented.
`filename`	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to the coded topic name for these methods (currently, `f` followed by `"-methods.Rd"`). Can also be `FALSE` or `NA` (see below).
`methods`	Optional methods list object giving the methods to be documented. By default, the first methods object for this generic is used (for example, if the current global environment has some methods for `f`, these would be documented). If this argument is supplied, it is likely to be `getMethods(f, where)`, with `where` some package containing methods for `f`.

Details

If filename is FALSE, the text created is returned, presumably to be inserted some other documentation file, such as the documentation of the generic function itself (see prompt).

If filename is NA, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to cat(unlist(x), file = filename, sep = "\n"), where x is the list-style representation.

Otherwise, the documentation shell is written to the file specified by filename.

Value

If filename is FALSE, the text generated; if filename is NA, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

prompt and promptClass

---

representation *Construct a Representation or a Prototype for a Class Definition*

Description

In calls to setClass, these two functions construct, respectively, the representation and prototype arguments. They do various checks and handle special cases. You're encouraged to use them when defining classes that, for example, extend other classes as a data part or have multiple superclasses, or that combine extending a class and slots.

Usage

```
representation(...)
prototype(...)
```

Arguments

...	The call to representation takes arguments that are single character strings. Unnamed arguments are classes that a newly defined class extends; named arguments name the explicit slots in the new class, and specify what class each slot should have. In the call to prototype, if an unnamed argument is supplied, it unconditionally forms the basis for the prototype object. Remaining arguments are taken to correspond to slots of this object. It is an error to supply more than one unnamed argument.

Details

The `representation` function applies tests for the validity of the arguments. Each must specify the name of a class.

The classes named don't have to exist when `representation` is called, but if they do, then the function will check for any duplicate slot names introduced by each of the inherited classes.

The arguments to `prototype` are usually named initial values for slots, plus an optional first argument that gives the object itself. The unnamed argument is typically useful if there is a data part to the definition (see the examples below).

Value

The value pf `representation` is just the list of arguments, after these have been checked for validity.

The value of `prototype` is the object to be used as the prototype. Slots will have been set consistently with the arguments, but the construction does *not* use the class definition to test validity of the contents (it hardly can, since the prototype object is usually supplied to create the definition).

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`setClass`

Examples

```
## representation for a new class with a directly define slot "smooth"
## which should be a "numeric" object, and extending class "track"
representation("track", smooth ="numeric")


setClass("Character",representation("character"))
setClass("TypedCharacter",representation("Character",type="character"),
          prototype(character(0),type="plain"))
ttt <- new("TypedCharacter", "foo", type = "character")


setClass("num1", representation(comment = "character"),
         contains = "numeric",
         prototype = prototype(pi, comment = "Start with pi"))
```

S3Part *S3-style Objects and S4-class Objects*

Description

Old-style (S3) classes may be registered as S4 classes (by calling setOldClass, and many have been. These classes can then be contained in (that is, superclasses of) regular S4 classes, allowing formal methods and slots to be added to the S3 behavior. The function S3Part extracts or replaces the S3 part of such an object and S3Class extracts or replaces the S3-style class. Most computations that need to deal with the S3 aspects can coerce the objects automatically, without direct calls to these functions. The functions also work for classes that extend basic vector or matrix object types. See the details below.

Also discussed are S3 <-> S4 coercion; see the section "S3 and S4 objects"

Usage

```
S3Part(object, strictS3 = FALSE, S3Class)

S3Part(object, strictS3 = FALSE, needClass = ) <- value

S3Class(object)

S3Class(object) <-  value

isXS3Class(classDef)

slotsFromS3(object)
```

Arguments

object	An object from some class that extends a registered S3 class, usually because the class has as one of its superclasses an S3 class registered by a call to setOldClass, or from a class that extends a basic vector, matrix or array object type. See the details. For most of the functions, an S3 object can also be supplied, with the interpretation that it is its own S3 part.
strictS3	If TRUE, the value returned by S3Part will be an S3 object, with all the S4 slots removed. Otherwise, an S4 object will always be returned; for example, from the S4 class created by setOldClass as a proxy for an S3 class, rather than the underlying S3 object.
S3Class	The character vector to be stored as the S3 class slot in the object returned. Usually, and by default, retains the slot from object.
needClass	Require that the replacement value be this class or a subclass of it.
value	For S3Part<-, the replacement value for the S3 part of the object. This does *not* need to be an S4 object; in fact, the usual way to create objects from these classes is by giving an S3 object of the right class as an argument to new. For S3Class<-, the character vector to assign as the S3 class.
classDef	A class definition object, as returned by getClass.

Details

Classes that register S3 classes by a call to setOldClass have slot ".S3Class" to hold the corresponding S3 vector of class strings. The prototype of such a class has the value for this slot determined by the argument to setOldClass.

As a result, new S4 classes that extend (contain) such registered S3 classes also have the same slot, and by default the prototype has the value determined by the contains= argument to setClass. Individual objects from the S4 class may have an S3 class corresponding to the value in the prototype or to an (S3) subclass of that value. See the examples below.

S3Part() with strictS3 = TRUE constructs the underlying S3 object by eliminating all the formally defined slots and turning off the S4 bit of the object. With strictS3 = FALSE the object returned is from the corresponding S4 class. For consistency and generality, S3Part() works also for classes that extend the basic vector, matrix and array classes. Since R is somewhat arbitrary about what it treats as an S3 class ("ts" is, but "matrix" is not), S3Part() tries to return an S3 (that is, non-S4) object whenever the S4 class has a suitable superclass, of either S3 or basic object type.

One general application that relies on this generality is to use S3Part() to get a superclass object that is guaranteed not to be an S4 object. If you are calling some function that checks for S4 objects, you need to be careful not to end up in a closed loop (fooS4 calls fooS3, which checks for an S4 object and calls fooS4 again, maybe indirectly). Using S3Part() with strictS3 = TRUE is a mechanism to avoid such loops.

Because the contents of S3 class objects have no definition or guarantee, the computations involving S3 parts do *not* check for slot validity. Slots are implemented internally in R as attributes, which are copied when present in the S3 part. For this reason, grave problems can occur if an S4 class extending an S3 class uses the name of an S3 attribute as the name of an S4 slot. It's tempting to "promote" an attribute to a slot, but the resulting confusion between slot and attribute is hard to handle.

Frequently, S3Part can and should be avoided by simply coercing objects to the desired class; methods are automatically defined to deal correctly with the slots when as is called to extract or replace superclass objects.

The function slotsFromS3() is a generic function used internally to access the slots associated with the S3 part of the object. Methods for this function are created automatically when setOldClass is called with the S4Class argument. Usually, there is only one S3 slot, containing the S3 class, but the S4Class argument may provide additional slots, in the case that the S3 class has some guaranteed attributes that can be used as formal S4 slots. See the corresponding section in the documentation of setOldClass.

Value

S3Part: Returns or sets the S3 information (and possibly some S4 slots as well, depending on arguments S3Class and keepSlots). See the discussion of argument strict above. If it is TRUE the value returned is an S3 object.

S3Class: For an object from a class extending S3 classes, returns or sets the character vector of class(es) stored in the object. Note that S3Class on any other object returns the value of class.

isXS3Class: Returns TRUE or FALSE according to whether the class defined by ClassDef extends S3 classes (specifically, whether it has the slot for holding the S3 class).

slotsFromS3: returns a list of the relevant slot classes, or an empty list for any other object.

S3 and S4 Objects: Conversion Mechanisms

Objects in R have an internal bit that indicates whether or not to treat the object as coming from an S4 class. This bit is tested by `isS4` and can be set on or off by `asS4`. The latter function, however, does no checking or interpretation; you should only use it if you are very certain every detail has been handled correctly.

As a friendlier alternative, methods have been defined for coercing to the virtual classes `"S3"` and `"S4"`. The expressions `as(object, "S3")` and `as(object, "S4")` return S3 and S4 objects, respectively. In addition, they attempt to do conversions in a valid way, and also check validity when coercing to S4.

The expression `as(object, "S3")` can be used in two ways. For objects from one of the registered S3 classes, the expression will ensure that the class attribute is the full multi-string S3 class implied by `class(object)`. If the registered class has known attribute/slots, these will also be provided.

Another use of `as(object, "S3")` is to take an S4 object and turn it into an S3 object with corresponding attributes. This is only meaningful with S4 classes that have a data part. If you want to operate on the object without invoking S4 methods, this conversion is usually the safest way.

The expression `as(object, "S4")` will use the attributes in the object to create an object from the S4 definition of `class(object)`. This is a general mechanism to create partially defined version of S4 objects via S3 computations (not much different from invoking `new` with corresponding arguments, but usable in this form even if the S4 object has an initialize method with different arguments).

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version).

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`setOldClass`

Examples

```
## two examples extending S3 class "lm", class "xlm"  directly and "ylm" indirectly
setClass("xlm", representation(eps = "numeric"), contains = "lm")
setClass("ylm", representation(header = "character"), contains = "xlm")

## lm.D9 is as computed in the example for stats::lm
y1 = new("ylm", lm.D9, header = "test", eps = .1)
xx = new("xlm", lm.D9, eps =.1)
y2 = new("ylm", xx, header = "test")
stopifnot(inherits(y2, "lm"))
stopifnot(identical(y1, y2))
stopifnot(identical(S3Part(y1, strict = TRUE), lm.D9))

## note the these classes can insert an S3 subclass of "lm" as the S3 part:
myData <- data.frame(time = 1:10, y = (1:10)^.5)
myLm <- lm(cbind(y, y^3)  ~ time, myData) # S3 class: c("mlm", "lm")
ym1 = new("ylm", myLm, header = "Example", eps = 0.)

##similar classes to "xlm" and "ylm", but extending S3 class c("mlm", "lm")
setClass("xmm", representation(eps = "numeric"), contains = "mlm")
```

```
setClass("ymm", representation(header="character"), contains = "xmm")

ym2 <- new("ymm", myLm, header = "Example2", eps = .001)

# but for class "ymm", an S3 part of class "lm" is an error:
try(new("ymm", lm.D9, header = "Example2", eps = .001))

setClass("dataFrameD", representation(date = "Date"), contains = "data.frame")
myDD <- new("dataFrameD", myData, date = Sys.Date())

## S3Part() applied to classes with a data part (.Data slot)

setClass("NumX", contains="numeric", representation(id="character"))
nn = new("NumX", 1:10, id="test")
stopifnot(identical(1:10, S3Part(nn, strict = TRUE)))

m1 = cbind(group, weight)
setClass("MatX", contains = "matrix", representation(date = "Date"))
mx1 = new("MatX", m1, date = Sys.Date())
stopifnot(identical(m1, S3Part(mx1, strict = TRUE)))
```

`S4groupGeneric` *Group Generic Functions*

Description

Methods can be defined for *group generic functions*. Each group generic function has a number of *member* generic functions associated with it.

Methods defined for a group generic function cause the same method to be defined for each member of the group, but a method explicitly defined for a member of the group takes precedence over a method defined, with the same signature, for the group generic.

The functions shown in this documentation page all reside in the **methods** package, but the mechanism is available to any programmer, by calling `setGroupGeneric`.

Usage

```
## S4 group generics:
Arith(e1, e2)
Compare(e1, e2)
Ops(e1, e2)
Logic(e1, e2)
Math(x)
Math2(x, digits)
Summary(x, ..., na.rm = FALSE)
Complex(z)
```

Arguments

`x, z, e1, e2`	objects.
`digits`	number of digits to be used in `round` or `signif`.

`...`	further arguments passed to or from methods.
`na.rm`	logical: should missing values be removed?

Details

Methods can be defined for the group generic functions by calls to `setMethod` in the usual way. Note that the group generic functions should never be called directly – a suitable error message will result if they are. When metadata for a group generic is loaded, the methods defined become methods for the members of the group, but only if no method has been specified directly for the member function for the same signature. The effect is that group generic definitions are selected before inherited methods but after directly specified methods. For more on method selection, see `Methods`.

There are also S3 groups `Math`, `Ops`, `Summary` and `Complex`, see `?S3groupGeneric`, with no corresponding R objects, but these are irrelevant for S4 group generic functions.

The members of the group defined by a particular generic can be obtained by calling `getGroupMembers`. For the group generic functions currently defined in this package the members are as follows:

`Arith` `"+", "-", "*", "^", "%%", "%/%", "/"`

`Compare` `"==", ">", "<", "!=", "<=", ">="`

`Logic` `"&", "|".`

`Ops` `"Arith", "Compare", "Logic"`

`Math` `"abs", "sign", "sqrt", "ceiling", "floor", "trunc", "cummax", "cummin", "cumprod", "cumsum", "log", "log10", "log2", "log1p", "acos", "acosh", "asin", "asinh", "atan", "atanh", "exp", "expm1", "cos", "cosh", "sin", "sinh", "tan", "tanh", "gamma", "lgamma", "digamma", "trigamma"`

`Math2` `"round", "signif"`

`Summary` `"max", "min", "range", "prod", "sum", "any", "all"`

`Complex` `"Arg", "Conj", "Im", "Mod", "Re"`

Note that `Ops` merely consists of three sub groups.

All the functions in these groups (other than the group generics themselves) are basic functions in R. They are not by default S4 generic functions, and many of them are defined as primitives. However, you can still define formal methods for them, both individually and via the group generics. It all works more or less as you might expect, admittedly via a bit of trickery in the background. See Methods for details.

Note that two members of the `Math` group, `log` and `trunc`, have ... as an extra formal argument. Since methods for `Math` will have only one formal argument, you must set a specific method for these functions in order to call them with the extra argument(s).

For further details about group generic functions see section 10.5 of *Software for Data Analysis*.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version).

See Also

The function `callGeneric` is nearly always relevant when writing a method for a group generic. See the examples below and in section 10.5 of *Software for Data Analysis*.

See S3groupGeneric for S3 group generics.

Examples

```
setClass("testComplex", representation(zz = "complex"))
## method for whole group "Complex"
setMethod("Complex", "testComplex",
          function(z) c("groupMethod", callGeneric(z@zz)))
## exception for Arg() :
setMethod("Arg", "testComplex",
          function(z) c("ArgMethod", Arg(z@zz)))
z1 <- 1+2i
z2 <- new("testComplex", zz = z1)
stopifnot(identical(Mod(z2), c("groupMethod", Mod(z1))))
stopifnot(identical(Arg(z2), c("ArgMethod", Arg(z1))))
```

SClassExtension-class

Class to Represent Inheritance (Extension) Relations

Description

An object from this class represents a single 'is' relationship; lists of these objects are used to represent all the extensions (superclasses) and subclasses for a given class. The object contains information about how the relation is defined and methods to coerce, test, and replace correspondingly.

Objects from the Class

Objects from this class are generated by `setIs`, from direct calls and from the `contains=` information in a call to `setClass`, and from class unions created by `setClassUnion`. In the last case, the information is stored in defining the *subclasses* of the union class (allowing unions to contain sealed classes).

Slots

`subClass, superClass:` The classes being extended: corresponding to the `from`, and `to` arguments to `setIs`.

`package:` The package to which that class belongs.

`coerce:` A function to carry out the as() computation implied by the relation. Note that these functions should *not* be used directly. They only deal with the `strict=TRUE` calls to the `as` function, with the full method constructed from this mechanically.

`test:` The function that would test whether the relation holds. Except for explicitly specified `test` arguments to `setIs`, this function is trivial.

`replace:` The method used to implement `as(x, Class) <- value`.

simple: A "logical" flag, TRUE if this is a simple relation, either because one class is contained in the definition of another, or because a class has been explicitly stated to extend a virtual class. For simple extensions, the three methods are generated automatically.

by: If this relation has been constructed transitively, the first intermediate class from the subclass.

dataPart: A "logical" flag, TRUE if the extended class is in fact the data part of the subclass. In this case the extended class is a basic class (i.e., a type).

distance: The distance between the two classes, 1 for directly contained classes, plus the number of generations between otherwise.

Methods

No methods defined with class "SClassExtension" in the signature.

See Also

is, as, and the classRepresentation class.

---

setClass *Create a Class Definition*

Description

Create a class definition, specifying the representation (the slots) and/or the classes contained in this one (the superclasses), plus other optional details.

Usage

```
setClass(Class, representation, prototype, contains=character(),
         validity, access, where, version, sealed, package)
```

Arguments

Class	character string name for the class.
representation	a named list of the slots that the new class should have, the names giving the names of the slots and the corresponding elements being the character string names of the corresponding classes. Usually a call to the representation function. Backward compatibility and compatibility with S-Plus allows unnamed elements for superclasses, but the recommended style is to use the contains= argument instead.
prototype	an object providing the default data for the slots in this class. Usually and preferably the result of a call to prototype.
contains	what classes does this class extend? (These are called *superclasses* in some languages.) When these classes have slots, all their slots will be contained in the new class as well.
where	the environment in which to store or remove the definition. Defaults to the top-level environment of the calling function (the global environment for ordinary computations, and the environment or name space of a package in the source code for that package).

`validity` if supplied, should be a validity-checking method for objects from this class (a function that returns `TRUE` if its argument is a valid object of this class and one or more strings describing the failures otherwise). See `validObject` for details.

`access, version` access and version, included for compatibility with S-Plus, but currently ignored.

`sealed` if `TRUE`, the class definition will be sealed, so that another call to `setClass` will fail on this class name.

`package` an optional package name for the class. By default (and usually) the name of the package in which the class definition is assigned.

Basic Use: Slots and Inheritance

The two essential arguments, other than the class name are `representation` and `contains`, defining the explicit slots and the inheritance (superclasses). Together, these arguments define all the information in an object from this class; that is, the names of all the slots and the classes required for each of them.

The name of the class determines which methods apply directly to objects from this class. The inheritance information specifies which methods apply indirectly, through inheritance. See Methods.

The slots in a class definition will be the union of all the slots specified directly by `representation` and all the slots in all the contained classes. There can only be one slot with a given name; specifically, the direct and inherited slot names must be unique. That does not, however, prevent the same class from being inherited via more than one path.

One kind of element in the `contains=` argument is special, specifying one of the R vector types or one of a few other special R types (`matrix` and `array`). A class can contain at most one of these, directly or indirectly. When it does, that contained class determines the "data part" of the class, and objects from the new class inherit all the built in behavior of the contained type. Two direct consequences are: for any object `x` from the class, `typeof(x)` will be the contained basic type; and a special pseudo-slot, `.Data`, will be shown with the corresponding class.

For an object from any class that does *not* contain such a type, `typeof(x)` will be `"S4"` and the class will have no `.Data` slot.

Prototypes

Defining new classes that inherit from ('extend') other classes is a powerful technique, but has to be used carefully and not over-used. Otherwise, you will often get unintended results when you start to compute with objects from the new class.

As shown in the examples below, the simplest and safest form of inheritance is to start with an explicit class, with some slots, that does not extend anything else. It only does what we say it does.

Then extensions will add some new slots and new behavior.

Another variety of extension starts with one of the built-in objects, perhaps with the intention of modifying R's standard behavior for that class. In this case, the new class inherits the built-in data type as its 'data' part. See the `"numWithId"` example below.

When such a class definition is printed, the data part shows up as a pseudo-slot named `.Data`.

Classes and Packages

Class definitions normally belong to packages (but can be defined in the global environment as well, by evaluating the expression on the command line or in a file sourced from the command line). The corresponding package name is part of the class definition; that is, part of the

`classRepresentation` object holding that definition. Thus, two classes with the same name can exist in different packages, for most purposes.

When a class name is supplied for a slot or a superclass, a corresponding class definition will be found, looking from the name space or environment of the current package, assuming the call to `setClass` in question appears directly in the source for the package. That's where it should appear, to avoid ambiguity.

In particular, if the current package has a name space then the class must be found in the current package itself, in the imports defined by that name space, or in the base package.

When this rule does not identify a class uniquely (because it appears in more than one imported package) then the `packageSlot` of the character string name needs to be supplied with the name. This should be a rare occurrence.

Note

Certain slot names are not allowed in the current implementation, as they correspond to attributes which are treated specially. These are `class`, `comment`, `dim`, `dimnames`, `names`, `row.names` and `tsp`.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`Classes` for a general discussion of classes, `Methods` for an analogous discussion of methods, `makeClassRepresentation`

Examples

```
## A simple class with two slots
setClass("track",
         representation(x="numeric", y="numeric"))
## A class extending the previous, adding one more slot
setClass("trackCurve",
    representation(smooth = "numeric"),
    contains = "track")
## A class similar to "trackCurve", but with different structure
## allowing matrices for the "y" and "smooth" slots
setClass("trackMultiCurve",
         representation(x="numeric", y="matrix", smooth="matrix"),
         prototype = list(x=numeric(), y=matrix(0,0,0),
                          smooth= matrix(0,0,0)))
##
## Suppose we want trackMultiCurve to be like trackCurve when there's
## only one column.
## First, the wrong way.
try(setIs("trackMultiCurve", "trackCurve",
    test = function(obj) {ncol(slot(obj, "y")) == 1}))

## Why didn't that work?  You can only override the slots "x", "y",
## and "smooth" if you provide an explicit coerce function to correct
```

```
## any inconsistencies:

setIs("trackMultiCurve", "trackCurve",
  test = function(obj) {ncol(slot(obj, "y")) == 1},
  coerce = function(obj) {
     new("trackCurve",
         x = slot(obj, "x"),
         y = as.numeric(slot(obj,"y")),
         smooth = as.numeric(slot(obj, "smooth")))
  })

## A class that extends the built-in data type "numeric"

setClass("numWithId", representation(id = "character"),
         contains = "numeric")

new("numWithId", 1:3, id = "An Example")
```

`setClassUnion` *Classes Defined as the Union of Other Classes*

Description

A class may be defined as the *union* of other classes; that is, as a virtual class defined as a superclass of several other classes. Class unions are useful in method signatures or as slots in other classes, when we want to allow one of several classes to be supplied.

Usage

```
setClassUnion(name, members, where)
isClassUnion(Class)
```

Arguments

`name`	the name for the new union class.
`members`	the classes that should be members of this union.
`where`	where to save the new class definition; by default, the environment of the package in which the `setClassUnion` call appears, or the global environment if called outside of the source of a package.
`Class`	the name or definition of a class.

Details

The classes in `members` must be defined before creating the union. However, members can be added later on to an existing union, as shown in the example below. Class unions can be members of other class unions.

Class unions are the only way to create a class that is extended by a class whose definition is sealed (for example, the basic datatypes or other classes defined in the base or methods package in R are sealed). You cannot say `setIs("function", "other")` unless `"other"` is a class union.

In general, a `setIs` call of this form changes the definition of the first class mentioned (adding `"other"` to the list of superclasses contained in the definition of `"function"`).

Class unions get around this by not modifying the first class definition, relying instead on storing information in the subclasses slot of the class union. In order for this technique to work, the internal computations for expressions such as `extends(class1, class2)` work differently for class unions than for regular classes; specifically, they test whether any class is in common between the superclasses of `class1` and the subclasses of `class2`.

The different behavior for class unions is made possible because the class definition object for class unions has itself a special class, `"ClassUnionRepresentation"`, an extension of class `classRepresentation`.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

Examples

```
## a class for either numeric or logical data
setClassUnion("maybeNumber", c("numeric", "logical"))

## use the union as the data part of another class
setClass("withId", representation("maybeNumber", id = "character"))

w1 <- new("withId", 1:10, id = "test 1")
w2 <- new("withId", sqrt(w1)%%1 < .01, id = "Perfect squares")

## add class "complex" to the union "maybeNumber"
setIs("complex", "maybeNumber")

w3 <- new("withId", complex(real = 1:10, imaginary = sqrt(1:10)))

## a class union containing the existing class  union "OptionalFunction"
setClassUnion("maybeCode",
    c("expression", "language", "OptionalFunction"))

is(quote(sqrt(1:10)), "maybeCode")  ## TRUE
```

---

setGeneric *Define a New Generic Function*

Description

Create a new generic function of the given name, that is, a function that dispatches methods according to the classes of the arguments, from among the formal methods defined for this function.

Usage

```
setGeneric(name, def= , group=list(), valueClass=character(),
           where= , package= , signature= , useAsDefault= ,
           genericFunction= , simpleInheritanceOnly = )

setGroupGeneric(name, def= , group=list(), valueClass=character(),
                knownMembers=list(), package= , where= )
```

Arguments

`name`	The character string name of the generic function. The simplest (and recommended) call, `setGeneric(name)`, looks for a function with this name and creates a corresponding generic function, if the function found was not generic.
`def`	An optional function object, defining the generic. Don't supply this argument if you want an existing non-generic function to supply the arguments. Do supply it if there is no current function of this name, or if you want the generic function to have different arguments. In that case, the formal arguments and default values for the generic are taken from `def`. You can also supply this argument if you want the generic function to do something other than just dispatch methods. Note that `def` is *not* the default method; use argument `useAsDefault` if you want to specify the default separately.
`group`	Optionally, a character string giving the name of the group generic function to which this function belongs. See Methods for details of group generic functions in method selection.
`valueClass`	An optional character vector of one or more class names. The value returned by the generic function must have (or extend) this class, or one of the classes; otherwise, an error is generated.
`package`	The name of the package with which this function is associated. Usually determined automatically (as the package containing the non-generic version if there is one, or else the package where this generic is to be saved).
`where`	Where to store the resulting initial methods definition, and possibly the generic function; by default, stored into the top-level environment.
`signature`	Optionally, the vector of names, from among the formal arguments to the function, that can appear in the signature of methods for this function, in calls to `setMethod`. If ... is one of the formal arguments, it is treated specially. Starting with version 2.8.0 of R, ... may be signature of the generic function. Methods will then be selected if their signature matches all the ... arguments. See the documentation for topic dotsMethods for details. In the present version, it is not possible to mix ... and other arguments in the signature (this restriction may be lifted in later versions). By default, the signature is inferred from the implicit generic function corresponding to a non-generic function. If no implicit generic function has been defined, the default is all the formal arguments except ..., in the order they appear in the function definition. In the case that ... is the only formal argument, that is also the default signature. To use ... as the signature in a function that has any other arguments, you must supply the signature argument explicitly. See the "Implicit Generic" section below for more details.
`useAsDefault`	Override the usual choice of default argument (an existing non-generic function or no default if there is no such function). Argument `useAsDefault` can be supplied, either as a function to use for the default, or as a logical value. `FALSE`

says not to have a default method at all, so that an error occurs if there is not an explicit or inherited method for a call. TRUE says to use the existing function as default, unconditionally (hardly ever needed as an explicit argument). See the section on details.

simpleInheritanceOnly
: Supply this argument as TRUE to require that methods selected be inherited through simple inheritance only; that is, from superclasses specified in the contains= argument to setClass, or by simple inheritance to a class union or other virtual class. Generic functions should require simple inheritance if they need to be assured that they get the complete original object, not one that has been transformed. Examples of functions requiring simple inheritance are initialize, because by definition it must return an object from the same class as its argument, and show, because it claims to give a full description of the object provided as its argument.

genericFunction
: Don't use; for (possible) internal use only.

knownMembers
: (For setGroupGeneric only.) The names of functions that are known to be members of this group. This information is used to reset cached definitions of the member generics when information about the group generic is changed.

Value

The setGeneric function exists for its side effect: saving the generic function to allow methods to be specified later. It returns name.

Basic Use

The setGeneric function is called to initialize a generic function as preparation for defining some methods for that function.

The simplest and most common situation is that name is already an ordinary non-generic non-primitive function, and you now want to turn this function into a generic. In this case you will most often supply only name, for example:

```
setGeneric("colSums")
```

There must be an existing function of this name, on some attached package (in this case package "base"). A generic version of this function will be created in the current package (or in the global environment if the call to setGeneric() is from an ordinary source file or is entered on the command line). The existing function becomes the default method, and the package slot of the new generic function is set to the location of the original function ("base" in the example). It's an important feature that the same generic function definition is created each time, depending in the example only on the definition of print and where it is found. The signature of the generic function, defining which of the formal arguments can be used in specifying methods, is set by default to all the formal arguments except ....

Note that calling setGeneric() in this form is not strictly necessary before calling setMethod() for the same function. If the function specified in the call to setMethod is not generic, setMethod will execute the call to setGeneric itself. Declaring explicitly that you want the function to be generic can be considered better programming style; the only difference in the result, however, is that not doing so produces a message noting the creation of the generic function.

You cannot (and never need to) create an explicit generic version of the primitive functions in the base package. Those which can be treated as generic functions have methods selected and

dispatched from the internal C code, to satisfy concerns for efficiency, and the others cannot be made generic. See the section on Primitive Functions below.

The description above is the effect when the package that owns the non-generic function has not created an implicit generic version. Otherwise, it is this implicit generic function that is used. See the section on Implicit Generic Functions below. Either way, the essential result is that the *same* version of the generic function will be created each time.

The second common use of `setGeneric()` is to create a new generic function, unrelated to any existing function, and frequently having no default method. In this case, you need to supply a skeleton of the function definition, to define the arguments for the function. The body of a generic function is usually a standard form, `standardGeneric(name)` where `name` is the quoted name of the generic function. When calling `setGeneric` in this form, you would normally supply the `def` argument as a function of this form. See the second and third examples below.

The `useAsDefault` argument controls the default method for the new generic. If not told otherwise, `setGeneric` will try to find a non-generic version of the function to use as a default. So, if you do have a suitable default method, it is often simpler to first set this up as a non-generic function, and then use the one-argument call to `setGeneric` at the beginning of this section. See the first example in the Examples section below.

If you *don't* want the existing function to be taken as default, supply the argument `useAsDefault`. That argument can be the function you want to be the default method, or `FALSE` to force no default (i.e., to cause an error if there is no direct or inherited method selected for a call to the function).

Details

If you want to change the behavior of an existing function (typically, one in another package) when you create a generic version, you must supply arguments to `setGeneric` correspondingly. Whatever changes are made, the new generic function will be assigned with a package slot set to the *current* package, not the one in which the non-generic version of the function is found. This step is required because the version you are creating is no longer the same as that implied by the function in the other package. A message will be printed to indicate that this has taken place and noting one of the differences between the two functions.

The body of a generic function usually does nothing except for dispatching methods by a call to `standardGeneric`. Under some circumstances you might just want to do some additional computation in the generic function itself. As long as your function eventually calls `standardGeneric` that is permissible (though perhaps not a good idea, in that it may make the behavior of your function less easy to understand). If your explicit definition of the generic function does *not* call `standardGeneric` you are in trouble, because none of the methods for the function will ever be dispatched.

By default, the generic function can return any object. If `valueClass` is supplied, it should be a vector of class names; the value returned by a method is then required to satisfy `is(object, Class)` for one of the specified classes. An empty (i.e., zero length) vector of classes means anything is allowed. Note that more complicated requirements on the result can be specified explicitly, by defining a non-standard generic function.

The `setGroupGeneric` function behaves like `setGeneric` except that it constructs a group generic function, differing in two ways from an ordinary generic function. First, this function cannot be called directly, and the body of the function created will contain a stop call with this information. Second, the group generic function contains information about the known members of the group, used to keep the members up to date when the group definition changes, through changes in the search list or direct specification of methods, etc.

Implicit Generic Functions

Saying that a non-generic function "is converted to a generic" is more precisely state that the function is converted to the corresponding *implicit* generic function. If no special action has been taken, any function corresponds implicitly to a generic function with the same arguments, in which all arguments other than ... can be used. The signature of this generic function is the vector of formal arguments, in order, except for ....

The source code for a package can define an implicit generic function version of any function in that package (see implicitGeneric for the mechanism). You can not, generally, define an implicit generic function in someone else's package. The usual reason for defining an implicit generic is to prevent certain arguments from appearing in the signature, which you must do if you want the arguments to be used literally or if you want to enforce lazy evaluation for any reason. An implicit generic can also contain some methods that you want to be predefined; in fact, the implicit generic can be any generic version of the non-generic function. The implicit generic mechanism can also be used to prohibit a generic version (see `prohibitGeneric`).

Whether defined or inferred automatically, the implicit generic will be compared with the generic function that `setGeneric` creates, when the implicit generic is in another package. If the two functions are identical, then the `package` slot of the created generic will have the name of the package containing the implicit generic. Otherwise, the slot will be the name of the package in which the generic is assigned.

The purpose of this rule is to ensure that all methods defined for a particular combination of generic function and package names correspond to a single, consistent version of the generic function. Calling `setGeneric` with only `name` and possibly `package` as arguments guarantees getting the implicit generic version, if one exists.

Including any of the other arguments can force a new, local version of the generic function. If you don't want to create a new version, don't use the extra arguments.

Generic Functions and Primitive Functions

A number of the basic R functions are specially implemented as primitive functions, to be evaluated directly in the underlying C code rather than by evaluating an R language definition. Most have implicit generics (see `implicitGeneric`), and become generic as soon as methods (including group methods) are defined on them. Others cannot be made generic.

Even when methods are defined for such functions, the generic version is not visible on the search list, in order that the C version continues to be called. Method selection will be initiated in the C code. Note, however, that the result is to restrict methods for primitive functions to signatures in which at least one of the classes in the signature is a formal S4 class.

To see the generic version of a primitive function, use `getGeneric(name)`. The function `isGeneric` will tell you whether methods are defined for the function in the current session.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`Methods` and the links there for a general discussion, `dotsMethods` for methods that dispatch on "...", and `setMethod` for method definitions.

Examples

```
## create a new generic function, with a default method
props <- function(object) attributes(object)
setGeneric("props")

## A new generic function with no default method
setGeneric("increment",
  function(object, step, ...)
    standardGeneric("increment")
)

###   A non-standard generic function.  It insists that the methods
###   return a non-empty character vector (a stronger requirement than
###    valueClass = "character" in the call to setGeneric)

setGeneric("authorNames",
    function(text) {
      value <- standardGeneric("authorNames")
      if(!(is(value, "character") && any(nchar(value)>0)))
        stop("authorNames methods must return non-empty strings")
      value
      })



## An example of group generic methods, using the class
## "track"; see the documentation of setClass for its definition

## define a method for the Arith group

setMethod("Arith", c("track", "numeric"),
 function(e1, e2) {
  e1@y <- callGeneric(e1@y , e2)
  e1
})

setMethod("Arith", c("numeric", "track"),
 function(e1, e2) {
  e2@y <- callGeneric(e1, e2@y)
  e2
})

## now arithmetic operators  will dispatch methods:

t1 <- new("track", x=1:10, y=sort(stats::rnorm(10)))

t1 - 100
1/t1
```

`setMethod` *Create and Save a Method*

Description

Create and save a formal method for a given function and list of classes.

Usage

```
setMethod(f, signature=character(), definition,
          where = topenv(parent.frame()),
          valueClass = NULL, sealed = FALSE)

removeMethod(f, signature, where)
```

Arguments

f	A generic function or the character-string name of the function.
signature	A match of formal argument names for f with the character-string names of corresponding classes. See the details below; however, if the signature is not trivial, you should use method.skeleton to generate a valid call to setMethod.
definition	A function definition, which will become the method called when the arguments in a call to f match the classes in signature, directly or through inheritance.
where	the environment in which to store the definition of the method. For setMethod, it is recommended to omit this argument and to include the call in source code that is evaluated at the top level; that is, either in an R session by something equivalent to a call to source, or as part of the R source code for a package. For removeMethod, the default is the location of the (first) instance of the method for this signature.
valueClass	Obsolete and unused, but see the same argument for setGeneric.
sealed	If TRUE, the method so defined cannot be redefined by another call to setMethod (although it can be removed and then re-assigned).

Details

The call to setMethod stores the supplied method definition in the metadata table for this generic function in the environment, typically the global environment or the name space of a package. In the case of a package, the table object becomes part of the name space or environment of the package. When the package is loaded into a later session, the methods will be merged into the table of methods in the corresponding generic function object.

Generic functions are referenced by the combination of the function name and the package name; for example, the function "show" from the package "methods". Metadata for methods is identified by the two strings; in particular, the generic function object itself has slots containing its name and its package name. The package name of a generic is set according to the package from which it originally comes; in particular, and frequently, the package where a non-generic version of the function originated. For example, generic functions for all the functions in package base will have "base" as the package name, although none of them is an S4 generic on that package.

Multiple packages can have methods for the same generic function; that is, for the same combination of generic function name and package name. Even though the methods are stored in separate tables in separate environments, loading the corresponding packages adds the methods to the table in the generic function itself, for the duration of the session.

The class names in the signature can be any formal class, including basic classes such as "numeric", "character", and "matrix". Two additional special class names can appear:

"ANY", meaning that this argument can have any class at all; and "missing", meaning that this argument *must not* appear in the call in order to match this signature. Don't confuse these two: if an argument isn't mentioned in a signature, it corresponds implicitly to class "ANY", not to "missing". See the example below. Old-style ('S3') classes can also be used, if you need compatibility with these, but you should definitely declare these classes by calling setOldClass if you want S3-style inheritance to work.

Method definitions can have default expressions for arguments, but a current limitation is that the generic function must have *some* default expression for the same argument in order for the method's defaults to be used. If so, and if the corresponding argument is missing in the call to the generic function, the default expression in the method is used. If the method definition has no default for the argument, then the expression supplied in the definition of the generic function itself is used, but note that this expression will be evaluated using the enclosing environment of the method, not of the generic function. Note also that specifying class "missing" in the signature does not require any default expressions, and method selection does not evaluate default expressions. All actual (non-missing) arguments in the signature of the generic function will be evaluated when a method is selected—when the call to standardGeneric(f) occurs.

It is possible to have some differences between the formal arguments to a method supplied to setMethod and those of the generic. Roughly, if the generic has ... as one of its arguments, then the method may have extra formal arguments, which will be matched from the arguments matching ... in the call to f. (What actually happens is that a local function is created inside the method, with the modified formal arguments, and the method is re-defined to call that local function.)

Method dispatch tries to match the class of the actual arguments in a call to the available methods collected for f. If there is a method defined for the exact same classes as in this call, that method is used. Otherwise, all possible signatures are considered corresponding to the actual classes or to superclasses of the actual classes (including "ANY"). The method having the least distance from the actual classes is chosen; if more than one method has minimal distance, one is chosen (the lexicographically first in terms of superclasses) but a warning is issued. All inherited methods chosen are stored in another table, so that the inheritance calculations only need to be done once per session per sequence of actual classes. See Methods for more details.

The function removeMethod removes the specified method from the metadata table in the corresponding environment. It's not a function that is used much, since one normally wants to redefine a method rather than leave no definition.

Value

These functions exist for their side-effect, in setting or removing a method in the object defining methods for the specified generic.

The value returned by removeMethod is TRUE if a method was found to be removed.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

method.skeleton, which is the recommended way to generate a skeleton of the call to setMethod, with the correct formal arguments and other details.

Methods and the links there for a general discussion, dotsMethods for methods that dispatch on "...", and setGeneric for generic functions.

Examples

```
require(graphics)
## methods for plotting track objects (see the example for setClass)
##
## First, with only one object as argument:
setMethod("plot", signature(x="track", y="missing"),
  function(x,  y, ...) plot(slot(x, "x"), slot(x, "y"), ...)
)
## Second, plot the data from the track on the y-axis against anything
## as the x data.
setMethod("plot", signature(y = "track"),
 function(x, y, ...) plot(x, slot(y, "y"), ...)
)
## and similarly with the track on the x-axis (using the short form of
## specification for signatures)
setMethod("plot", "track",
 function(x, y, ...) plot(slot(x, "y"), y,  ...)
)
t1 <- new("track", x=1:20, y=(1:20)^2)
tc1 <- new("trackCurve", t1)
slot(tc1, "smooth") <- smooth.spline(slot(tc1, "x"), slot(tc1, "y"))$y #$
plot(t1)
plot(qnorm(ppoints(20)), t1)
## An example of inherited methods, and of conforming method arguments
## (note the dotCurve argument in the method, which will be pulled out
## of ... in the generic.
setMethod("plot", c("trackCurve", "missing"),
function(x, y, dotCurve = FALSE, ...) {
  plot(as(x, "track"))
  if(length(slot(x, "smooth") > 0))
    lines(slot(x, "x"), slot(x, "smooth"),
          lty = if(dotCurve) 2 else 1)
  }
)
## the plot of tc1 alone has an added curve; other uses of tc1
## are treated as if it were a "track" object.
plot(tc1, dotCurve = TRUE)
plot(qnorm(ppoints(20)), tc1)

## defining methods for a special function.
## Although "[" and "length" are not ordinary functions
## methods can be defined for them.
setMethod("[", "track",
  function(x, i, j, ..., drop) {
    x@x <- x@x[i]; x@y <- x@y[i]
    x
  })
plot(t1[1:15])

setMethod("length", "track", function(x)length(x@y))
length(t1)

## methods can be defined for missing arguments as well
setGeneric("summary") ## make the function into a generic
```

```
## A method for summary()
## The method definition can include the arguments, but
## if they're omitted, class "missing" is assumed.

setMethod("summary", "missing", function() "<No Object>")
```

`setOldClass` *Register Old-Style (S3) Classes and Inheritance*

Description

Register an old-style (a.k.a. 'S3') class as a formally defined class. The `Classes` argument is the character vector used as the `class` attribute; in particular, if there is more than one string, old-style class inheritance is mimicked. Registering via `setOldClass` allows S3 classes to appear in method signatures, as a slot in an S4 class, or as a superclass of an S4 class.

Usage

```
setOldClass(Classes, prototype, where, test = FALSE, S4Class)
```

Arguments

`Classes`	A character vector, giving the names for old-style classes, as they would appear on the right side of an assignment of the `class` attribute in S3 computations.
`prototype`	An optional object to use as the prototype. This should be provided as the default S3 object for the class. If omitted, the S4 class created to register the S3 class is `VIRTUAL`. See the details.
`where`	Where to store the class definitions, the global or top-level environment by default. (When either function is called in the source for a package, the class definitions will be included in the package's environment by default.)
`test`	flag, if `TRUE`, arrange to test inheritance explicitly for each object, needed if the S3 class can have a different set of class strings, with the same first string. This is a different mechanism in implementation and should be specified separately for each pair of classes that have an optional inheritance. See the details below.
`S4Class`	optionally, the class definition or the class name of an S4 class. The new class will have all the slots and other properties of this class, plus its S3 inheritance as defined by the `Classses` argument. Arguments `prototype` and `test` must not be supplied in this case. See the section on "S3 classes with known attributes" below.

Details

Each of the names will be defined as an S4 class, extending the remaining classes in `Classes`, and the class `oldClass`, which is the 'root' of all old-style classes. S3 classes have no formal definition, and therefore no formally defined slots. If a `prototype` argument is supplied in the call to `setOldClass()`, objects from the class can be generated, by a call to `new`; however, this usually not as relevant as generating objects from subclasses (see the section on extending S3 classes below). If a prototype is not provided, the class will be created as a virtual S4 class. The

main disadvantage is that the prototype object in an S4 class that uses this class as a slot will have a NULL object in that slot, which can sometimes lead to confusion.

Beginning with version 2.8.0 of R, support is provided for using a (registered) S3 class as a super class of a new S4 class. See the section on extending S3 classes below, and the examples.

See Methods for the details of method dispatch and inheritance.

Some S3 classes cannot be represented as an ordinary combination of S4 classes and superclasses, because objects from the S3 class can have a variable set of strings in the class. It is still possible to register such classes as S4 classes, but now the inheritance has to be verified for each object, and you must call setOldClass with argument test=TRUE once for each superclass.

For example, ordered factors *always* have the S3 class c("ordered", "factor"). This is proper behavior, and maps simply into two S4 classes, with "ordered" extending "factor".

But objects whose class attribute has "POSIXt" as the first string may have either (or neither) of "POSIXct" or "POSIXlt" as the second string. This behavior can be mapped into S4 classes but now to evaluate is(x, "POSIXlt"), for example, requires checking the S3 class attribute on each object. Supplying the test=TRUE argument to setOldClass causes an explicit test to be included in the class definitions. It's never wrong to have this test, but since it adds significant overhead to methods defined for the inherited classes, you should only supply this argument if it's known that object-specific tests are needed.

The list .OldClassesList contains the old-style classes that are defined by the methods package. Each element of the list is an old-style list, with multiple character strings if inheritance is included. Each element of the list was passed to setOldClass when creating the **methods** package; therefore, these classes can be used in setMethod calls, with the inheritance as implied by the list.

Extending S3 classes

A call to setOldClass creates formal classes corresponding to S3 classes, allows these to be used as slots in other classes or in a signature in setMethod, and mimics the S3 inheritance.

In documentation for the initial implementation of S4 classes in R, users were warned against defining S4 classes that contained S3 classes, even if those had been registered. The warning was based mainly on two points. 1: The S3 behavior of the objects would fail because the S3 class would not be visible, for example, when S3 methods are dispatched. 2: Because S3 classes have no formal definition, nothing can be asserted in general about the S3 part of an object from such a class. (The warning was repeated as recently as the first reference below.)

Nevertheless, defining S4 classes to contain an S3 class and extend its behavior is attractive in many applications. The alternative is to be stuck with S3 programming, without the flexibility and security of formal class and method definitions.

Beginning with version 2.8.0, R provides support for extending registered S3 classes; that is, for new classes defined by a call to setClass in which the contains= argument includes an S3 class. See the examples below. The support is aimed primarily at providing the S3 class information for all classes that extend class oldClass, in particular by ensuring that all objects from such classes contain the S3 class in a special slot.

There are three different ways to indicate an extension to an existing S3 class: setOldClass(), setClass() and setIs(). In most cases, calling setOldClass is the best approach, but the alternatives may be preferred in the special circumstances described below.

Suppose "A" is any class extending "oldClass". then

```
setOldClass(c("B", "A"))
```

creates a new class "B" whose S3 class concatenates "B" with S3Class("A"). The new class is a virtual class. If "A" was defined with known attribute/slots, then "B" has these slots also;

therefore, you must believe that the corresponding S3 objects from class "B" do indeed have the claimed attributes. Notice that you can supply an S4 definition for the new class to specify additional attributes (as described in the next section.) The first alternative call produces a non-virtual class.

```
setClass("B", contains = "A")
```

This creates a non-virtual class with the same slots and superclasses as class "A". However, class "B" is not included in the S3 class slot of the new class, unless you provide it explicitly in the prototype.

```
setClass("B"); setIs("B", "A", .....)
```

This creates a virtual class that extends "A", but does not contain the slots of "A". The additional arguments to setIs should provide a coerce and replacement method. In order for the new class to inherit S3 methods, the coerce method must ensure that the class "A" object produced has a suitable S3 class. The only likely reason to prefer this third approach is that class "B" is not consistent with known attributes in class "A".

S3 Classes with known attributes

A further specification of an S3 class can be made *if* the class is guaranteed to have some attributes of known class (where as with slots, "known" means that the attribute is an object of a specified class, or a subclass of that class).

In this case, the call to setOldClass() can supply an S4 class definition representing the known structure. Since S4 slots are implemented as attributes (largely for just this reason), the know attributes can be specified in the representation of the S4 class. The usual technique will be to create an S4 class with the desired structure, and then supply the class name or definition as the argument S4Class to setOldClass().

See the definition of class "ts" in the examples below. The call to setClass to create the S4 class can use the same class name, as here, so long as the class definition is not sealed. In the example, we define "ts" as a vector structure with a numeric slot for "tsp". The validity of this definition relies on an assertion that all the S3 code for this class is consistent with that definition; specifically, that all "ts" objects will behave as vector structures and will have a numeric "tsp" attribute. We believe this to be true of all the base code in R, but as always with S3 classes, no guarantee is possible.

The S4 class definition can have virtual superclasses (as in the "ts" case) if the S3 class is asserted to behave consistently with these (in the example, time-series objects are asserted to be consistent with the structure class).

For another example, look at the S4 class definition for "data.frame".

Be warned that failures of the S3 class to live up to its asserted behavior will usually go uncorrected, since S3 classes inherently have no definition, and the resulting invalid S4 objects can cause all sorts of grief. Many S3 classes are not candidates for known slots, either because the presence or class of the attributes are not guaranteed (e.g., dimnames in arrays, although these are not even S3 classes), or because the class uses named components of a list rather than attributes (e.g., "lm"). An attribute that is sometimes missing cannot be represented as a slot, not even by pretending that it is present with class "NULL", because attributes unlike slots can not have value NULL.

One irregularity that is usually tolerated, however, is to optionally add other attributes to those guaranteed to exist (for example, "terms" in "data.frame" objects returned by model.frame). As of version 2.8.0, validity checks by validObject ignore extra attributes; even if this check is tightened in the future, classes extending S3 classes would likely be exempted because extra attributes are so common.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version: see section 10.6 for method selection and section 13.4 for generic functions).

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

setClass, setMethod

Examples

```
require(stats)
setOldClass(c("mlm", "lm"))
setGeneric("dfResidual", function(model)standardGeneric("dfResidual"))
setMethod("dfResidual", "lm", function(model)model$df.residual)

## dfResidual will work on mlm objects as well as lm objects
myData <- data.frame(time = 1:10, y = (1:10)^.5)
myLm <- lm(cbind(y, y^3)  ~ time, myData)

## two examples extending S3 class "lm", class "xlm"  directly and "ylm" indirectly
setClass("xlm", representation(eps = "numeric"), contains = "lm")
setClass("ylm", representation(header = "character"), contains = "xlm")
ym1 = new("ylm", myLm, header = "Example", eps = 0.)
## for more examples, see ?S3Class.



## Examples of S3 classes with guaranteed attributes
## an S3 class "stamped" with a vector and  a "date" attribute
## Here is a generator function and an S3 print method.
## NOTE:  it's essential that the generator checks the attribute classes
stamped <- function(x, date = Sys.time()) {
    if(!inherits(date, "POSIXt"))
      stop("bad date argument")
    if(!is.vector(x))
      stop("x must be a vector")
    attr(x, "date") <- date
    class(x) <- "stamped"
    x
}

print.stamped <- function(x, ...) {
    print(as.vector(x))
    cat("Date: ",  format(attr(x,"date")), "\n")
}

## Now, an S4 class with the same structure:
setClass("stamped4", contains = "vector", representation(date = "POSIXt"))

## We can use the S4 class to register "stamped", with its attributes:
setOldClass("stamped", S4Class = "stamped4")
selectMethod("show", "stamped")
## and then remove "stamped4" to clean up
removeClass("stamped4")
```

```
someLetters <- stamped(sample(letters, 10),  ISOdatetime(2008, 10, 15, 12, 0, 0))

st <- new("stamped", someLetters)
st  # show() method prints the object's class, then calls the S3 print method.

stopifnot(identical(S3Part(st, TRUE), someLetters))

# creating the S4 object directly from its data part and slots
new("stamped", 1:10, date = ISOdatetime(1976, 5, 5, 15, 10, 0))

## Not run:
## The code in R that defines "ts" as an S4 class
setClass("ts", contains = "structure",
      representation(tsp = "numeric"),
      prototype(NA, tsp = rep(1,3))) # prototype to be a legal S3 time-series
## and now registers it as an S3 class
    setOldClass("ts", S4Class = "ts", where = envir)

## End(Not run)
```

`show` *Show an Object*

Description

Display the object, by printing, plotting or whatever suits its class. This function exists to be specialized by methods. The default method calls `showDefault`.

Formal methods for `show` will usually be invoked for automatic printing (see the details).

Usage

```
show(object)
```

Arguments

`object`	Any R object

Details

Objects from an S4 class (a class defined by a call to `setClass`) will be displayed automatically is if by a call to `show`. S4 objects that occur as attributes of S3 objects will also be displayed in this form; conversely, S3 objects encountered as slots in S4 objects will be printed using the S3 convention, as if by a call to `print`.

Methods defined for `show` will only be inherited by simple inheritance, since otherwise the method would not receive the complete, original object, with misleading results. See the `simpleInheritanceOnly` argument to `setGeneric` and the discussion in `setIs` for the general concept.

Value

`show` returns an invisible `NULL`.

See Also

`showMethods` prints all the methods for one or more functions; `showMlist` prints individual methods lists; `showClass` prints class definitions. Neither of the latter two normally needs to be called directly.

Examples

```
## following the example shown in the setMethod documentation ...
setClass("track",
         representation(x="numeric", y="numeric"))
setClass("trackCurve",
         representation("track", smooth = "numeric"))

t1 <- new("track", x=1:20, y=(1:20)^2)

tc1 <- new("trackCurve", t1)

setMethod("show", "track",
  function(object)print(rbind(x = object@x, y=object@y))
)
## The method will now be used for automatic printing of t1

t1

## Not run:
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    2    3    4    5    6    7    8    9    10    11    12
y    1    4    9   16   25   36   49   64   81   100   121   144
  [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
x    13    14    15    16    17    18    19    20
y   169   196   225   256   289   324   361   400
## End(Not run)
## and also for tc1, an object of a class that extends "track"
tc1

## Not run:
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    2    3    4    5    6    7    8    9    10    11    12
y    1    4    9   16   25   36   49   64   81   100   121   144
  [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
x    13    14    15    16    17    18    19    20
y   169   196   225   256   289   324   361   400
## End(Not run)
```

---

showMethods *Show all the methods for the specified function(s)*

Description

Show a summary of the methods for one or more generic functions, possibly restricted to those involving specified classes.

Usage

```
showMethods(f = character(), where = topenv(parent.frame()),
            classes = NULL, includeDefs = FALSE,
            inherited = !includeDefs,
            showEmpty, printTo = stdout(), fdef =)
```

Arguments

f	one or more function names. If omitted, all functions will be shown that match the other arguments.
where	Where to find the generic function, if not supplied as an argument. When f is missing, or length 0, this also determines which generic functions to examine. If where is supplied, only the generic functions returned by getGenerics(where) are eligible for printing. If where is also missing, all the cached generic functions are considered.
classes	If argument classes is supplied, it is a vector of class names that restricts the displayed results to those methods whose signatures include one or more of those classes.
includeDefs	If includeDefs is TRUE, include the definitions of the individual methods in the printout.
inherited	logical indicating if methods that have been found by inheritance, so far in the session, will be included and marked as inherited. Note that an inherited method will not usually appear until it has been used in this session. See selectMethod if you want to know what method would be dispatched for particular classes of arguments.
showEmpty	logical indicating whether methods with no defined methods matching the other criteria should be shown at all. By default, TRUE if and only if argument f is not missing.
printTo	The connection on which the information will be shown; by default, on standard output.
fdef	Optionally, the generic function definition to use; if missing, one is found, looking in where if that is specified. See also comment in 'Details'.

Details

The name and package of the generic are followed by the list of signatures for which methods are currently defined, according to the criteria determined by the various arguments. Note that the package refers to the source of the generic function. Individual methods for that generic can come from other packages as well.

When more than one generic function is involved, either as specified or because f was missing, the functions are found and showMethods is recalled for each, including the generic as the argument fdef. In complicated situations, this can avoid some anomalous results.

Value

If printTo is FALSE, the character vector that would have been printed is returned; otherwise the value is the connection or filename, via invisible.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`setMethod`, and `GenericFunctions` for other tools involving methods; `selectMethod` will show you the method dispatched for a particular function and signature of classes for the arguments.

Examples

```
require(graphics)

## Assuming the methods for plot
## are set up as in the example of help(setMethod),
## print (without definitions) the methods that involve class "track":
showMethods("plot", classes = "track")
## Not run:
Function "plot":
x = ANY, y = track
x = track, y = missing
x = track, y = ANY

require("Matrix")# a CRAN package with much S4 functionality
showMethods("%*%")# many!
    methods(class = "Matrix")# nothing
showMethods(class = "Matrix")# everything
## End(Not run)

not.there <- !any("package:stats4" == search())
if(not.there) library(stats4)
showMethods(classes = "mle")
if(not.there) detach("package:stats4")
```

`signature-class` *Class "signature" For Method Definitions*

Description

This class represents the mapping of some of the formal arguments of a function onto the names of some classes. It is used as one of two slots in the `MethodDefinition` class.

Objects from the Class

Objects can be created by calls of the form `new("signature", functionDef, ...)`. The `functionDef` argument, if it is supplied as a function object, defines the formal names. The other arguments define the classes.

Slots

.Data: Object of class "character" the classes.

names: Object of class "character" the corresponding argument names.

Extends

Class "character", from data part. Class "vector", by class "character".

Methods

initialize signature(object = "signature"): see the discussion of objects from the class, above.

See Also

class MethodDefinition for the use of this class.

---

slot *The Slots in an Object from a Formal Class*

Description

These functions return or set information about the individual slots in an object.

Usage

```
object@name
object@name <- value

slot(object, name)
slot(object, name, check = TRUE) <- value

slotNames(x)
getSlots(x)
```

Arguments

object	An object from a formally defined class.
name	The name of the slot. The operator takes a fixed name, which can be unquoted if it is syntactically a name in the language. A slot name can be any non-empty string, but if the name is not made up of letters, numbers, and ., it needs to be quoted (by backticks or single or double quotes). In the case of the slot function, name can be any expression that evaluates to a valid slot in the class definition. Generally, the only reason to use the functional form rather than the simpler operator is *because* the slot name has to be computed.
value	A new value for the named slot. The value must be valid for this slot in this object's class.

check — In the replacement version of slot, a flag. If TRUE, check the assigned value for validity as the value of this slot. User's coded should not set this to FALSE in normal use, since the resulting object can be invalid.

x — Either the name of a class or a class definition. If given an argument that is neither a character string nor a class definition, slotNames (only) uses class(x) instead.

Details

The definition of the class specifies all slots directly and indirectly defined for that class. Each slot has a name and an associated class. Extracting a slot returns an object from that class. Setting a slot first coerces the value to the specified slot and then stores it.

Unlike general attributes, slots are not partially matched, and asking for (or trying to set) a slot with an invalid name for that class generates an error.

The @ extraction operator and slot function themselves do no checking against the class definition, simply matching the name in the object itself. The replacement forms do check (except for slot in the case check=FALSE). So long as slots are set without cheating, the extracted slots will be valid.

Be aware that there are two ways to cheat, both to be avoided but with no guarantees. The obvious way is to assign a slot with check=FALSE. Also, slots in R are implemented as attributes, for the sake of some back compatibility. The current implementation does not prevent attributes being assigned, via attr<-, and such assignments are not checked for legitimate slot names.

Value

The "@" operator and the slot function extract or replace the formally defined slots for the object.

Functions slotNames and getSlots return respectively the names of the slots and the classes associated with the slots in the specified class definition. Except for its extended interpretation of x (above), slotNames(x) is just names(getSlots(x)).

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

@, Classes, Methods, getClass

Examples

```
setClass("track", representation(x="numeric", y="numeric"))
myTrack <- new("track", x = -4:4, y = exp(-4:4))
slot(myTrack, "x")
slot(myTrack, "y") <- log(slot(myTrack, "y"))
utils::str(myTrack)

getSlots("track") # or
getSlots(getClass("track"))
slotNames(class(myTrack)) # is the same as
```

```
slotNames(myTrack)
```

`StructureClasses` *Classes Corresponding to Basic Structures*

Description

The virtual class `structure` and classes that extend it are formal classes analogous to S language structures such as arrays and time-series.

Usage

```
## The following class names can appear in method signatures,
## as the class in as() and is() expressions, and, except for
## the classes commented as VIRTUAL, in calls to new()

"matrix"
"array"
"ts"

"structure" ## VIRTUAL
```

Objects from the Classes

Objects can be created by calls of the form `new(Class, ...)`, where `Class` is the quoted name of the specific class (e.g., `"matrix"`), and the other arguments, if any, are interpreted as arguments to the corresponding function, e.g., to function `matrix()`. There is no particular advantage over calling those functions directly, unless you are writing software designed to work for multiple classes, perhaps with the class name and the arguments passed in.

Objects created from the classes `"matrix"` and `"array"` are unusual, to put it mildly, and have been for some time. Although they may appear to be objects from these classes, they do not have the internal structure of either an S3 or S4 class object. In particular, they have no `"class"` attribute and are not recognized as objects with classes (that is, both `is.object` and `isS4` will return `FALSE` for such objects).

That the objects still behave as if they came from the corresponding class (most of the time, anyway) results from special code recognizing such objects being built into the base code of R. For most purposes, treating the classes in the usual way will work, fortunately. One consequence of the special treatment is that these two classes*may* be used as the data part of an S4 class; for example, you can get away with `contains = "matrix"` in a call to `setGeneric` to create an S4 class that is a subclass of `"matrix"`. There is no guarantee that everything will work perfectly, but a number of classes have been written in this form successfully.

The class `"ts"` is basically an S3 class that has been registered with S4, using the `setOldClass` mechanism. Versions of R through 2.7.0 treated this class as a pure S4 class, which was in principal a good idea, but in practice did not allow subclasses to be defined and had other intrinsic problems. (For example, setting the `"tsp"` parameters as a slot often fails because the built-in implementation does not allow the slot to be temporarily inconsistent with the length of the data. Also, the S4 class prevented the correct specification of the S3 inheritance for class `"mts"`.)

The current behavior (beginning with version 2.8.0 of R) registers `"ts"` as an S3 class, using an S4-style definition (see the documentation for `setOldClass` in the examples section for an abbreviated listing of how this is done. The S3 inheritance of `"mts"` in package `stats` is also registered. These classes, as well as `"matrix"` and `"array"` should be valid in most examples as superclasses for new S4 class definitions.

Extends

The specific classes all extend class `"structure"`, directly, and class `"vector"`, by class `"structure"`.

Methods

coerce Methods are defined to coerce arbitrary objects to these classes, by calling the corresponding basic function, for example, `as(x, "matrix")` calls `as.matrix(x)`.

Ops, Math Group methods (see, e.g., `S4groupGeneric`) are defined for combinations of structures and vectors (including special cases for array and matrix), implementing the concept of vector structures as in the reference.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (for the original vector structures).

Examples

```
showClass("structure")

## explore a bit :
showClass("ts")
(ts0 <- new("ts"))
str(ts0)

showMethods("Ops") # six methods from these classes, but maybe many more
```

`TraceClasses` *Classes Used Internally to Control Tracing*

Description

The classes described here are used by the R function `trace` to create versions of functions and methods including browser calls, etc., and also to `untrace` the same objects.

Usage

```
### Objects from the following classes are generated
### by calling trace() on an object from the corresponding
### class without the "WithTrace" in the name.

"functionWithTrace"
"MethodDefinitionWithTrace"
"MethodWithNextWithTrace"
"genericFunctionWithTrace"
"groupGenericFunctionWithTrace"

### the following is a virtual class extended by each of the
### classes above

"traceable"
```

Objects from the Class

Objects will be created from these classes by calls to `trace`. (There is an `initialize` method for class `"traceable"`, but you are unlikely to need it directly.)

Slots

`.Data`: The data part, which will be `"function"` for class `"functionWithTrace"`, and similarly for the other classes.

`original`: Object of the original class; e.g., `"function"` for class `"functionWithTrace"`.

Extends

Each of the classes extends the corresponding untraced class, from the data part; e.g., `"functionWithTrace"` extends `"function"`. Each of the specific classes extends `"traceable"`, directly, and class `"VIRTUAL"`, by class `"traceable"`.

Methods

The point of the specific classes is that objects generated from them, by function `trace()`, remain callable or dispatchable, in addition to their new trace information.

See Also

function `trace`

`validObject` *Test the Validity of an Object*

Description

The validity of `object` related to its class definition is tested. If the object is valid, `TRUE` is returned; otherwise, either a vector of strings describing validity failures is returned, or an error is generated (according to whether `test` is `TRUE`). Optionally, all slots in the object can also be validated.

The function `setValidity` sets the validity method of a class (but more normally, this method will be supplied as the `validity` argument to `setClass`). The method should be a function of one object that returns `TRUE` or a description of the non-validity.

Usage

```
validObject(object, test = FALSE, complete = FALSE)

setValidity(Class, method, where = topenv(parent.frame()) )

getValidity(ClassDef)
```

Arguments

`object`	any object, but not much will happen unless the object's class has a formal definition.
`test`	logical; if `TRUE` and validity fails, the function returns a vector of strings describing the problems. If `test` is `FALSE` (the default) validity failure generates an error.
`complete`	logical; if `TRUE`, validity methods will be applied recursively to any of the slots that have such methods.
`Class`	the name or class definition of the class whose validity method is to be set.
`ClassDef`	a class definition object, as from `getClassDef`.
`method`	a validity method; that is, either `NULL` or a function of one argument (`object`). Like `validObject`, the function should return `TRUE` if the object is valid, and one or more descriptive strings if any problems are found. Unlike `validObject`, it should never generate an error.
`where`	the modified class definition will be stored in this environment.

Note that validity methods do not have to check validity of superclasses: the logic of `validObject` ensures these tests are done once only. As a consequence, if one validity method wants to use another, it should extract and call the method from the other definition of the other class by calling `getValidity()`: it should *not* call `validObject`.

Details

Validity testing takes place 'bottom up': Optionally, if `complete=TRUE`, the validity of the object's slots, if any, is tested. Then, in all cases, for each of the classes that this class extends (the 'superclasses'), the explicit validity method of that class is called, if one exists. Finally, the validity method of `object`'s class is called, if there is one.

Testing generally stops at the first stage of finding an error, except that all the slots will be examined even if a slot has failed its validity test.

The standard validity test (with `complete=FALSE`) is applied when an object is created via `new` with any optional arguments (without the extra arguments the result is just the class prototype object).

An attempt is made to fix up the definition of a validity method if its argument is not `object`.

Value

`validObject` returns `TRUE` if the object is valid. Otherwise a vector of strings describing problems found, except that if `test` is `FALSE`, validity failure generates an error, with the corresponding strings in the error message.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`setClass`; class `classRepresentation`.

Examples

```
setClass("track",
          representation(x="numeric", y = "numeric"))
t1 <- new("track", x=1:10, y=sort(stats::rnorm(10)))
## A valid "track" object has the same number of x, y values
validTrackObject <- function(object) {
    if(length(object@x) == length(object@y)) TRUE
    else paste("Unequal x,y lengths: ", length(object@x), ", ",
               length(object@y), sep="")
}
## assign the function as the validity method for the class
setValidity("track", validTrackObject)
## t1 should be a valid "track" object
validObject(t1)
## Now we do something bad
t2 <- t1
t2@x <- 1:20
## This should generate an error
## Not run: try(validObject(t2))


setClass("trackCurve",
         representation("track", smooth = "numeric"))

## all superclass validity methods are used when validObject
## is called from initialize() with arguments, so this fails
## Not run: trynew("trackCurve", t2)


setClass("twoTrack", representation(tr1 = "track", tr2 ="track"))

## validity tests are not applied recursively by default,
## so this object is created (invalidly)
tT  <- new("twoTrack", tr2 = t2)

## A stricter test detects the problem
## Not run: try(validObject(tT, complete = TRUE))
```

Chapter 7

The `stats` package

`stats-package` *The R Stats Package*

Description

R statistical functions

Details

This package contains functions for statistical calculations and random number generation.

For a complete list of functions, use `library(help="stats")`.

Author(s)

R Development Core Team and contributors worldwide

Maintainer: R Core Team ⟨R-core@r-project.org⟩

`.checkMFClasses` *Functions to Check the Type of Variables passed to Model Frames*

Description

`.checkMFClasses` checks if the variables used in a predict method agree in type with those used for fitting.

`.MFclass` categorizes variables for this purpose.

Usage

```
.checkMFClasses(cl, m, ordNotOK = FALSE)
.MFclass(x)
.getXlevels(Terms, m)
```

Arguments

cl	a character vector of class descriptions to match.
m	a model frame.
x	any R object.
ordNotOK	logical: are ordered factors different?
Terms	a terms object.

Details

For applications involving model.matrix such as linear models we do not need to differentiate between ordered factors and factors as although these affect the coding, the coding used in the fit is already recorded and imposed during prediction. However, other applications may treat ordered factors differently: rpart does, for example.

Value

.MFclass returns a character string, one of "logical", "ordered", "factor", "numeric", "nmatrix.*" (a numeric matrix with a number of columns appended) or "other".

.getXlevels returns a named character vector, or NULL.

---

acf — *Auto- and Cross- Covariance and -Correlation Function Estimation*

Description

The function acf computes (and by default plots) estimates of the autocovariance or autocorrelation function. Function pacf is the function used for the partial autocorrelations. Function ccf computes the cross-correlation or cross-covariance of two univariate series.

Usage

```
acf(x, lag.max = NULL,
    type = c("correlation", "covariance", "partial"),
    plot = TRUE, na.action = na.fail, demean = TRUE, ...)

pacf(x, lag.max, plot, na.action, ...)

## Default S3 method:
pacf(x, lag.max = NULL, plot = TRUE, na.action = na.fail,
    ...)

ccf(x, y, lag.max = NULL, type = c("correlation", "covariance"),
    plot = TRUE, na.action = na.fail, ...)

## S3 method for class 'acf':
x[i, j]
```

Arguments

`x, y`	a univariate or multivariate (not `ccf`) numeric time series object or a numeric vector or matrix, or an `"acf"` object.
`lag.max`	maximum lag at which to calculate the acf. Default is $10\log_{10}(N/m)$ where N is the number of observations and m the number of series. Will be automatically limited to one less than the number of observations in the series.
`type`	character string giving the type of acf to be computed. Allowed values are `"correlation"` (the default), `"covariance"` or `"partial"`.
`plot`	logical. If `TRUE` (the default) the acf is plotted.
`na.action`	function to be called to handle missing values. `na.pass` can be used.
`demean`	logical. Should the covariances be about the sample means?
`...`	further arguments to be passed to `plot.acf`.
`i`	a set of lags (time differences) to retain.
`j`	a set of series (names or numbers) to retain.

Details

For `type = "correlation"` and `"covariance"`, the estimates are based on the sample covariance. (The lag 0 autocorrelation is fixed at 1 by convention.)

By default, no missing values are allowed. If the `na.action` function passes through missing values (as `na.pass` does), the covariances are computed from the complete cases. This means that the estimate computed may well not be a valid autocorrelation sequence, and may contain missing values. Missing values are not allowed when computing the PACF of a multivariate time series.

The partial correlation coefficient is estimated by fitting autoregressive models of successively higher orders up to `lag.max`.

The generic function `plot` has a method for objects of class `"acf"`.

The lag is returned and plotted in units of time, and not numbers of observations.

There are `print` and subsetting methods for objects of class `"acf"`.

Value

An object of class `"acf"`, which is a list with the following elements:

`lag`	A three dimensional array containing the lags at which the acf is estimated.
`acf`	An array with the same dimensions as `lag` containing the estimated acf.
`type`	The type of correlation (same as the `type` argument).
`n.used`	The number of observations in the time series.
`series`	The name of the series `x`.
`snames`	The series names for a multivariate time series.

The lag `k` value returned by `ccf(x,y)` estimates the correlation between `x[t+k]` and `y[t]`.

The result is returned invisibly if `plot` is `TRUE`.

Author(s)

Original: Paul Gilbert, Martyn Plummer. Extensive modifications and univariate case of `pacf` by B. D. Ripley.

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth Edition. Springer-Verlag.

(This contains the exact definitions used.)

See Also

`plot.acf`, `ARMAacf` for the exact autocorrelations of a given ARMA process.

Examples

```
require(graphics)

## Examples from Venables & Ripley
acf(lh)
acf(lh, type = "covariance")
pacf(lh)

acf(ldeaths)
acf(ldeaths, ci.type = "ma")
acf(ts.union(mdeaths, fdeaths))
ccf(mdeaths, fdeaths, ylab = "cross-correlation")
# (just the cross-correlations)

presidents # contains missing values
acf(presidents, na.action = na.pass)
pacf(presidents, na.action = na.pass)
```

---

acf2AR *Compute an AR Process Exactly Fitting an ACF*

Description

Compute an AR process exactly fitting an autocorrelation function.

Usage

```
acf2AR(acf)
```

Arguments

`acf`	An autocorrelation or autocovariance sequence.

Value

A matrix, with one row for the computed AR(p) coefficients for `1 <= p <= length(acf)`.

See Also

`ARMAacf`, `ar.yw` which does this from an empirical ACF.

Examples

```
(Acf <- ARMAacf(c(0.6, 0.3, -0.2)))
acf2AR(Acf)
```

`add1` *Add or Drop All Possible Single Terms to a Model*

Description

Compute all the single terms in the `scope` argument that can be added to or dropped from the model, fit those models and compute a table of the changes in fit.

Usage

```
add1(object, scope, ...)

## Default S3 method:
add1(object, scope, scale = 0, test = c("none", "Chisq"),
     k = 2, trace = FALSE, ...)

## S3 method for class 'lm':
add1(object, scope, scale = 0, test = c("none", "Chisq", "F"),
     x = NULL, k = 2, ...)

## S3 method for class 'glm':
add1(object, scope, scale = 0, test = c("none", "Chisq", "F"),
     x = NULL, k = 2, ...)

drop1(object, scope, ...)

## Default S3 method:
drop1(object, scope, scale = 0, test = c("none", "Chisq"),
      k = 2, trace = FALSE, ...)

## S3 method for class 'lm':
drop1(object, scope, scale = 0, all.cols = TRUE,
      test = c("none", "Chisq", "F"), k = 2, ...)

## S3 method for class 'glm':
drop1(object, scope, scale = 0, test = c("none", "Chisq", "F"),
      k = 2, ...)
```

Arguments

`object`	a fitted model object.
`scope`	a formula giving the terms to be considered for adding or dropping.
`scale`	an estimate of the residual mean square to be used in computing C_p. Ignored if `0` or `NULL`.

test	should the results include a test statistic relative to the original model? The F test is only appropriate for lm and aov models or perhaps for glm fits with estimated dispersion. The χ^2 test can be an exact test (lm models with known scale) or a likelihood-ratio test or a test of the reduction in scaled deviance depending on the method.
k	the penalty constant in AIC / C_p.
trace	if TRUE, print out progress reports.
x	a model matrix containing columns for the fitted model and all terms in the upper scope. Useful if add1 is to be called repeatedly. **Warning:** no checks are done on its validity.
all.cols	(Provided for compatibility with S.) Logical to specify whether all columns of the design matrix should be used. If FALSE then non-estimable columns are dropped, but the result is not usually statistically meaningful.
...	further arguments passed to or from other methods.

Details

For drop1 methods, a missing scope is taken to be all terms in the model. The hierarchy is respected when considering terms to be added or dropped: all main effects contained in a second-order interaction must remain, and so on.

In a scope formula . means 'what is already there'.

The methods for lm and glm are more efficient in that they do not recompute the model matrix and call the fit methods directly.

The default output table gives AIC, defined as minus twice log likelihood plus $2p$ where p is the rank of the model (the number of effective parameters). This is only defined up to an additive constant (like log-likelihoods). For linear Gaussian models with fixed scale, the constant is chosen to give Mallows' C_p, $RSS/scale + 2p - n$. Where C_p is used, the column is labelled as Cp rather than AIC.

The F tests for the "glm" methods are based on analysis of deviance tests, so if the dispersion is estimated it is based on the residual deviance, unlike the F tests of anova.glm.

Value

An object of class "anova" summarizing the differences in fit between the models.

Warning

The model fitting must apply the models to the same dataset. Most methods will attempt to use a subset of the data with no missing values for any of the variables if na.action=na.omit, but this may give biased results. Only use these functions with data containing missing values with great care.

Note

These are not fully equivalent to the functions in S. There is no keep argument, and the methods used are not quite so computationally efficient.

Their authors' definitions of Mallows' C_p and Akaike's AIC are used, not those of the authors of the models chapter of S.

Author(s)

The design was inspired by the S functions of the same names described in Chambers (1992).

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`step`, `aov`, `lm`, `extractAIC`, `anova`

Examples

```
require(graphics); require(utils)
example(step)#-> swiss
add1(lm1, ~ I(Education^2) + .^2)
drop1(lm1, test="F")  # So called 'type II' anova

example(glm)
drop1(glm.D93, test="Chisq")
drop1(glm.D93, test="F")
```

---

addmargins *Puts Arbitrary Margins on Multidimensional Tables or Arrays*

Description

For a given table one can specify which of the classifying factors to expand by one or more levels to hold margins to be calculated. One may for example form sums and means over the first dimension and medians over the second. The resulting table will then have two extra levels for the first dimension and one extra level for the second. The default is to sum over all margins in the table. Other possibilities may give results that depend on the order in which the margins are computed. This is flagged in the printed output from the function.

Usage

```
addmargins(A, margin = 1:length(dim(A)), FUN = sum, quiet = FALSE)
```

Arguments

`A`	table or array. The function uses the presence of the `"dim"` and `"dimnames"` attributes of `A`.
`margin`	vector of dimensions over which to form margins. Margins are formed in the order in which dimensions are specified in `margin`.
`FUN`	list of the same length as `margin`, each element of the list being either a function or a list of functions. Names of the list elements will appear as levels in dimnames of the result. Unnamed list elements will have names constructed: the name of a function or a constructed name based on the position in the table.
`quiet`	logical which suppresses the message telling the order in which the margins were computed.

Details

If the functions used to form margins are not commutative the result depends on the order in which margins are computed. Annotation of margins is done via naming the FUN list.

Value

A table or array with the same number of dimensions as A, but with extra levels of the dimensions mentioned in margin. The number of levels added to each dimension is the length of the entries in FUN. A message with the order of computation of margins is printed.

Author(s)

Bendix Carstensen, Steno Diabetes Center & Department of Biostatistics, University of Copenhagen, http://www.biostat.ku.dk/~bxc, autumn 2003. Margin naming enhanced by Duncan Murdoch.

See Also

table, ftable, margin.table.

Examples

```
Aye <- sample(c("Yes", "Si", "Oui"), 177, replace = TRUE)
Bee <- sample(c("Hum", "Buzz"), 177, replace = TRUE)
Sea <- sample(c("White", "Black", "Red", "Dead"), 177, replace = TRUE)
(A <- table(Aye, Bee, Sea))
addmargins(A)

ftable(A)
ftable(addmargins(A))

# Non-commutative functions - note differences between resulting tables:
ftable(addmargins(A, c(1,3),
       FUN = list(Sum = sum, list(Min = min, Max = max))))
ftable(addmargins(A, c(3,1),
       FUN = list(list(Min = min, Max = max), Sum = sum)))

# Weird function needed to return the N when computing percentages
sqsm <- function(x) sum(x)^2/100
B <- table(Sea, Bee)
round(sweep(addmargins(B, 1, list(list(All = sum, N = sqsm))), 2,
            apply(B, 2, sum)/100, "/"), 1)
round(sweep(addmargins(B, 2, list(list(All = sum, N = sqsm))), 1,
            apply(B, 1, sum)/100, "/"), 1)

# A total over Bee requires formation of the Bee-margin first:
mB <-  addmargins(B, 2, FUN = list(list(Total = sum)))
round(ftable(sweep(addmargins(mB, 1, list(list(All = sum, N = sqsm))), 2,
                   apply(mB,2,sum)/100, "/")), 1)

## Zero.Printing table+margins:
set.seed(1)
x <- sample( 1:7, 20, replace=TRUE)
y <- sample( 1:7, 20, replace=TRUE)
tx <- addmargins( table(x, y) )
print(tx, zero.print = ".")
```

aggregate *Compute Summary Statistics of Data Subsets*

Description

Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

Usage

```
aggregate(x, ...)

## Default S3 method:
aggregate(x, ...)

## S3 method for class 'data.frame':
aggregate(x, by, FUN, ...)

## S3 method for class 'ts':
aggregate(x, nfrequency = 1, FUN = sum, ndeltat = 1,
          ts.eps = getOption("ts.eps"), ...)
```

Arguments

x	an R object.
by	a list of grouping elements, each as long as the variables in x.
FUN	a scalar function to compute the summary statistics which can be applied to all data subsets.
nfrequency	new number of observations per unit of time; must be a divisor of the frequency of x.
ndeltat	new fraction of the sampling period between successive observations; must be a divisor of the sampling interval of x.
ts.eps	tolerance used to decide if nfrequency is a sub-multiple of the original frequency.
...	further arguments passed to or used by methods.

Details

aggregate is a generic function with methods for data frames and time series.

The default method aggregate.default uses the time series method if x is a time series, and otherwise coerces x to a data frame and calls the data frame method.

aggregate.data.frame is the data frame method. If x is not a data frame, it is coerced to one, which must have a non-zero number of rows. Then, each of the variables (columns) in x is split into subsets of cases (rows) of identical combinations of the components of by, and FUN is applied to each such subset with further arguments in ... passed to it. (I.e., tapply(VAR, by, FUN, ..., simplify = FALSE) is done for each variable VAR in x, conveniently wrapped into one call to lapply().) Empty subsets are removed, and the result is reformatted into a data frame containing the variables in by and x. The ones arising from by contain the unique combinations of grouping values used for determining the subsets, and the ones arising from x the corresponding

summary statistics for the subset of the respective variables in `x`. Rows with missing values in any of the `by` variables will be omitted from the result.

`aggregate.ts` is the time series method. If `x` is not a time series, it is coerced to one. Then, the variables in `x` are split into appropriate blocks of length `frequency(x) / nfrequency`, and `FUN` is applied to each such block, with further (named) arguments in `...` passed to it. The result returned is a time series with frequency `nfrequency` holding the aggregated values. Note that this make most sense for a quarterly or yearly result when the original series covers a whole number of quarters or years: in particular aggregating a monthly series to quarters starting in February does not give a conventional quarterly series.

Value

For the time series method, a time series of class `"ts"` or class `c("mts", "ts")`.

For the data frame method, a data frame with columns corresponding to the grouping variables in `by` followed by aggregated columns from `x`. If the `by` has names, the non-empty times are used to label the columns in the results, with unnamed grouping variables being named `Group.`*i* for `by[[`*i*`]]`.

Note: prior to R 2.6.0 the grouping variables were reported as factors with levels in alphabetical order in the current locale. Now the variable in the result is found by subsetting the original variable.

Author(s)

Kurt Hornik

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`apply`, `lapply`, `tapply`.

Examples

```
## Compute the averages for the variables in 'state.x77', grouped
## according to the region (Northeast, South, North Central, West) that
## each state belongs to.
aggregate(state.x77, list(Region = state.region), mean)

## Compute the averages according to region and the occurrence of more
## than 130 days of frost.
aggregate(state.x77,
          list(Region = state.region,
               Cold = state.x77[,"Frost"] > 130),
          mean)
## (Note that no state in 'South' is THAT cold.)

## example with character variables and NAs
testDF <- data.frame(v1 = c(1,3,5,7,8,3,5,NA,4,5,7,9),
                     v2 = c(11,33,55,77,88,33,55,NA,44,55,77,99) )
by1 <- c("red","blue",1,2,NA,"big",1,2,"red",1,NA,12)
by2 <- c("wet","dry",99,95,NA,"damp",95,99,"red",99,NA,NA)
aggregate(x = testDF, by = list(by1, by2), FUN = "mean")
```

```
# and if you want to treat NAs as a group
fby1 <- factor(by1, exclude = "")
fby2 <- factor(by2, exclude = "")
aggregate(x = testDF, by = list(fby1, fby2), FUN = "mean")

## Compute the average annual approval ratings for American presidents.
aggregate(presidents, nfrequency = 1, FUN = mean)
## Give the summer less weight.
aggregate(presidents, nfrequency = 1,
          FUN = weighted.mean, w = c(1, 1, 0.5, 1))
```

`AIC` *Akaike's An Information Criterion*

Description

Generic function calculating the Akaike information criterion for one or several fitted model objects for which a log-likelihood value can be obtained, according to the formula -2log-likelihood $+ kn_{par}$, where n_{par} represents the number of parameters in the fitted model, and $k = 2$ for the usual AIC, or $k = \log(n)$ (n the number of observations) for the so-called BIC or SBC (Schwarz's Bayesian criterion).

Usage

```
AIC(object, ..., k = 2)
```

Arguments

`object`	a fitted model object, for which there exists a `logLik` method to extract the corresponding log-likelihood, or an object inheriting from class `logLik`.
`...`	optionally more fitted model objects.
`k`	numeric, the *penalty* per parameter to be used; the default `k = 2` is the classical AIC.

Details

The default method for `AIC`, `AIC.default()` entirely relies on the existence of a `logLik` method computing the log-likelihood for the given class.

When comparing fitted objects, the smaller the AIC, the better the fit.

The log-likelihood and hence the AIC is only defined up to an additive constant. Different constants have conventionally be used for different purposes and so `extractAIC` and `AIC` may give different values (and do for models of class `"lm"`: see the help for `extractAIC`).

Value

If just one object is provided, returns a numeric value with the corresponding AIC (or BIC, or ..., depending on `k`); if multiple objects are provided, returns a `data.frame` with rows corresponding to the objects and columns representing the number of parameters in the model (`df`) and the AIC.

Author(s)

Jose Pinheiro and Douglas Bates

References

Sakamoto, Y., Ishiguro, M., and Kitagawa G. (1986). *Akaike Information Criterion Statistics*. D. Reidel Publishing Company.

See Also

`extractAIC`, `logLik`.

Examples

```
lm1 <- lm(Fertility ~ . , data = swiss)
AIC(lm1)
stopifnot(all.equal(AIC(lm1),
                    AIC(logLik(lm1))))
## a version of BIC or Schwarz' BC :
AIC(lm1, k = log(nrow(swiss)))
```

---

alias *Find Aliases (Dependencies) in a Model*

Description

Find aliases (linearly dependent terms) in a linear model specified by a formula.

Usage

```
alias(object, ...)

## S3 method for class 'formula':
alias(object, data, ...)

## S3 method for class 'lm':
alias(object, complete = TRUE, partial = FALSE,
      partial.pattern = FALSE, ...)
```

Arguments

`object`	A fitted model object, for example from `lm` or `aov`, or a formula for `alias.formula`.
`data`	Optionally, a data frame to search for the objects in the formula.
`complete`	Should information on complete aliasing be included?
`partial`	Should information on partial aliasing be included?
`partial.pattern`	Should partial aliasing be presented in a schematic way? If this is done, the results are presented in a more compact way, usually giving the deciles of the coefficients.
`...`	further arguments passed to or from other methods.

Details

Although the main method is for class "lm", alias is most useful for experimental designs and so is used with fits from aov. Complete aliasing refers to effects in linear models that cannot be estimated independently of the terms which occur earlier in the model and so have their coefficients omitted from the fit. Partial aliasing refers to effects that can be estimated less precisely because of correlations induced by the design.

Value

A list (of class "listof") containing components

Model	Description of the model; usually the formula.
Complete	A matrix with columns corresponding to effects that are linearly dependent on the rows.
Partial	The correlations of the estimable effects, with a zero diagonal. An object of class "mtable" which has its own print method.

Note

The aliasing pattern may depend on the contrasts in use: Helmert contrasts are probably most useful.

The defaults are different from those in S.

Author(s)

The design was inspired by the S function of the same name described in Chambers *et al.* (1992).

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Examples

```
## From Venables and Ripley (2002) p.165.
utils::data(npk, package="MASS")

op <- options(contrasts=c("contr.helmert", "contr.poly"))
npk.aov <- aov(yield ~ block + N*P*K, npk)
alias(npk.aov)
options(op)# reset
```

---

anova *Anova Tables*

Description

Compute analysis of variance (or deviance) tables for one or more fitted model objects.

Usage

```
anova(object, ...)
```

Arguments

object	an object containing the results returned by a model fitting function (e.g., lm or glm).
...	additional objects of the same type.

Value

This (generic) function returns an object of class anova. These objects represent analysis-of-variance and analysis-of-deviance tables. When given a single argument it produces a table which tests whether the model terms are significant.

When given a sequence of objects, anova tests the models against one another in the order specified.

The print method for anova objects prints tables in a 'pretty' form.

Warning

The comparison between two or more models will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R's default of na.action = na.omit is used.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*, Wadsworth & Brooks/Cole.

See Also

coefficients, effects, fitted.values, residuals, summary, drop1, add1.

---

anova.glm *Analysis of Deviance for Generalized Linear Model Fits*

Description

Compute an analysis of deviance table for one or more generalized linear model fits.

Usage

```
## S3 method for class 'glm':
anova(object, ..., dispersion = NULL, test = NULL)
```

Arguments

object, ...	objects of class glm, typically the result of a call to glm, or a list of objects for the "glmlist" method.
dispersion	the dispersion parameter for the fitting family. By default it is obtained from the object(s).
test	a character string, (partially) matching one of "Chisq", "F" or "Cp". See stat.anova.

Details

Specifying a single object gives a sequential analysis of deviance table for that fit. That is, the reductions in the residual deviance as each term of the formula is added in turn are given in as the rows of a table, plus the residual deviances themselves.

If more than one object is specified, the table has a row for the residual degrees of freedom and deviance for each model. For all but the first model, the change in degrees of freedom and deviance is also given. (This only makes statistical sense if the models are nested.) It is conventional to list the models from smallest to largest, but this is up to the user.

The table will optionally contain test statistics (and P values) comparing the reduction in deviance for the row to the residuals. For models with known dispersion (e.g., binomial and Poisson fits) the chi-squared test is most appropriate, and for those with dispersion estimated by moments (e.g., `gaussian`, `quasibinomial` and `quasipoisson` fits) the F test is most appropriate. Mallows' C_p statistic is the residual deviance plus twice the estimate of σ^2 times the residual degrees of freedom, which is closely related to AIC (and a multiple of it if the dispersion is known).

The dispersion estimate will be taken from the largest model, using the value returned by `summary.glm`. As this will in most cases use a Chisquared-based estimate, the F tests are not based on the residual deviance in the analysis of deviance table shown.

Value

An object of class `"anova"` inheriting from class `"data.frame"`.

Warning

The comparison between two or more models by `anova` or `anova.glmlist` will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R's default of `na.action = na.omit` is used, and `anova.glmlist` will detect this with an error.

References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`glm`, `anova`.

`drop1` for so-called 'type II' anova where each term is dropped one at a time respecting their hierarchy.

Examples

```
## --- Continuing the Example from  '?glm':

anova(glm.D93)
anova(glm.D93, test = "Cp")
anova(glm.D93, test = "Chisq")
```

`anova.lm` *ANOVA for Linear Model Fits*

Description

Compute an analysis of variance table for one or more linear model fits.

Usage

```
## S3 method for class 'lm':
anova(object, ...)

anova.lmlist(object, ..., scale = 0, test = "F")
```

Arguments

`object, ...`	objects of class `lm`, usually, a result of a call to `lm`.
`test`	a character string specifying the test statistic to be used. Can be one of `"F"`, `"Chisq"` or `"Cp"`, with partial matching allowed, or `NULL` for no test.
`scale`	numeric. An estimate of the noise variance σ^2. If zero this will be estimated from the largest model considered.

Details

Specifying a single object gives a sequential analysis of variance table for that fit. That is, the reductions in the residual sum of squares as each term of the formula is added in turn are given in as the rows of a table, plus the residual sum of squares.

The table will contain F statistics (and P values) comparing the mean square for the row to the residual mean square.

If more than one object is specified, the table has a row for the residual degrees of freedom and sum of squares for each model. For all but the first model, the change in degrees of freedom and sum of squares is also given. (This only make statistical sense if the models are nested.) It is conventional to list the models from smallest to largest, but this is up to the user.

Optionally the table can include test statistics. Normally the F statistic is most appropriate, which compares the mean square for a row to the residual sum of squares for the largest model considered. If `scale` is specified chi-squared tests can be used. Mallows' C_p statistic is the residual sum of squares plus twice the estimate of σ^2 times the residual degrees of freedom.

Value

An object of class `"anova"` inheriting from class `"data.frame"`.

Warning

The comparison between two or more models will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R's default of `na.action = na.omit` is used, and `anova.lmlist` will detect this with an error.

Note

Versions of R prior to 1.2.0 based F tests on pairwise comparisons, and this behaviour can still be obtained by a direct call to `anovalist.lm`.

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

The model fitting function `lm`, `anova`.

`drop1` for so-called 'type II' anova where each term is dropped one at a time respecting their hierarchy.

Examples

```
## sequential table
fit <- lm(sr ~ ., data = LifeCycleSavings)
anova(fit)

## same effect via separate models
fit0 <- lm(sr ~ 1, data = LifeCycleSavings)
fit1 <- update(fit0, . ~ . + pop15)
fit2 <- update(fit1, . ~ . + pop75)
fit3 <- update(fit2, . ~ . + dpi)
fit4 <- update(fit3, . ~ . + ddpi)
anova(fit0, fit1, fit2, fit3, fit4, test="F")

anova(fit4, fit2, fit0, test="F") # unconventional order
```

`anova.mlm` *Comparisons between Multivariate Linear Models*

Description

Compute a (generalized) analysis of variance table for one or more multivariate linear models.

Usage

```
## S3 method for class 'mlm':
anova(object, ...,
      test =
         c("Pillai", "Wilks", "Hotelling-Lawley", "Roy", "Spherical"),
      Sigma = diag(nrow = p), T = Thin.row(proj(M) - proj(X)),
       M = diag(nrow = p), X = ~0,
      idata = data.frame(index = seq_len(p)), tol = 1e-7)
```

Arguments

object	an object of class "mlm".
...	further objects of class "mlm".
test	choice of test statistic (see below).
Sigma	(only relevant if test == "Spherical"). Covariance matrix assumed proportional to Sigma.
T	transformation matrix. By default computed from M and X.
M	formula or matrix describing the outer projection (see below).
X	formula or matrix describing the inner projection (see below).
idata	data frame describing intra-block design.
tol	tolerance to be used in deciding if the residuals are rank-deficient: see qr.

Details

The anova.mlm method uses either a multivariate test statistic for the summary table, or a test based on sphericity assumptions (i.e. that the covariance is proportional to a given matrix).

For the multivariate test, Wilks' statistic is most popular in the literature, but the default Pillai–Bartlett statistic is recommended by Hand and Taylor (1987). See summary.manova for further details.

For the "Spherical" test, proportionality is usually with the identity matrix but a different matrix can be specified using Sigma). Corrections for asphericity known as the Greenhouse–Geisser, respectively Huynh–Feldt, epsilons are given and adjusted F tests are performed.

It is common to transform the observations prior to testing. This typically involves transformation to intra-block differences, but more complicated within-block designs can be encountered, making more elaborate transformations necessary. A transformation matrix T can be given directly or specified as the difference between two projections onto the spaces spanned by M and X, which in turn can be given as matrices or as model formulas with respect to idata (the tests will be invariant to parametrization of the quotient space M/X).

As with anova.lm, all test statistics use the SSD matrix from the largest model considered as the (generalized) denominator.

Contrary to other anova methods, the intercept is not excluded from the display in the single-model case. When contrast transformations are involved, it often makes good sense to test for a zero intercept.

Value

An object of class "anova" inheriting from class "data.frame"

Note

The Huynh–Feldt epsilon differs from that calculated by SAS (as of v. 8.2) except when the DF is equal to the number of observations minus one. This is believed to be a bug in SAS, not in R.

References

Hand, D. J. and Taylor, C. C. (1987) *Multivariate Analysis of Variance and Repeated Measures.* Chapman and Hall.

See Also

summary.manova

Examples

```
require(graphics)
utils::example(SSD) # Brings in the mlmfit and reacttime objects

mlmfit0 <- update(mlmfit, ~0)

### Traditional tests of intrasubj. contrasts
## Using MANOVA techniques on contrasts:
anova(mlmfit, mlmfit0, X=~1)

## Assuming sphericity
anova(mlmfit, mlmfit0, X=~1, test="Spherical")

### tests using intra-subject 3x2 design
idata <- data.frame(deg=gl(3,1,6,labels=c(0,4,8)),
                    noise=gl(2,3,6,labels=c("A","P")))

anova(mlmfit, mlmfit0, X = ~ deg + noise,
      idata = idata, test = "Spherical")
anova(mlmfit, mlmfit0, M = ~ deg + noise, X = ~ noise,
      idata = idata, test="Spherical" )
anova(mlmfit, mlmfit0, M = ~ deg + noise, X = ~ deg,
      idata = idata, test="Spherical" )

f <- factor(rep(1:2,5)) # bogus, just for illustration
mlmfit2 <- update(mlmfit, ~f)
anova(mlmfit2, mlmfit, mlmfit0, X = ~1, test = "Spherical")
anova(mlmfit2, X = ~1, test = "Spherical")
# one-model form, eqiv. to previous

### There seems to be a strong interaction in these data
plot(colMeans(reacttime))
```

---

ansari.test *Ansari-Bradley Test*

Description

Performs the Ansari-Bradley two-sample test for a difference in scale parameters.

Usage

```
ansari.test(x, ...)

## Default S3 method:
ansari.test(x, y,
            alternative = c("two.sided", "less", "greater"),
            exact = NULL, conf.int = FALSE, conf.level = 0.95,
            ...)
```

```
## S3 method for class 'formula':
ansari.test(formula, data, subset, na.action, ...)
```

Arguments

x	numeric vector of data values.
y	numeric vector of data values.
alternative	indicates the alternative hypothesis and must be one of "two.sided", "greater" or "less". You can specify just the initial letter.
exact	a logical indicating whether an exact p-value should be computed.
conf.int	a logical,indicating whether a confidence interval should be computed.
conf.level	confidence level of the interval.
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs a factor with two levels giving the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

Suppose that x and y are independent samples from distributions with densities $f((t-m)/s)/s$ and $f(t-m)$, respectively, where m is an unknown nuisance parameter and s, the ratio of scales, is the parameter of interest. The Ansari-Bradley test is used for testing the null that s equals 1, the two-sided alternative being that $s \neq 1$ (the distributions differ only in variance), and the one-sided alternatives being $s > 1$ (the distribution underlying x has a larger variance, "greater") or $s < 1$ ("less").

By default (if exact is not specified), an exact p-value is computed if both samples contain less than 50 finite values and there are no ties. Otherwise, a normal approximation is used.

Optionally, a nonparametric confidence interval and an estimator for s are computed. If exact p-values are available, an exact confidence interval is obtained by the algorithm described in Bauer (1972), and the Hodges-Lehmann estimator is employed. Otherwise, the returned confidence interval and point estimate are based on normal approximations.

Note that mid-ranks are used in the case of ties rather than average scores as employed in Hollander & Wolfe (1973). See, e.g., Hajek, Sidak and Sen (1999), pages 131ff, for more information.

Value

A list with class "htest" containing the following components:

statistic	the value of the Ansari-Bradley test statistic.
p.value	the p-value of the test.
null.value	the ratio of scales s under the null, 1.
alternative	a character string describing the alternative hypothesis.

`method`	the string `"Ansari-Bradley test"`.
`data.name`	a character string giving the names of the data.
`conf.int`	a confidence interval for the scale parameter. (Only present if argument `conf.int = TRUE`.)
`estimate`	an estimate of the ratio of scales. (Only present if argument `conf.int = TRUE`.)

Note

To compare results of the Ansari-Bradley test to those of the F test to compare two variances (under the assumption of normality), observe that s is the ratio of scales and hence s^2 is the ratio of variances (provided they exist), whereas for the F test the ratio of variances itself is the parameter of interest. In particular, confidence intervals are for s in the Ansari-Bradley test but for s^2 in the F test.

References

David F. Bauer (1972), Constructing confidence sets using rank statistics. *Journal of the American Statistical Association* **67**, 687–690.

Jaroslav Hajek, Zbynek Sidak & Pranab K. Sen (1999), *Theory of Rank Tests*. San Diego, London: Academic Press.

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric Statistical Methods.* New York: John Wiley & Sons. Pages 83–92.

See Also

`fligner.test` for a rank-based (nonparametric) k-sample test for homogeneity of variances; `mood.test` for another rank-based two-sample test for a difference in scale parameters; `var.test` and `bartlett.test` for parametric tests for the homogeneity in variance.

`ansari_test` in package **coin** for exact and approximate *conditional* p-values for the Ansari-Bradley test, as well as different methods for handling ties.

Examples

```
## Hollander & Wolfe (1973, p. 86f):
## Serum iron determination using Hyland control sera
ramsay <- c(111, 107, 100, 99, 102, 106, 109, 108, 104, 99,
            101, 96, 97, 102, 107, 113, 116, 113, 110, 98)
jung.parekh <- c(107, 108, 106, 98, 105, 103, 110, 105, 104,
            100, 96, 108, 103, 104, 114, 114, 113, 108, 106, 99)
ansari.test(ramsay, jung.parekh)

ansari.test(rnorm(10), rnorm(10, 0, 2), conf.int = TRUE)

## try more points - failed in 2.4.1
ansari.test(rnorm(100), rnorm(100, 0, 2), conf.int = TRUE)
```

aov *Fit an Analysis of Variance Model*

Description

Fit an analysis of variance model by a call to `lm` for each stratum.

Usage

```
aov(formula, data = NULL, projections = FALSE, qr = TRUE,
    contrasts = NULL, ...)
```

Arguments

`formula`	A formula specifying the model.
`data`	A data frame in which the variables specified in the formula will be found. If missing, the variables are searched for in the standard way.
`projections`	Logical flag: should the projections be returned?
`qr`	Logical flag: should the QR decomposition be returned?
`contrasts`	A list of contrasts to be used for some of the factors in the formula. These are not used for any `Error` term, and supplying contrasts for factors only in the `Error` term will give a warning.
`...`	Arguments to be passed to `lm`, such as `subset` or `na.action`.

Details

This provides a wrapper to `lm` for fitting linear models to balanced or unbalanced experimental designs.

The main difference from `lm` is in the way `print`, `summary` and so on handle the fit: this is expressed in the traditional language of the analysis of variance rather than that of linear models.

If the formula contains a single `Error` term, this is used to specify error strata, and appropriate models are fitted within each error stratum.

The formula can specify multiple responses.

Weights can be specified by a `weights` argument, but should not be used with an `Error` term, and are incompletely supported (e.g., not by `model.tables`).

Value

An object of class `c("aov", "lm")` or for multiple responses of class `c("maov", "aov", "mlm", "lm")` or for multiple error strata of class `"aovlist"`. There are `print` and `summary` methods available for these.

Note

`aov` is designed for balanced designs, and the results can be hard to interpret without balance: beware that missing values in the response(s) will likely lose the balance. If there are two or more error strata, the methods used are statistically inefficient without balance, and it may be better to use `lme`.

Balance can be checked with the `replications` function.

The default 'contrasts' in R are not orthogonal contrasts, and `aov` and its helper functions will work better with such contrasts: see the examples for how to select these.

Author(s)

The design was inspired by the S function of the same name described in Chambers *et al.* (1992).

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`lm`, `summary.aov`, `replications`, `alias`, `proj`, `model.tables`, `TukeyHSD`

Examples

```
## From Venables and Ripley (2002) p.165.
utils::data(npk, package="MASS")

## Set orthogonal contrasts.
op <- options(contrasts=c("contr.helmert", "contr.poly"))
( npk.aov <- aov(yield ~ block + N*P*K, npk) )
summary(npk.aov)
coefficients(npk.aov)

## to show the effects of re-ordering terms contrast the two fits
aov(yield ~ block + N * P + K, npk)
aov(terms(yield ~ block + N * P + K, keep.order=TRUE), npk)

## as a test, not particularly sensible statistically
npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
npk.aovE
summary(npk.aovE)
options(op)# reset to previous
```

`approxfun` *Interpolation Functions*

Description

Return a list of points which linearly interpolate given data points, or a function performing the linear (or constant) interpolation.

Usage

```
approx   (x, y = NULL, xout, method="linear", n=50,
          yleft, yright, rule = 1, f = 0, ties = mean)

approxfun(x, y = NULL,       method="linear",
          yleft, yright, rule = 1, f = 0, ties = mean)
```

Arguments

x, y	vectors giving the coordinates of the points to be interpolated. Alternatively a single plotting structure can be specified: see xy.coords.
xout	an optional set of values specifying where interpolation is to take place.
method	specifies the interpolation method to be used. Choices are "linear" or "constant".
n	If xout is not specified, interpolation takes place at n equally spaced points spanning the interval [min(x), max(x)].
yleft	the value to be returned when input x values are less than min(x). The default is defined by the value of rule given below.
yright	the value to be returned when input x values are greater than max(x). The default is defined by the value of rule given below.
rule	an integer describing how interpolation is to take place outside the interval [min(x), max(x)]. If rule is 1 then NAs are returned for such points and if it is 2, the value at the closest data extreme is used.
f	For method="constant" a number between 0 and 1 inclusive, indicating a compromise between left- and right-continuous step functions. If y0 and y1 are the values to the left and right of the point then the value is y0*(1-f)+y1*f so that f=0 is right-continuous and f=1 is left-continuous.
ties	Handling of tied x values. Either a function with a single vector argument returning a single number result or the string "ordered".

Details

The inputs can contain missing values which are deleted, so at least two complete (x, y) pairs are required (for method = "linear", one otherwise). If there are duplicated (tied) x values and ties is a function it is applied to the y values for each distinct x value. Useful functions in this context include mean, min, and max. If ties="ordered" the x values are assumed to be already ordered. The first y value will be used for interpolation to the left and the last one for interpolation to the right.

Value

approx returns a list with components x and y, containing n coordinates which interpolate the given data points according to the method (and rule) desired.

The function approxfun returns a function performing (linear or constant) interpolation of the given data points. For a given set of x values, this function will return the corresponding interpolated values. This is often more useful than approx.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

spline and splinefun for spline interpolation.

Examples

```
require(graphics)

x <- 1:10
y <- rnorm(10)
par(mfrow = c(2,1))
plot(x, y, main = "approx(.) and approxfun(.)")
points(approx(x, y), col = 2, pch = "*")
points(approx(x, y, method = "constant"), col = 4, pch = "*")

f <- approxfun(x, y)
curve(f(x), 0, 10, col = "green")
points(x, y)
is.function(fc <- approxfun(x, y, method = "const")) # TRUE
curve(fc(x), 0, 10, col = "darkblue", add = TRUE)

## Show treatment of 'ties' :

x <- c(2,2:4,4,4,5,5,7,7,7)
y <- c(1:6, 5:4, 3:1)
approx(x,y, xout=x)$y # warning
(ay <- approx(x,y, xout=x, ties = "ordered")$y)
stopifnot(ay == c(2,2,3,6,6,6,4,4,1,1,1))
approx(x,y, xout=x, ties = min)$y
approx(x,y, xout=x, ties = max)$y
```

---

ar *Fit Autoregressive Models to Time Series*

Description

Fit an autoregressive time series model to the data, by default selecting the complexity by AIC.

Usage

```
ar(x, aic = TRUE, order.max = NULL,
   method=c("yule-walker", "burg", "ols", "mle", "yw"),
   na.action, series, ...)

ar.burg(x, ...)
## Default S3 method:
ar.burg(x, aic = TRUE, order.max = NULL,
        na.action = na.fail, demean = TRUE, series,
        var.method = 1, ...)
## S3 method for class 'mts':
ar.burg(x, aic = TRUE, order.max = NULL,
        na.action = na.fail, demean = TRUE, series,
        var.method = 1, ...)

ar.yw(x, ...)
## Default S3 method:
```

```
ar.yw(x, aic = TRUE, order.max = NULL,
      na.action = na.fail, demean = TRUE, series, ...)
## S3 method for class 'mts':
ar.yw(x, aic = TRUE, order.max = NULL,
      na.action = na.fail, demean = TRUE, series,
      var.method = 1, ...)

ar.mle(x, aic = TRUE, order.max = NULL, na.action = na.fail,
       demean = TRUE, series, ...)

## S3 method for class 'ar':
predict(object, newdata, n.ahead = 1, se.fit = TRUE, ...)
```

Arguments

x	A univariate or multivariate time series.
aic	Logical flag. If TRUE then the Akaike Information Criterion is used to choose the order of the autoregressive model. If FALSE, the model of order order.max is fitted.
order.max	Maximum order (or order) of model to fit. Defaults to the smaller of $N - 1$ and $10 \log_{10}(N)$ where N is the number of observations except for method="mle" where it is the minimum of this quantity and 12.
method	Character string giving the method used to fit the model. Must be one of the strings in the default argument (the first few characters are sufficient). Defaults to "yule-walker".
na.action	function to be called to handle missing values.
demean	should a mean be estimated during fitting?
series	names for the series. Defaults to deparse(substitute(x)).
var.method	the method to estimate the innovations variance (see 'Details').
...	additional arguments for specific methods.
object	a fit from ar.
newdata	data to which to apply the prediction.
n.ahead	number of steps ahead at which to predict.
se.fit	logical: return estimated standard errors of the prediction error?

Details

For definiteness, note that the AR coefficients have the sign in

$$x_t - \mu = a_1(x_{t-1} - \mu) + \cdots + a_p(x_{t-p} - \mu) + e_t$$

ar is just a wrapper for the functions ar.yw, ar.burg, ar.ols and ar.mle.

Order selection is done by AIC if aic is true. This is problematic, as of the methods here only ar.mle performs true maximum likelihood estimation. The AIC is computed as if the variance estimate were the MLE, omitting the determinant term from the likelihood. Note that this is not the same as the Gaussian likelihood evaluated at the estimated parameter values. In ar.yw the variance matrix of the innovations is computed from the fitted coefficients and the autocovariance of x.

ar.burg allows two methods to estimate the innovations variance and hence AIC. Method 1 is to use the update given by the Levinson-Durbin recursion (Brockwell and Davis, 1991, (8.2.6) on page 242), and follows S-PLUS. Method 2 is the mean of the sum of squares of the forward and backward prediction errors (as in Brockwell and Davis, 1996, page 145). Percival and Walden (1998) discuss both. In the multivariate case the estimated coefficients will depend (slightly) on the variance estimation method.

Remember that ar includes by default a constant in the model, by removing the overall mean of x before fitting the AR model, or (ar.mle) estimating a constant to subtract.

Value

For ar and its methods a list of class "ar" with the following elements:

order	The order of the fitted model. This is chosen by minimizing the AIC if aic=TRUE, otherwise it is order.max.
ar	Estimated autoregression coefficients for the fitted model.
var.pred	The prediction variance: an estimate of the portion of the variance of the time series that is not explained by the autoregressive model.
x.mean	The estimated mean of the series used in fitting and for use in prediction.
x.intercept	(ar.ols only.) The intercept in the model for x - x.mean.
aic	The value of the aic argument.
n.used	The number of observations in the time series.
order.max	The value of the order.max argument.
partialacf	The estimate of the partial autocorrelation function up to lag order.max.
resid	residuals from the fitted model, conditioning on the first order observations. The first order residuals are set to NA. If x is a time series, so is resid.
method	The value of the method argument.
series	The name(s) of the time series.
frequency	The frequency of the time series.
call	The matched call.
asy.var.coef	(univariate case, order > 0.) The asymptotic-theory variance matrix of the coefficient estimates.

For predict.ar, a time series of predictions, or if se.fit = TRUE, a list with components pred, the predictions, and se, the estimated standard errors. Both components are time series.

Note

Only the univariate case of ar.mle is implemented.

Fitting by method="mle" to long series can be very slow.

Author(s)

Martyn Plummer. Univariate case of ar.yw, ar.mle and C code for univariate case of ar.burg by B. D. Ripley.

References

Brockwell, P. J. and Davis, R. A. (1991) *Time Series and Forecasting Methods.* Second edition. Springer, New York. Section 11.4.

Brockwell, P. J. and Davis, R. A. (1996) *Introduction to Time Series and Forecasting.* Springer, New York. Sections 5.1 and 7.6.

Percival, D. P. and Walden, A. T. (1998) *Spectral Analysis for Physical Applications.* Cambridge University Press.

Whittle, P. (1963) On the fitting of multivariate autoregressions and the approximate canonical factorization of a spectral density matrix. *Biometrika* **40**, 129–134.

See Also

`ar.ols`, `arima` for ARMA models; `acf2AR`, for AR construction from the ACF.

`arima.sim` for simulation of AR processes.

Examples

```
ar(lh)
ar(lh, method="burg")
ar(lh, method="ols")
ar(lh, FALSE, 4) # fit ar(4)

(sunspot.ar <- ar(sunspot.year))
predict(sunspot.ar, n.ahead=25)
## try the other methods too

ar(ts.union(BJsales, BJsales.lead))
## Burg is quite different here, as is OLS (see ar.ols)
ar(ts.union(BJsales, BJsales.lead), method="burg")
```

---

ar.ols *Fit Autoregressive Models to Time Series by OLS*

Description

Fit an autoregressive time series model to the data by ordinary least squares, by default selecting the complexity by AIC.

Usage

```
ar.ols(x, aic = TRUE, order.max = NULL, na.action = na.fail,
       demean = TRUE, intercept = demean, series, ...)
```

Arguments

`x`	A univariate or multivariate time series.
`aic`	Logical flag. If `TRUE` then the Akaike Information Criterion is used to choose the order of the autoregressive model. If `FALSE`, the model of order `order.max` is fitted.

order.max	Maximum order (or order) of model to fit. Defaults to $10\log_{10}(N)$ where N is the number of observations.
na.action	function to be called to handle missing values.
demean	should the AR model be for x minus its mean?
intercept	should a separate intercept term be fitted?
series	names for the series. Defaults to deparse(substitute(x)).
...	further arguments to be passed to or from methods.

Details

ar.ols fits the general AR model to a possibly non-stationary and/or multivariate system of series x. The resulting unconstrained least squares estimates are consistent, even if some of the series are non-stationary and/or co-integrated. For definiteness, note that the AR coefficients have the sign in

$$x_t - \mu = a_0 + a_1(x_{t-1} - \mu) + \cdots + a_p(x_{t-p} - \mu) + e_t$$

where a_0 is zero unless intercept is true, and μ is the sample mean if demean is true, zero otherwise.

Order selection is done by AIC if aic is true. This is problematic, as ar.ols does not perform true maximum likelihood estimation. The AIC is computed as if the variance estimate (computed from the variance matrix of the residuals) were the MLE, omitting the determinant term from the likelihood. Note that this is not the same as the Gaussian likelihood evaluated at the estimated parameter values.

Some care is needed if intercept is true and demean is false. Only use this is the series are roughly centred on zero. Otherwise the computations may be inaccurate or fail entirely.

Value

A list of class "ar" with the following elements:

order	The order of the fitted model. This is chosen by minimizing the AIC if aic=TRUE, otherwise it is order.max.
ar	Estimated autoregression coefficients for the fitted model.
var.pred	The prediction variance: an estimate of the portion of the variance of the time series that is not explained by the autoregressive model.
x.mean	The estimated mean (or zero if demean is false) of the series used in fitting and for use in prediction.
x.intercept	The intercept in the model for x - x.mean, or zero if intercept is false.
aic	The value of the aic argument.
n.used	The number of observations in the time series.
order.max	The value of the order.max argument.
partialacf	NULL. For compatibility with ar.
resid	residuals from the fitted model, conditioning on the first order observations. The first order residuals are set to NA. If x is a time series, so is resid.
method	The character string "Unconstrained LS".
series	The name(s) of the time series.
frequency	The frequency of the time series.
call	The matched call.
asy.se.coef	The asymptotic-theory standard errors of the coefficient estimates.

Author(s)

Adrian Trapletti, Brian Ripley.

References

Luetkepohl, H. (1991): *Introduction to Multiple Time Series Analysis.* Springer Verlag, NY, pp. 368–370.

See Also

ar

Examples

```
ar(lh, method="burg")
ar.ols(lh)
ar.ols(lh, FALSE, 4) # fit ar(4)

ar.ols(ts.union(BJsales, BJsales.lead))

x <- diff(log(EuStockMarkets))
ar.ols(x, order.max=6, demean=FALSE, intercept=TRUE)
```

---

arima *ARIMA Modelling of Time Series*

Description

Fit an ARIMA model to a univariate time series.

Usage

```
arima(x, order = c(0, 0, 0),
      seasonal = list(order = c(0, 0, 0), period = NA),
      xreg = NULL, include.mean = TRUE,
      transform.pars = TRUE,
      fixed = NULL, init = NULL,
      method = c("CSS-ML", "ML", "CSS"),
      n.cond, optim.control = list(), kappa = 1e6)
```

Arguments

x	a univariate time series
order	A specification of the non-seasonal part of the ARIMA model: the three components (p, d, q) are the AR order, the degree of differencing, and the MA order.
seasonal	A specification of the seasonal part of the ARIMA model, plus the period (which defaults to frequency(x)). This should be a list with components order and period, but a specification of just a numeric vector of length 3 will be turned into a suitable list with the specification as the order.
xreg	Optionally, a vector or matrix of external regressors, which must have the same number of rows as x.

include.mean Should the ARMA model include a mean/intercept term? The default is TRUE for undifferenced series, and it is ignored for ARIMA models with differencing.

transform.pars
Logical. If true, the AR parameters are transformed to ensure that they remain in the region of stationarity. Not used for method = "CSS".

fixed optional numeric vector of the same length as the total number of parameters. If supplied, only NA entries in fixed will be varied. transform.pars = TRUE will be overridden (with a warning) if any AR parameters are fixed. It may be wise to set transform.pars = FALSE when fixing MA parameters, especially near non-invertibility.

init optional numeric vector of initial parameter values. Missing values will be filled in, by zeroes except for regression coefficients. Values already specified in fixed will be ignored.

method Fitting method: maximum likelihood or minimize conditional sum-of-squares. The default (unless there are missing values) is to use conditional-sum-of-squares to find starting values, then maximum likelihood.

n.cond Only used if fitting by conditional-sum-of-squares: the number of initial observations to ignore. It will be ignored if less than the maximum lag of an AR term.

optim.control
List of control parameters for optim.

kappa the prior variance (as a multiple of the innovations variance) for the past observations in a differenced model. Do not reduce this.

Details

Different definitions of ARMA models have different signs for the AR and/or MA coefficients. The definition used here has

$$X_t = a_1 X_{t-1} + \cdots + a_p X_{t-p} + e_t + b_1 e_{t-1} + \ldots + b_q e_{t-q}$$

and so the MA coefficients differ in sign from those of S-PLUS. Further, if include.mean is true (the default for an ARMA model), this formula applies to $X - m$ rather than X. For ARIMA models with differencing, the differenced series follows a zero-mean ARMA model. If am xreg term is included, a linear regression (with a constant term if include.mean is true and there is no differencing) is fitted with an ARMA model for the error term.

The variance matrix of the estimates is found from the Hessian of the log-likelihood, and so may only be a rough guide.

Optimization is done by optim. It will work best if the columns in xreg are roughly scaled to zero mean and unit variance, but does attempt to estimate suitable scalings.

Value

A list of class "Arima" with components:

coef a vector of AR, MA and regression coefficients, which can be extracted by the coef method.

sigma2 the MLE of the innovations variance.

var.coef the estimated variance matrix of the coefficients coef, which can be extracted by the vcov method.

loglik	the maximized log-likelihood (of the differenced data), or the approximation to it used.
arma	A compact form of the specification, as a vector giving the number of AR, MA, seasonal AR and seasonal MA coefficients, plus the period and the number of non-seasonal and seasonal differences.
aic	the AIC value corresponding to the log-likelihood. Only valid for method = "ML" fits.
residuals	the fitted innovations.
call	the matched call.
series	the name of the series x.
code	the convergence value returned by optim.
n.cond	the number of initial observations not used in the fitting.
model	A list representing the Kalman Filter used in the fitting. See KalmanLike.

Fitting methods

The exact likelihood is computed via a state-space representation of the ARIMA process, and the innovations and their variance found by a Kalman filter. The initialization of the differenced ARMA process uses stationarity and is based on Gardner *et al.* (1980). For a differenced process the non-stationary components are given a diffuse prior (controlled by kappa). Observations which are still controlled by the diffuse prior (determined by having a Kalman gain of at least 1e4) are excluded from the likelihood calculations. (This gives comparable results to arima0 in the absence of missing values, when the observations excluded are precisely those dropped by the differencing.)

Missing values are allowed, and are handled exactly in method "ML".

If transform.pars is true, the optimization is done using an alternative parametrization which is a variation on that suggested by Jones (1980) and ensures that the model is stationary. For an AR(p) model the parametrization is via the inverse tanh of the partial autocorrelations: the same procedure is applied (separately) to the AR and seasonal AR terms. The MA terms are not constrained to be invertible during optimization, but they will be converted to invertible form after optimization if transform.pars is true.

Conditional sum-of-squares is provided mainly for expositional purposes. This computes the sum of squares of the fitted innovations from observation n.cond on, (where n.cond is at least the maximum lag of an AR term), treating all earlier innovations to be zero. Argument n.cond can be used to allow comparability between different fits. The 'part log-likelihood' is the first term, half the log of the estimated mean square. Missing values are allowed, but will cause many of the innovations to be missing.

When regressors are specified, they are orthogonalized prior to fitting unless any of the coefficients is fixed. It can be helpful to roughly scale the regressors to zero mean and unit variance.

Note

The results are likely to be different from S-PLUS's arima.mle, which computes a conditional likelihood and does not include a mean in the model. Further, the convention used by arima.mle reverses the signs of the MA coefficients.

arima is very similar to arima0 for ARMA models or for differenced models without missing values, but handles differenced models with missing values exactly. It is somewhat slower than arima0, particularly for seasonally differenced models.

References

Brockwell, P. J. and Davis, R. A. (1996) *Introduction to Time Series and Forecasting.* Springer, New York. Sections 3.3 and 8.3.

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press.

Gardner, G, Harvey, A. C. and Phillips, G. D. A. (1980) Algorithm AS154. An algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of Kalman filtering. *Applied Statistics* **29**, 311–322.

Harvey, A. C. (1993) *Time Series Models*, 2nd Edition, Harvester Wheatsheaf, sections 3.3 and 4.4.

Jones, R. H. (1980) Maximum likelihood fitting of ARMA models to time series with missing observations. *Technometrics* **20** 389–395.

See Also

`predict.Arima`, `arima.sim` for simulating from an ARIMA model, `tsdiag`, `arima0`, `ar`

Examples

```
arima(lh, order = c(1,0,0))
arima(lh, order = c(3,0,0))
arima(lh, order = c(1,0,1))

arima(lh, order = c(3,0,0), method = "CSS")

arima(USAccDeaths, order = c(0,1,1), seasonal = list(order=c(0,1,1)))
arima(USAccDeaths, order = c(0,1,1), seasonal = list(order=c(0,1,1)),
      method = "CSS") # drops first 13 observations.
# for a model with as few years as this, we want full ML

arima(LakeHuron, order = c(2,0,0), xreg = time(LakeHuron)-1920)

## presidents contains NAs
## graphs in example(acf) suggest order 1 or 3
require(graphics)
(fit1 <- arima(presidents, c(1, 0, 0)))
tsdiag(fit1)
(fit3 <- arima(presidents, c(3, 0, 0)))  # smaller AIC
tsdiag(fit3)
```

`arima.sim` *Simulate from an ARIMA Model*

Description

Simulate from an ARIMA model.

Usage

```
arima.sim(model, n, rand.gen = rnorm, innov = rand.gen(n, ...),
          n.start = NA, start.innov = rand.gen(n.start, ...),
          ...)
```

Arguments

model	A list with component ar and/or ma giving the AR and MA coefficients respectively. Optionally a component order can be used. An empty list gives an ARIMA(0, 0, 0) model, that is white noise.
n	length of output series, before un-differencing.
rand.gen	optional: a function to generate the innovations.
innov	an optional times series of innovations. If not provided, rand.gen is used.
n.start	length of 'burn-in' period. If NA, the default, a reasonable value is computed.
start.innov	an optional times series of innovations to be used for the burn-in period. If supplied there must be at least n.start values (and n.start is by default computed inside the function).
...	additional arguments for rand.gen. Most usefully, the standard deviation of the innovations generated by rnorm can be specified by sd.

Details

See arima for the precise definition of an ARIMA model.

The ARMA model is checked for stationarity.

ARIMA models are specified via the order component of model, in the same way as for arima. Other aspects of the order component are ignored, but inconsistent specifications of the MA and AR orders are detected. The un-differencing assumes previous values of zero, and to remind the user of this, those values are returned.

Random inputs for the 'burn-in' period are generated by calling rand.gen.

Value

A time-series object of class "ts".

See Also

arima

Examples

```
require(graphics)

arima.sim(n = 63, list(ar = c(0.8897, -0.4858), ma = c(-0.2279, 0.2488)),
          sd = sqrt(0.1796))
# mildly long-tailed
arima.sim(n = 63, list(ar=c(0.8897, -0.4858), ma=c(-0.2279, 0.2488)),
          rand.gen = function(n, ...) sqrt(0.1796) * rt(n, df = 5))

# An ARIMA simulation
ts.sim <- arima.sim(list(order = c(1,1,0), ar = 0.7), n = 200)
ts.plot(ts.sim)
```

`arima0` *ARIMA Modelling of Time Series – Preliminary Version*

Description

Fit an ARIMA model to a univariate time series, and forecast from the fitted model.

Usage

```
arima0(x, order = c(0, 0, 0),
       seasonal = list(order = c(0, 0, 0), period = NA),
       xreg = NULL, include.mean = TRUE, delta = 0.01,
       transform.pars = TRUE, fixed = NULL, init = NULL,
       method = c("ML", "CSS"), n.cond, optim.control = list())

## S3 method for class 'arima0':
predict(object, n.ahead = 1, newxreg, se.fit = TRUE, ...)
```

Arguments

`x`	a univariate time series
`order`	A specification of the non-seasonal part of the ARIMA model: the three components (p, d, q) are the AR order, the degree of differencing, and the MA order.
`seasonal`	A specification of the seasonal part of the ARIMA model, plus the period (which defaults to `frequency(x)`). This should be a list with components `order` and `period`, but a specification of just a numeric vector of length 3 will be turned into a suitable list with the specification as the `order`.
`xreg`	Optionally, a vector or matrix of external regressors, which must have the same number of rows as `x`.
`include.mean`	Should the ARIMA model include a mean term? The default is `TRUE` for undifferenced series, `FALSE` for differenced ones (where a mean would not affect the fit nor predictions).
`delta`	A value to indicate at which point 'fast recursions' should be used. See the 'Details' section.
`transform.pars`	Logical. If true, the AR parameters are transformed to ensure that they remain in the region of stationarity. Not used for `method = "CSS"`.
`fixed`	optional numeric vector of the same length as the total number of parameters. If supplied, only `NA` entries in `fixed` will be varied. `transform.pars = TRUE` will be overridden (with a warning) if any ARMA parameters are fixed.
`init`	optional numeric vector of initial parameter values. Missing values will be filled in, by zeroes except for regression coefficients. Values already specified in `fixed` will be ignored.
`method`	Fitting method: maximum likelihood or minimize conditional sum-of-squares.
`n.cond`	Only used if fitting by conditional-sum-of-squares: the number of initial observations to ignore. It will be ignored if less than the maximum lag of an AR term.

optim.control	List of control parameters for optim.
object	The result of an arima0 fit.
newxreg	New values of xreg to be used for prediction. Must have at least n.ahead rows.
n.ahead	The number of steps ahead for which prediction is required.
se.fit	Logical: should standard errors of prediction be returned?
...	arguments passed to or from other methods.

Details

Different definitions of ARMA models have different signs for the AR and/or MA coefficients. The definition here has

$$X_t = a_1 X_{t-1} + \cdots + a_p X_{t-p} + e_t + b_1 e_{t-1} + \ldots + b_q e_{t-q}$$

and so the MA coefficients differ in sign from those of S-PLUS. Further, if include.mean is true, this formula applies to $X - m$ rather than X. For ARIMA models with differencing, the differenced series follows a zero-mean ARMA model.

The variance matrix of the estimates is found from the Hessian of the log-likelihood, and so may only be a rough guide, especially for fits close to the boundary of invertibility.

Optimization is done by optim. It will work best if the columns in xreg are roughly scaled to zero mean and unit variance, but does attempt to estimate suitable scalings.

Finite-history prediction is used. This is only statistically efficient if the MA part of the fit is invertible, so predict.arima0 will give a warning for non-invertible MA models.

Value

For arima0, a list of class "arima0" with components:

coef	a vector of AR, MA and regression coefficients,
sigma2	the MLE of the innovations variance.
var.coef	the estimated variance matrix of the coefficients coef.
loglik	the maximized log-likelihood (of the differenced data), or the approximation to it used.
arma	A compact form of the specification, as a vector giving the number of AR, MA, seasonal AR and seasonal MA coefficients, plus the period and the number of non-seasonal and seasonal differences.
aic	the AIC value corresponding to the log-likelihood. Only valid for method = "ML" fits.
residuals	the fitted innovations.
call	the matched call.
series	the name of the series x.
convergence	the value returned by optim.
n.cond	the number of initial observations not used in the fitting.

For predict.arima0, a time series of predictions, or if se.fit = TRUE, a list with components pred, the predictions, and se, the estimated standard errors. Both components are time series.

Fitting methods

The exact likelihood is computed via a state-space representation of the ARMA process, and the innovations and their variance found by a Kalman filter based on Gardner *et al.* (1980). This has the option to switch to 'fast recursions' (assume an effectively infinite past) if the innovations variance is close enough to its asymptotic bound. The argument `delta` sets the tolerance: at its default value the approximation is normally negligible and the speed-up considerable. Exact computations can be ensured by setting `delta` to a negative value.

If `transform.pars` is true, the optimization is done using an alternative parametrization which is a variation on that suggested by Jones (1980) and ensures that the model is stationary. For an AR(p) model the parametrization is via the inverse tanh of the partial autocorrelations: the same procedure is applied (separately) to the AR and seasonal AR terms. The MA terms are also constrained to be invertible during optimization by the same transformation if `transform.pars` is true. Note that the MLE for MA terms does sometimes occur for MA polynomials with unit roots: such models can be fitted by using `transform.pars = FALSE` and specifying a good set of initial values (often obtainable from a fit with `transform.pars = TRUE`).

As from R 1.5.0 missing values are allowed, but any missing values will force `delta` to be ignored and full recursions used. Note that missing values will be propagated by differencing, so the procedure used in this function is not fully efficient in that case.

Conditional sum-of-squares is provided mainly for expositional purposes. This computes the sum of squares of the fitted innovations from observation `n.cond` on, (where `n.cond` is at least the maximum lag of an AR term), treating all earlier innovations to be zero. Argument `n.cond` can be used to allow comparability between different fits. The 'part log-likelihood' is the first term, half the log of the estimated mean square. Missing values are allowed, but will cause many of the innovations to be missing.

When regressors are specified, they are orthogonalized prior to fitting unless any of the coefficients is fixed. It can be helpful to roughly scale the regressors to zero mean and unit variance.

Note

This is a preliminary version, and will be replaced by `arima`.

The standard errors of prediction exclude the uncertainty in the estimation of the ARMA model and the regression coefficients.

The results are likely to be different from S-PLUS's `arima.mle`, which computes a conditional likelihood and does not include a mean in the model. Further, the convention used by `arima.mle` reverses the signs of the MA coefficients.

References

Brockwell, P. J. and Davis, R. A. (1996) *Introduction to Time Series and Forecasting.* Springer, New York. Sections 3.3 and 8.3.

Gardner, G, Harvey, A. C. and Phillips, G. D. A. (1980) Algorithm AS154. An algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of Kalman filtering. *Applied Statistics* **29**, 311–322.

Harvey, A. C. (1993) *Time Series Models*, 2nd Edition, Harvester Wheatsheaf, sections 3.3 and 4.4.

Harvey, A. C. and McKenzie, C. R. (1982) Algorithm AS182. An algorithm for finite sample prediction from ARIMA processes. *Applied Statistics* **31**, 180–187.

Jones, R. H. (1980) Maximum likelihood fitting of ARMA models to time series with missing observations. *Technometrics* **20** 389–395.

See Also

arima, ar, tsdiag

Examples

```
## Not run: arima0(lh, order = c(1,0,0))
arima0(lh, order = c(3,0,0))
arima0(lh, order = c(1,0,1))
predict(arima0(lh, order = c(3,0,0)), n.ahead = 12)

arima0(lh, order = c(3,0,0), method = "CSS")

# for a model with as few years as this, we want full ML
(fit <- arima0(USAccDeaths, order = c(0,1,1),
               seasonal = list(order=c(0,1,1)), delta = -1))
predict(fit, n.ahead = 6)

arima0(LakeHuron, order = c(2,0,0), xreg = time(LakeHuron)-1920)
## Not run:
## presidents contains NAs
## graphs in example(acf) suggest order 1 or 3
(fit1 <- arima0(presidents, c(1, 0, 0), delta = -1))  # avoid warning
tsdiag(fit1)
(fit3 <- arima0(presidents, c(3, 0, 0), delta = -1))  # smaller AIC
tsdiag(fit3)
## End(Not run)
```

ARMAacf *Compute Theoretical ACF for an ARMA Process*

Description

Compute the theoretical autocorrelation function or partial autocorrelation function for an ARMA process.

Usage

```
ARMAacf(ar = numeric(0), ma = numeric(0), lag.max = r, pacf = FALSE)
```

Arguments

ar	numeric vector of AR coefficients
ma	numeric vector of MA coefficients
lag.max	integer. Maximum lag required. Defaults to max(p, q+1), where p, q are the numbers of AR and MA terms respectively.
pacf	logical. Should the partial autocorrelations be returned?

Details

The methods used follow Brockwell & Davis (1991, section 3.3). Their equations (3.3.8) are solved for the autocovariances at lags $0, \ldots, \max(p, q+1)$, and the remaining autocorrelations are given by a recursive filter.

Value

A vector of (partial) autocorrelations, named by the lags.

References

Brockwell, P. J. and Davis, R. A. (1991) *Time Series: Theory and Methods*, Second Edition. Springer.

See Also

`arima`, `ARMAtoMA`, `acf2AR` for inverting part of `ARMAacf`; further `filter`.

Examples

```
ARMAacf(c(1.0, -0.25), 1.0, lag.max = 10)

## Example from Brockwell & Davis (1991, pp.92-4)
## answer 2^(-n) * (32/3 + 8 * n) /(32/3)
n <- 1:10; 2^(-n) * (32/3 + 8 * n) /(32/3)
ARMAacf(c(1.0, -0.25), 1.0, lag.max = 10, pacf = TRUE)
ARMAacf(c(1.0, -0.25), lag.max = 10, pacf = TRUE)

## Cov-Matrix of length-7 sub-sample of AR(1) example:
toeplitz(ARMAacf(0.8, lag.max = 7))
```

`ARMAtoMA` *Convert ARMA Process to Infinite MA Process*

Description

Convert ARMA process to infinite MA process.

Usage

```
ARMAtoMA(ar = numeric(0), ma = numeric(0), lag.max)
```

Arguments

`ar`	numeric vector of AR coefficients
`ma`	numeric vector of MA coefficients
`lag.max`	Largest MA(Inf) coefficient required.

Value

A vector of coefficients.

References

Brockwell, P. J. and Davis, R. A. (1991) *Time Series: Theory and Methods*, Second Edition. Springer.

See Also

arima, ARMAacf.

Examples

```
ARMAtoMA(c(1.0, -0.25), 1.0, 10)
## Example from Brockwell & Davis (1991, p.92)
## answer (1 + 3*n)*2^(-n)
n <- 1:10; (1 + 3*n)*2^(-n)
```

---

as.hclust *Convert Objects to Class hclust*

Description

Converts objects from other hierarchical clustering functions to class "hclust".

Usage

```
as.hclust(x, ...)
```

Arguments

x	Hierarchical clustering object
...	further arguments passed to or from other methods.

Details

Currently there is only support for converting objects of class "twins" as produced by the functions diana and agnes from the package **cluster**. The default method throws an error unless passed an "hclust" object.

Value

An object of class "hclust".

See Also

hclust, diana, agnes

Examples

```
x <- matrix(rnorm(30), ncol=3)
hc <- hclust(dist(x), method="complete")

if(require(cluster, quietly=TRUE)) {# is a recommended package
  ag <- agnes(x, method="complete")
  hcag <- as.hclust(ag)
  ## The dendrograms order slightly differently:
  op <- par(mfrow=c(1,2))
  plot(hc) ;  mtext("hclust", side=1)
  plot(hcag); mtext("agnes",  side=1)
}
```

asOneSidedFormula *Convert to One-Sided Formula*

Description

Names, expressions, numeric values, and character strings are converted to one-sided formulae. If `object` is a formula, it must be one-sided, in which case it is returned unaltered.

Usage

```
asOneSidedFormula(object)
```

Arguments

`object`	a one-sided formula, an expression, a numeric value, or a character string.

Value

a one-sided formula representing `object`

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`formula`

Examples

```
asOneSidedFormula("age")
asOneSidedFormula(~ age)
```

ave *Group Averages Over Level Combinations of Factors*

Description

Subsets of `x[]` are averaged, where each subset consist of those observations with the same factor levels.

Usage

```
ave(x, ..., FUN = mean)
```

Arguments

`x`	A numeric.
`...`	Grouping variables, typically factors, all of the same `length` as `x`.
`FUN`	Function to apply for each factor level combination.

Value

A numeric vector, say `y` of length `length(x)`. If `...` is `g1,g2`, e.g., `y[i]` is equal to `FUN(x[j]`, for all `j` with `g1[j] == g1[i]` and `g2[j] == g2[i])`.

See Also

`mean`, `median`.

Examples

```
require(graphics)

ave(1:3)# no grouping -> grand mean

attach(warpbreaks)
ave(breaks, wool)
ave(breaks, tension)
ave(breaks, tension, FUN = function(x)mean(x, trim=.1))
plot(breaks, main =
     "ave( Warpbreaks )  for   wool  x  tension  combinations")
lines(ave(breaks, wool, tension           ), type='s', col = "blue")
lines(ave(breaks, wool, tension, FUN=median), type='s', col = "green")
legend(40,70, c("mean","median"), lty=1,col=c("blue","green"), bg="gray90")
detach()
```

`bandwidth` *Bandwidth Selectors for Kernel Density Estimation*

Description

Bandwidth selectors for gaussian windows in `density`.

Usage

```
bw.nrd0(x)

bw.nrd(x)

bw.ucv(x, nb = 1000, lower = 0.1 * hmax, upper = hmax)

bw.bcv(x, nb = 1000, lower = 0.1 * hmax, upper = hmax)

bw.SJ(x, nb = 1000, lower = 0.1 * hmax, upper = hmax,
      method = c("ste", "dpi"))
```

Arguments

`x`	A data vector.
`nb`	number of bins to use.
`lower, upper`	Range over which to minimize. The default is almost always satisfactory. `hmax` is calculated internally from a normal reference bandwidth.
`method`	Either `"ste"` ("solve-the-equation") or `"dpi"` ("direct plug-in").

Details

`bw.nrd0` implements a rule-of-thumb for choosing the bandwidth of a Gaussian kernel density estimator. It defaults to 0.9 times the minimum of the standard deviation and the interquartile range divided by 1.34 times the sample size to the negative one-fifth power (= Silverman's 'rule of thumb', Silverman (1986, page 48, eqn (3.31)) *unless* the quartiles coincide when a positive result will be guaranteed.

`bw.nrd` is the more common variation given by Scott (1992), using factor 1.06.

`bw.ucv` and `bw.bcv` implement unbiased and biased cross-validation respectively.

`bw.SJ` implements the methods of Sheather & Jones (1991) to select the bandwidth using pilot estimation of derivatives.
The algorithm solves an equation (via `uniroot`) and because of that, enlarges the interval `c(lower,upper)` when the boundaries were not user-specified and do not bracket the root.

Value

A bandwidth on a scale suitable for the `bw` argument of `density`.

References

Scott, D. W. (1992) *Multivariate Density Estimation: Theory, Practice, and Visualization.* Wiley.

Sheather, S. J. and Jones, M. C. (1991) A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society series B*, **53**, 683–690.

Silverman, B. W. (1986) *Density Estimation.* London: Chapman and Hall.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Springer.

See Also

`density`.

`bandwidth.nrd`, `ucv`, `bcv` and `width.SJ` in package **MASS**, which are all scaled to the `width` argument of `density` and so give answers four times as large.

Examples

```
require(graphics)

plot(density(precip, n = 1000))
rug(precip)
lines(density(precip, bw="nrd"), col = 2)
lines(density(precip, bw="ucv"), col = 3)
lines(density(precip, bw="bcv"), col = 4)
lines(density(precip, bw="SJ-ste"), col = 5)
lines(density(precip, bw="SJ-dpi"), col = 6)
legend(55, 0.035,
       legend = c("nrd0", "nrd", "ucv", "bcv", "SJ-ste", "SJ-dpi"),
       col = 1:6, lty = 1)
```

`bartlett.test` *Bartlett Test of Homogeneity of Variances*

Description

Performs Bartlett's test of the null that the variances in each of the groups (samples) are the same.

Usage

```
bartlett.test(x, ...)

## Default S3 method:
bartlett.test(x, g, ...)

## S3 method for class 'formula':
bartlett.test(formula, data, subset, na.action, ...)
```

Arguments

`x`	a numeric vector of data values, or a list of numeric data vectors representing the respective samples, or fitted linear model objects (inheriting from class `"lm"`).
`g`	a vector or factor object giving the group for the corresponding elements of `x`. Ignored if `x` is a list.
`formula`	a formula of the form `lhs ~ rhs` where `lhs` gives the data values and `rhs` the corresponding groups.
`data`	an optional matrix or data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`subset`	an optional vector specifying a subset of observations to be used.
`na.action`	a function which indicates what should happen when the data contain `NA`s. Defaults to `getOption("na.action")`.
`...`	further arguments to be passed to or from methods.

Details

If `x` is a list, its elements are taken as the samples or fitted linear models to be compared for homogeneity of variances. In this case, the elements must either all be numeric data vectors or fitted linear model objects, `g` is ignored, and one can simply use `bartlett.test(x)` to perform the test. If the samples are not yet contained in a list, use `bartlett.test(list(x, ...))`.

Otherwise, `x` must be a numeric data vector, and `g` must be a vector or factor object of the same length as `x` giving the group for the corresponding elements of `x`.

Value

A list of class `"htest"` containing the following components:

`statistic`	Bartlett's K-squared test statistic.
`parameter`	the degrees of freedom of the approximate chi-squared distribution of the test statistic.

`p.value`	the p-value of the test.
`method`	the character string `"Bartlett test of homogeneity of variances"`.
`data.name`	a character string giving the names of the data.

References

Bartlett, M. S. (1937). Properties of sufficiency and statistical tests. *Proceedings of the Royal Society of London Series A* **160**, 268–282.

See Also

`var.test` for the special case of comparing variances in two samples from normal distributions; `fligner.test` for a rank-based (nonparametric) k-sample test for homogeneity of variances; `ansari.test` and `mood.test` for two rank based two-sample tests for difference in scale.

Examples

```
require(graphics)

plot(count ~ spray, data = InsectSprays)
bartlett.test(InsectSprays$count, InsectSprays$spray)
bartlett.test(count ~ spray, data = InsectSprays)
```

`Beta` *The Beta Distribution*

Description

Density, distribution function, quantile function and random generation for the Beta distribution with parameters `shape1` and `shape2` (and optional non-centrality parameter `ncp`).

Usage

```
dbeta(x, shape1, shape2, ncp = 0, log = FALSE)
pbeta(q, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qbeta(p, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE)
rbeta(n, shape1, shape2, ncp = 0)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`shape1, shape2`	positive parameters of the Beta distribution.
`ncp`	non-centrality parameter.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The Beta distribution with parameters `shape1` $= a$ and `shape2` $= b$ has density

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^a (1-x)^b$$

for $a > 0$, $b > 0$ and $0 \le x \le 1$ where the boundary values at $x = 0$ or $x = 1$ are defined as by continuity (as limits).
The mean is $a/(a+b)$ and the variance is $ab/((a+b)^2(a+b+1))$.

`pbeta` is closely related to the incomplete beta function. As defined by Abramowitz and Stegun 6.6.1

$$B_x(a,b) = \int_0^x t^{a-1}(1-t)^{b-1} dt,$$

and 6.6.2 $I_x(a,b) = B_x(a,b)/B(a,b)$ where $B(a,b) = B_1(a,b)$ is the Beta function (`beta`).

$I_x(a,b)$ is `pbeta(x,a,b)`.

The noncentral Beta distribution (with `ncp` $= \lambda$) is defined (Johnson et al, 1995, pp. 502) as the distribution of $X/(X+Y)$ where $X \sim \chi^2_{2a}(\lambda)$ and $Y \sim \chi^2_{2b}$.

Value

`dbeta` gives the density, `pbeta` the distribution function, `qbeta` the quantile function, and `rbeta` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Source

The central `dbeta` is based on a binomial probability, using code contributed by Catherine Loader (see `dbinom`) if either shape parameter is larger than one, otherwise directly from the definition. The non-central case is based on the derivation as a Poisson mixture of betas (Johnson *et al*, 1995, pp. 502–3).

The central `pbeta` uses a C translation (and enhancement for `log_p=TRUE`) of

Didonato, A. and Morris, A., Jr, (1992) Algorithm 708: Significant digit computation of the incomplete beta function ratios, *ACM Transactions on Mathematical Software*, **18**, 360–373. (See also Brown, B. and Lawrence Levy, L. (1994) Certification of algorithm 708: Significant digit computation of the incomplete beta, *ACM Transactions on Mathematical Software*, **20**, 393–397.)

The non-central `pbeta` uses a C translation of

Lenth, R. V. (1987) Algorithm AS226: Computing noncentral beta probabilities. *Appl. Statist*, **36**, 241–244, incorporating
Frick, H. (1990)'s AS R84, *Appl. Statist*, **39**, 311–2, and
Lam, M.L. (1995)'s AS R95, *Appl. Statist*, **44**, 551–2.

`qbeta` is based on a C translation of

Cran, G. W., K. J. Martin and G. E. Thomas (1977). Remark AS R19 and Algorithm AS 109, *Applied Statistics*, **26**, 111–114, and subsequent remarks (AS83 and correction).

`rbeta` is based on a C translation of

R. C. H. Cheng (1978). Generating beta variates with nonintegral shape parameters. *Communications of the ACM*, **21**, 317–322.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* New York: Dover. Chapter 6: Gamma and Related Functions.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 2, especially chapter 25. Wiley, New York.

See Also

`beta` for the Beta function, and `dgamma` for the Gamma distribution.

Examples

```
x <- seq(0, 1, length=21)
dbeta(x, 1, 1)
pbeta(x, 1, 1)
```

`binom.test` *Exact Binomial Test*

Description

Performs an exact test of a simple null hypothesis about the probability of success in a Bernoulli experiment.

Usage

```
binom.test(x, n, p = 0.5,
           alternative = c("two.sided", "less", "greater"),
           conf.level = 0.95)
```

Arguments

`x`	number of successes, or a vector of length 2 giving the numbers of successes and failures, respectively.
`n`	number of trials; ignored if `x` has length 2.
`p`	hypothesized probability of success.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter.
`conf.level`	confidence level for the returned confidence interval.

Details

Confidence intervals are obtained by a procedure first given in Clopper and Pearson (1934). This guarantees that the confidence level is at least `conf.level`, but in general does not give the shortest-length confidence intervals.

Value

A list with class "htest" containing the following components:

statistic	the number of successes.
parameter	the number of trials.
p.value	the p-value of the test.
conf.int	a confidence interval for the probability of success.
estimate	the estimated probability of success.
null.value	the probability of success under the null, p.
alternative	a character string describing the alternative hypothesis.
method	the character string "Exact binomial test".
data.name	a character string giving the names of the data.

References

Clopper, C. J. & Pearson, E. S. (1934). The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, **26**, 404–413.

William J. Conover (1971), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 97–104.

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric Statistical Methods.* New York: John Wiley & Sons. Pages 15–22.

See Also

prop.test for a general (approximate) test for equal or given proportions.

Examples

```
## Conover (1971), p. 97f.
## Under (the assumption of) simple Mendelian inheritance, a cross
##  between plants of two particular genotypes produces progeny 1/4 of
##  which are "dwarf" and 3/4 of which are "giant", respectively.
##  In an experiment to determine if this assumption is reasonable, a
##  cross results in progeny having 243 dwarf and 682 giant plants.
##  If "giant" is taken as success, the null hypothesis is that p =
##  3/4 and the alternative that p != 3/4.
binom.test(c(682, 243), p = 3/4)
binom.test(682, 682 + 243, p = 3/4)   # The same.
## => Data are in agreement with the null hypothesis.
```

Binomial *The Binomial Distribution*

Description

Density, distribution function, quantile function and random generation for the binomial distribution with parameters size and prob.

Usage

```
dbinom(x, size, prob, log = FALSE)
pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)
qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)
rbinom(n, size, prob)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`size`	number of trials (zero or more).
`prob`	probability of success on each trial.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

The binomial distribution with `size` $= n$ and `prob` $= p$ has density

$$p(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

for $x = 0, \ldots, n$.

If an element of `x` is not integer, the result of `dbinom` is zero, with a warning. $p(x)$ is computed using Loader's algorithm, see the reference below.

The quantile is defined as the smallest value x such that $F(x) \ge p$, where F is the distribution function.

Value

`dbinom` gives the density, `pbinom` gives the distribution function, `qbinom` gives the quantile function and `rbinom` generates random deviates.

If `size` is not an integer, `NaN` is returned.

Source

For `dbinom` a saddle-point expansion is used: see

Catherine Loader (2000). *Fast and Accurate Computation of Binomial Probabilities*; available from `http://www.herine.net/stat/software/dbinom.html`.

`pbinom` uses `pbeta`.

`qbinom` uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

`rbinom` (for `size < .Machine$integer.max`) is based on

Kachitvichyanukul, V. and Schmeiser, B. W. (1988) Binomial random variate generation. *Communications of the ACM*, **31**, 216–222.

See Also

dnbinom for the negative binomial, and dpois for the Poisson distribution.

Examples

```
require(graphics)
# Compute P(45 < X < 55) for X Binomial(100,0.5)
sum(dbinom(46:54, 100, 0.5))

## Using "log = TRUE" for an extended range :
n <- 2000
k <- seq(0, n, by = 20)
plot (k, dbinom(k, n, pi/10, log=TRUE), type='l', ylab="log density",
      main = "dbinom(*, log=TRUE) is better than  log(dbinom(*))")
lines(k, log(dbinom(k, n, pi/10)), col='red', lwd=2)
## extreme points are omitted since dbinom gives 0.
mtext("dbinom(k, log=TRUE)", adj=0)
mtext("extended range", adj=0, line = -1, font=4)
mtext("log(dbinom(k))", col="red", adj=1)
```

---

biplot *Biplot of Multivariate Data*

Description

Plot a biplot on the current graphics device.

Usage

```
biplot(x, ...)

## Default S3 method:
biplot(x, y, var.axes = TRUE, col, cex = rep(par("cex"), 2),
       xlabs = NULL, ylabs = NULL, expand = 1,
       xlim  = NULL, ylim  = NULL, arrow.len = 0.1,
       main = NULL, sub = NULL, xlab = NULL, ylab = NULL, ...)
```

Arguments

x	The biplot, a fitted object. For biplot.default, the first set of points (a two-column matrix), usually associated with observations.
y	The second set of points (a two-column matrix), usually associated with variables.
var.axes	If TRUE the second set of points have arrows representing them as (unscaled) axes.
col	A vector of length 2 giving the colours for the first and second set of points respectively (and the corresponding axes). If a single colour is specified it will be used for both sets. If missing the default colour is looked for in the palette: if there it and the next colour as used, otherwise the first two colours of the palette are used.

`cex`	The character expansion factor used for labelling the points. The labels can be of different sizes for the two sets by supplying a vector of length two.
`xlabs`	A vector of character strings to label the first set of points: the default is to use the row dimname of `x`, or `1:n` is the dimname is `NULL`.
`ylabs`	A vector of character strings to label the second set of points: the default is to use the row dimname of `y`, or `1:n` is the dimname is `NULL`.
`expand`	An expansion factor to apply when plotting the second set of points relative to the first. This can be used to tweak the scaling of the two sets to a physically comparable scale.
`arrow.len`	The length of the arrow heads on the axes plotted in `var.axes` is true. The arrow head can be suppressed by `arrow.len = 0`.
`xlim, ylim`	Limits for the x and y axes in the units of the first set of variables.
`main, sub, xlab, ylab, ...`	graphical parameters.

Details

A biplot is plot which aims to represent both the observations and variables of a matrix of multivariate data on the same plot. There are many variations on biplots (see the references) and perhaps the most widely used one is implemented by `biplot.princomp`. The function `biplot.default` merely provides the underlying code to plot two sets of variables on the same figure.

Graphical parameters can also be given to `biplot`: the size of `xlabs` and `ylabs` is controlled by `cex`.

Side Effects

a plot is produced on the current graphics device.

References

K. R. Gabriel (1971). The biplot graphical display of matrices with application to principal component analysis. *Biometrika* **58**, 453–467.

J.C. Gower and D. J. Hand (1996). *Biplots*. Chapman & Hall.

See Also

`biplot.princomp`, also for examples.

`biplot.princomp` *Biplot for Principal Components*

Description

Produces a biplot (in the strict sense) from the output of `princomp` or `prcomp`

Usage

```
## S3 method for class 'prcomp':
biplot(x, choices = 1:2, scale = 1, pc.biplot = FALSE, ...)

## S3 method for class 'princomp':
biplot(x, choices = 1:2, scale = 1, pc.biplot = FALSE, ...)
```

Arguments

x	an object of class "princomp".
choices	length 2 vector specifying the components to plot. Only the default is a biplot in the strict sense.
scale	The variables are scaled by lambda ^ scale and the observations are scaled by lambda ^ (1-scale) where lambda are the singular values as computed by princomp. Normally 0 <= scale <= 1, and a warning will be issued if the specified scale is outside this range.
pc.biplot	If true, use what Gabriel (1971) refers to as a "principal component biplot", with lambda = 1 and observations scaled up by sqrt(n) and variables scaled down by sqrt(n). Then inner products between variables approximate covariances and distances between observations approximate Mahalanobis distance.
...	optional arguments to be passed to biplot.default.

Details

This is a method for the generic function biplot. There is considerable confusion over the precise definitions: those of the original paper, Gabriel (1971), are followed here. Gabriel and Odoroff (1990) use the same definitions, but their plots actually correspond to pc.biplot = TRUE.

Side Effects

a plot is produced on the current graphics device.

References

Gabriel, K. R. (1971). The biplot graphical display of matrices with applications to principal component analysis. *Biometrika*, **58**, 453–467.

Gabriel, K. R. and Odoroff, C. L. (1990). Biplots in biomedical research. *Statistics in Medicine*, **9**, 469–485.

See Also

biplot, princomp.

Examples

```
require(graphics)
biplot(princomp(USArrests))
```

`birthday` *Probability of coincidences*

Description

Computes approximate answers to a generalised *birthday paradox* problem. `pbirthday` computes the probability of a coincidence and `qbirthday` computes the number of observations needed to have a specified probability of coincidence.

Usage

```
qbirthday(prob = 0.5, classes = 365, coincident = 2)
pbirthday(n, classes = 365, coincident = 2)
```

Arguments

`classes`	How many distinct categories the people could fall into
`prob`	The desired probability of coincidence
`n`	The number of people
`coincident`	The number of people to fall in the same category

Details

The birthday paradox is that a very small number of people, 23, suffices to have a 50-50 chance that two of them have the same birthday. This function generalises the calculation to probabilities other than 0.5, numbers of coincident events other than 2, and numbers of classes other than 365.

This formula is approximate, as the example below shows. For `coincident=2` the exact computation is straightforward and may be preferable.

Value

`qbirthday`	Number of people needed for a probability `prob` that `k` of them have the same one out of `classes` equiprobable labels.
`pbirthday`	Probability of the specified coincidence

References

Diaconis, P. and Mosteller F. (1989) Methods for studying coincidences. J. American Statistical Association, **84**, 853-861.

Examples

```
require(graphics)

 ## the standard version
qbirthday()
 ## same 4-digit PIN number
qbirthday(classes=10^4)
 ## 0.9 probability of three coincident birthdays
qbirthday(coincident=3, prob=0.9)
## Chance of 4 coincident birthdays in 150 people
```

```
pbirthday(150,coincident=4)
## 100 coincident birthdays in 1000 people: *very* rare:
pbirthday(1000, coincident=100)

## Accuracy compared to exact calculation
x1<-  sapply(10:100, pbirthday)
x2<- 1-sapply(10:100, function(n)prod((365:(365-n+1))/rep(365,n)))
par(mfrow=c(2,2))
plot(x1, x2, xlab="approximate", ylab="exact")
abline(0,1)
plot(x1, x1-x2, xlab="approximate", ylab="error")
abline(h=0)
plot(x1, x2, log="xy", xlab="approximate", ylab="exact")
abline(0,1)
plot(1-x1, 1-x2, log="xy", xlab="approximate", ylab="exact")
abline(0,1)
```

`Box.test` *Box-Pierce and Ljung-Box Tests*

Description

Compute the Box–Pierce or Ljung–Box test statistic for examining the null hypothesis of independence in a given time series. These are sometimes known as 'portmanteau' tests.

Usage

```
Box.test(x, lag = 1, type = c("Box-Pierce", "Ljung-Box"), fitdf = 0)
```

Arguments

`x`	a numeric vector or univariate time series.
`lag`	the statistic will be based on `lag` autocorrelation coefficients.
`type`	test to be performed: partial matching is used.
`fitdf`	number of degrees of freedom to be subtracted if `x` is a series of residuals.

Details

These tests are sometimes applied to the residuals from an `ARMA(p, q)` fit, in which case the references suggest a better approximation to the null-hypothesis distribution is obtained by setting `fitdf = p+q`, provided of course that `lag > fitdf`.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic.
`parameter`	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
`p.value`	the p-value of the test.
`method`	a character string indicating which type of test was performed.
`data.name`	a character string giving the name of the data.

Note

Missing values are not handled.

Author(s)

A. Trapletti

References

Box, G. E. P. and Pierce, D. A. (1970), Distribution of residual correlations in autoregressive-integrated moving average time series models. *Journal of the American Statistical Association*, **65**, 1509–1526.

Ljung, G. M. and Box, G. E. P. (1978), On a measure of lack of fit in time series models. *Biometrika* **65**, 553–564.

Harvey, A. C. (1993) *Time Series Models*. 2nd Edition, Harvester Wheatsheaf, NY, pp. 44, 45.

Examples

```
x <- rnorm (100)
Box.test (x, lag = 1)
Box.test (x, lag = 1, type="Ljung")
```

C *Sets Contrasts for a Factor*

Description

Sets the `"contrasts"` attribute for the factor.

Usage

```
C(object, contr, how.many, ...)
```

Arguments

`object`	a factor or ordered factor
`contr`	which contrasts to use. Can be a matrix with one row for each level of the factor or a suitable function like `contr.poly` or a character string giving the name of the function
`how.many`	the number of contrasts to set, by default one less than `nlevels(object)`.
`...`	additional arguments for the function `contr`.

Details

For compatibility with S, `contr` can be `treatment`, `helmert`, `sum` or `poly` (without quotes) as shorthand for `contr.treatment` and so on.

Value

The factor `object` with the `"contrasts"` attribute set.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`contrasts`, `contr.sum`, etc.

Examples

```
## reset contrasts to defaults
options(contrasts=c("contr.treatment", "contr.poly"))
tens <- with(warpbreaks, C(tension, poly, 1))
attributes(tens)
## tension SHOULD be an ordered factor, but as it is not we can use
aov(breaks ~ wool + tens + tension, data=warpbreaks)

## show the use of ...  The default contrast is contr.treatment here
summary(lm(breaks ~ wool + C(tension, base=2), data=warpbreaks))

# following on from help(esoph)
model3 <- glm(cbind(ncases, ncontrols) ~ agegp + C(tobgp, , 1) +
     C(alcgp, , 1), data = esoph, family = binomial())
summary(model3)
```

---

cancor *Canonical Correlations*

Description

Compute the canonical correlations between two data matrices.

Usage

```
cancor(x, y, xcenter = TRUE, ycenter = TRUE)
```

Arguments

`x`	numeric matrix ($n \times p_1$), containing the x coordinates.
`y`	numeric matrix ($n \times p_2$), containing the y coordinates.
`xcenter`	logical or numeric vector of length p_1, describing any centering to be done on the x values before the analysis. If `TRUE` (default), subtract the column means. If `FALSE`, do not adjust the columns. Otherwise, a vector of values to be subtracted from the columns.
`ycenter`	analogous to `xcenter`, but for the y values.

Details

The canonical correlation analysis seeks linear combinations of the `y` variables which are well explained by linear combinations of the `x` variables. The relationship is symmetric as 'well explained' is measured by correlations.

Value

A list containing the following components:

`cor`	correlations.
`xcoef`	estimated coefficients for the `x` variables.
`ycoef`	estimated coefficients for the `y` variables.
`xcenter`	the values used to adjust the `x` variables.
`ycenter`	the values used to adjust the `x` variables.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Hotelling H. (1936). Relations between two sets of variables. *Biometrika*, **28**, 321–327.

Seber, G. A. F. (1984). *Multivariate Observations*. New York: Wiley, p. 506f.

See Also

`qr`, `svd`.

Examples

```
pop <- LifeCycleSavings[, 2:3]
oec <- LifeCycleSavings[, -(2:3)]
cancor(pop, oec)

x <- matrix(rnorm(150), 50, 3)
y <- matrix(rnorm(250), 50, 5)
(cxy <- cancor(x, y))
all(abs(cor(x %*% cxy$xcoef,
            y %*% cxy$ycoef)[,1:3] - diag(cxy $ cor)) < 1e-15)
all(abs(cor(x %*% cxy$xcoef) - diag(3)) < 1e-15)
all(abs(cor(y %*% cxy$ycoef) - diag(5)) < 1e-15)
```

`case/variable.names`

Case and Variable Names of Fitted Models

Description

Simple utilities returning (non-missing) case names, and (non-eliminated) variable names.

Usage

```
case.names(object, ...)
## S3 method for class 'lm':
case.names(object, full = FALSE, ...)

variable.names(object, ...)
## S3 method for class 'lm':
variable.names(object, full = FALSE, ...)
```

Arguments

`object`	an R object, typically a fitted model.
`full`	logical; if `TRUE`, all names (including zero weights, ...) are returned.
`...`	further arguments passed to or from other methods.

Value

A character vector.

See Also

`lm`; further, `all.names`, `all.vars` for functions with a similar name but only slightly related purpose.

Examples

```
x <- 1:20
y <-  x + (x/4 - 2)^3 + rnorm(20, sd=3)
names(y) <- paste("O",x,sep=".")
ww <- rep(1,20); ww[13] <- 0
summary(lmxy <- lm(y ~ x + I(x^2)+I(x^3) + I((x-10)^2),
                   weights = ww), cor = TRUE)
variable.names(lmxy)
variable.names(lmxy, full= TRUE)# includes the last
case.names(lmxy)
case.names(lmxy, full = TRUE)# includes the 0-weight case
```

`Cauchy` *The Cauchy Distribution*

Description

Density, distribution function, quantile function and random generation for the Cauchy distribution with location parameter `location` and scale parameter `scale`.

Usage

```
dcauchy(x, location = 0, scale = 1, log = FALSE)
pcauchy(q, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
qcauchy(p, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
rcauchy(n, location = 0, scale = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`location, scale`	location and scale parameters.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `location` or `scale` are not specified, they assume the default values of `0` and `1` respectively.

The Cauchy distribution with location l and scale s has density

$$f(x) = \frac{1}{\pi s}\left(1 + \left(\frac{x-l}{s}\right)^2\right)^{-1}$$

for all x.

Value

`dcauchy`, `pcauchy`, and `qcauchy` are respectively the density, distribution function and quantile function of the Cauchy distribution. `rcauchy` generates random deviates from the Cauchy.

Source

`dcauchy`, `pcauchy` and `qcauchy` are all calculated from numerically stable versions of the definitions.

`rcauchy` uses inversion.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 16. Wiley, New York.

See Also

`dt` for the t distribution which generalizes `dcauchy(*, l = 0, s = 1)`.

Examples

```
dcauchy(-1:4)
```

`chisq.test` *Pearson's Chi-squared Test for Count Data*

Description

`chisq.test` performs chi-squared contingency table tests and goodness-of-fit tests.

Usage

```
chisq.test(x, y = NULL, correct = TRUE,
           p = rep(1/length(x), length(x)), rescale.p = FALSE,
           simulate.p.value = FALSE, B = 2000)
```

Arguments

x	a vector or matrix.
y	a vector; ignored if x is a matrix.
correct	a logical indicating whether to apply continuity correction when computing the test statistic for 2x2 tables: one half is subtracted from all $\|O - E\|$ differences. No correction is done if simulate.p.value = TRUE.
p	a vector of probabilities of the same length of x. An error is given if any entry of p is negative.
rescale.p	a logical scalar; if TRUE then p is rescaled (if necessary) to sum to 1. If rescale.p is FALSE, and p does not sum to 1, an error is given.
simulate.p.value	a logical indicating whether to compute p-values by Monte Carlo simulation.
B	an integer specifying the number of replicates used in the Monte Carlo test.

Details

If x is a matrix with one row or column, or if x is a vector and y is not given, then a *goodness-of-fit test* is performed (x is treated as a one-dimensional contingency table). The entries of x must be non-negative integers. In this case, the hypothesis tested is whether the population probabilities equal those in p, or are all equal if p is not given.

If x is a matrix with at least two rows and columns, it is taken as a two-dimensional contingency table. Again, the entries of x must be non-negative integers. Otherwise, x and y must be vectors or factors of the same length; incomplete cases are removed, the objects are coerced into factor objects, and the contingency table is computed from these. Then, Pearson's chi-squared test of the null hypothesis that the joint distribution of the cell counts in a 2-dimensional contingency table is the product of the row and column marginals is performed.

If simulate.p.value is FALSE, the p-value is computed from the asymptotic chi-squared distribution of the test statistic; continuity correction is only used in the 2-by-2 case (if correct is TRUE, the default). Otherwise the p-value is computed for a Monte Carlo test (Hope, 1968) with B replicates.

In the contingency table case simulation is done by random sampling from the set of all contingency tables with given marginals, and works only if the marginals are strictly positive. (A C translation of the algorithm of Patefield (1981) is used.) Continuity correction is never used, and the statistic is quoted without it. Note that this is not the usual sampling situation for the chi-squared test but rather that for Fisher's exact test.

In the goodness-of-fit case simulation is done by random sampling from the discrete distribution specified by p, each sample being of size n = sum(x). This simulation is done in R and may be slow.

Value

A list with class "htest" containing the following components:

statistic	the value the chi-squared test statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic, NA if the p-value is computed by Monte Carlo simulation.
p.value	the p-value for the test.
method	a character string indicating the type of test performed, and whether Monte Carlo simulation or continuity correction was used.

`data.name`	a character string giving the name(s) of the data.
`observed`	the observed counts.
`expected`	the expected counts under the null hypothesis.
`residuals`	the Pearson residuals, `(observed - expected) / sqrt(expected)`.

References

Hope, A. C. A. (1968) A simplified Monte Carlo significance test procedure. *J. Roy, Statist. Soc. B* **30**, 582–598.

Patefield, W. M. (1981) Algorithm AS159. An efficient method of generating r x c tables with given row and column totals. *Applied Statistics* **30**, 91–97.

Examples

```
## Not really a good example
chisq.test(InsectSprays$count > 7, InsectSprays$spray)
                                # Prints test summary
chisq.test(InsectSprays$count > 7, InsectSprays$spray)$observed
                                # Counts observed
chisq.test(InsectSprays$count > 7, InsectSprays$spray)$expected
                                # Counts expected under the null

## Effect of simulating p-values
x <- matrix(c(12, 5, 7, 7), ncol = 2)
chisq.test(x)$p.value           # 0.4233
chisq.test(x, simulate.p.value = TRUE, B = 10000)$p.value
                                # around 0.29!

## Testing for population probabilities
## Case A. Tabulated data
x <- c(A = 20, B = 15, C = 25)
chisq.test(x)
chisq.test(as.table(x))         # the same
x <- c(89,37,30,28,2)
p <- c(40,20,20,15,5)
try(
chisq.test(x, p = p)            # gives an error
)
chisq.test(x, p = p, rescale.p = TRUE)
                                # works
p <- c(0.40,0.20,0.20,0.19,0.01)
                                # Expected count in category 5
                                # is 1.86 < 5 ==> chi square approx.
chisq.test(x, p = p)            #               maybe doubtful, but is ok!
chisq.test(x, p = p,simulate.p.value = TRUE)

## Case B. Raw data
x <- trunc(5 * runif(100))
chisq.test(table(x))            # NOT 'chisq.test(x)'!
```

`Chisquare` *The (non-central) Chi-Squared Distribution*

Description

Density, distribution function, quantile function and random generation for the chi-squared (χ^2) distribution with `df` degrees of freedom and optional non-centrality parameter `ncp`.

Usage

```
dchisq(x, df, ncp=0, log = FALSE)
pchisq(q, df, ncp=0, lower.tail = TRUE, log.p = FALSE)
qchisq(p, df, ncp=0, lower.tail = TRUE, log.p = FALSE)
rchisq(n, df, ncp=0)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`df`	degrees of freedom (non-negative, but can be non-integer).
`ncp`	non-centrality parameter (non-negative).
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The chi-squared distribution with `df`$= n \geq 0$ degrees of freedom has density

$$f_n(x) = \frac{1}{2^{n/2}\Gamma(n/2)} x^{n/2-1} e^{-x/2}$$

for $x > 0$. The mean and variance are n and $2n$.

The non-central chi-squared distribution with `df`$= n$ degrees of freedom and non-centrality parameter `ncp` $= \lambda$ has density

$$f(x) = e^{-\lambda/2} \sum_{r=0}^{\infty} \frac{(\lambda/2)^r}{r!} f_{n+2r}(x)$$

for $x \geq 0$. For integer n, this is the distribution of the sum of squares of n normals each with variance one, λ being the sum of squares of the normal means; further,
$E(X) = n + \lambda$, $Var(X) = 2(n + 2 * \lambda)$, and $E((X - E(X))^3) = 8(n + 3 * \lambda)$.

Note that the degrees of freedom `df`$= n$, can be non-integer, and also $n = 0$ which is relevant for non-centrality $\lambda > 0$, see Johnson et al. (1995, chapter 29).

Note that `ncp` values larger than about 1e5 may give inaccurate results with many warnings for `pchisq` and `qchisq`.

Value

`dchisq` gives the density, `pchisq` gives the distribution function, `qchisq` gives the quantile function, and `rchisq` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Source

The central cases are computed via the gamma distribution.

The non-central `dchisq` and `rchisq` are computed as a Poisson mixture central of chi-squares (Johnson et al, 1995, p.436).

The non-central `pchisq` is for `ncp < 80` computed from the Poisson mixture of central chi-squares and for larger `ncp` based on a C translation of

Ding, C. G. (1992) Algorithm AS275: Computing the non-central chi-squared distribution function. *Appl.Statist.*, **41** 478–482.

which computes the lower tail only (so the upper tail suffers from cancellation).

The non-central `qchisq` is based on inversion of `pchisq`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, chapters 18 (volume 1) and 29 (volume 2). Wiley, New York.

See Also

A central chi-squared distribution with n degrees of freedom is the same as a Gamma distribution with `shape` $\alpha = n/2$ and `scale` $\sigma = 2$. Hence, see `dgamma` for the Gamma distribution.

Examples

```
require(graphics)

dchisq(1, df=1:3)
pchisq(1, df= 3)
pchisq(1, df= 3, ncp = 0:4)# includes the above

x <- 1:10
## Chi-squared(df = 2) is a special exponential distribution
all.equal(dchisq(x, df=2), dexp(x, 1/2))
all.equal(pchisq(x, df=2), pexp(x, 1/2))

## non-central RNG -- df=0 with ncp > 0:  Z0 has point mass at 0!
Z0 <- rchisq(100, df = 0, ncp = 2.)
graphics::stem(Z0)

## Not run:
## visual testing
## do P-P plots for 1000 points at various degrees of freedom
L <- 1.2; n <- 1000; pp <- ppoints(n)
op <- par(mfrow = c(3,3), mar= c(3,3,1,1)+.1, mgp= c(1.5,.6,0),
          oma = c(0,0,3,0))
```

```
for(df in 2^(4*rnorm(9))) {
  plot(pp, sort(pchisq(rr <- rchisq(n,df=df, ncp=L), df=df, ncp=L)),
       ylab="pchisq(rchisq(.),.)", pch=".")
  mtext(paste("df = ",formatC(df, digits = 4)), line= -2, adj=0.05)
  abline(0,1,col=2)
}
mtext(expression("P-P plots : Noncentral  "*
                 chi^2 *"(n=1000, df=X, ncp= 1.2)"),
      cex = 1.5, font = 2, outer=TRUE)
par(op)
## End(Not run)

## "analytical" test
lam <- seq(0,100, by=.25)
p00 <- pchisq(0,      df=0, ncp=lam)
p.0 <- pchisq(1e-300, df=0, ncp=lam)
stopifnot(all.equal(p00, exp(-lam/2)),
          all.equal(p.0, exp(-lam/2)))
```

`clearNames` *Remove the Names from an Object*

Description

This function sets the `names` attribute of `object` to `NULL` and returns the object.

Usage

```
clearNames(object)
```

Arguments

`object`	an object that may have a `names` attribute

Value

An object similar to `object` but without names.

Author(s)

Douglas Bates and Saikat DebRoy

See Also

`setNames`

Examples

```
lapply( women, mean )               # has a names attribute
clearNames( lapply( women, mean ) ) # removes the names
```

`cmdscale` *Classical (Metric) Multidimensional Scaling*

Description

Classical multidimensional scaling of a data matrix. Also known as *principal coordinates analysis* (Gower, 1966).

Usage

```
cmdscale(d, k = 2, eig = FALSE, add = FALSE, x.ret = FALSE)
```

Arguments

`d`	a distance structure such as that returned by `dist` or a full symmetric matrix containing the dissimilarities.
`k`	the dimension of the space which the data are to be represented in; must be in $\{1, 2, \ldots, n-1\}$.
`eig`	indicates whether eigenvalues should be returned.
`add`	logical indicating if an additive constant $c*$ should be computed, and added to the non-diagonal dissimilarities such that all $n-1$ eigenvalues are non-negative.
`x.ret`	indicates whether the doubly centred symmetric distance matrix should be returned.

Details

Multidimensional scaling takes a set of dissimilarities and returns a set of points such that the distances between the points are approximately equal to the dissimilarities.

The functions `isoMDS` and `sammon` in package **MASS** provide alternative ordination techniques.

When `add = TRUE`, an additive constant $c*$ is computed, and the dissimilarities $d_{ij} + c*$ are used instead of the original d_{ij}'s.

Whereas S (Becker *et al.*, 1988) computes this constant using an approximation suggested by Torgerson, R uses the analytical solution of Cailliez (1983), see also Cox and Cox (1994).

Value

If `eig = FALSE` and `x.ret = FALSE` (default), a matrix with `k` columns whose rows give the coordinates of the points chosen to represent the dissimilarities.

Otherwise, a list containing the following components.

`points`	a matrix with `k` columns whose rows give the coordinates of the points chosen to represent the dissimilarities.
`eig`	the $n-1$ eigenvalues computed during the scaling process if `eig` is true.
`x`	the doubly centered distance matrix if `x.ret` is true.
`GOF`	a numeric vector of length 2, equal to say (g_1, g_2), where $g_i = (\sum_{j=1}^{k} \lambda_j)/(\sum_{j=1}^{n} T_i(\lambda_j))$, where λ_j are the eigenvalues (sorted decreasingly), $T_1(v) = \lvert v \rvert$, and $T_2(v) = max(v, 0)$.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cailliez, F. (1983) The analytical solution of the additive constant problem. *Psychometrika* **48**, 343–349.

Cox, T. F. and Cox, M. A. A. (1994) *Multidimensional Scaling*. Chapman and Hall.

Gower, J. C. (1966) Some distance properties of latent root and vector methods used in multivariate analysis. *Biometrika* **53**, 325–328.

Mardia, K. V., Kent, J. T. and Bibby, J. M. (1979). Chapter 14 of *Multivariate Analysis*, London: Academic Press.

Seber, G. A. F. (1984). *Multivariate Observations*. New York: Wiley.

Torgerson, W. S. (1958). *Theory and Methods of Scaling*. New York: Wiley.

See Also

`dist`. Also `isoMDS` and `sammon` in package **MASS**.

Examples

```
require(graphics)

loc <- cmdscale(eurodist)
x <- loc[,1]
y <- -loc[,2]
plot(x, y, type="n", xlab="", ylab="", main="cmdscale(eurodist)")
text(x, y, rownames(loc), cex=0.8)

cmdsE <- cmdscale(eurodist, k=20, add = TRUE, eig = TRUE, x.ret = TRUE)
utils::str(cmdsE)
```

---

coef *Extract Model Coefficients*

Description

`coef` is a generic function which extracts model coefficients from objects returned by modeling functions. `coefficients` is an *alias* for it.

Usage

```
coef(object, ...)
coefficients(object, ...)
```

Arguments

`object`	an object for which the extraction of model coefficients is meaningful.
`...`	other arguments.

Details

All object classes which are returned by model fitting functions should provide a `coef` method or use the default one. (Note that the method is for `coef` and not `coefficients`.)

Class `"aov"` has a `coef` method that does not report aliased coefficients (see `alias`).

Value

Coefficients extracted from the model object `object`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`fitted.values` and `residuals` for related methods; `glm`, `lm` for model fitting.

Examples

```
x <- 1:5; coef(lm(c(1:3,7,6) ~ x))
```

`complete.cases` *Find Complete Cases*

Description

Return a logical vector indicating which cases are complete, i.e., have no missing values.

Usage

```
complete.cases(...)
```

Arguments

`...`	a sequence of vectors, matrices and data frames.

Value

A logical vector specifying which observations/rows have no missing values across the entire sequence.

See Also

`is.na`, `na.omit`, `na.fail`.

Examples

```
x <- airquality[, -1] # x is a regression design matrix
y <- airquality[,  1] # y is the corresponding response

stopifnot(complete.cases(y) != is.na(y))
ok <- complete.cases(x,y)
sum(!ok) # how many are not "ok" ?
x <- x[ok,]
y <- y[ok]
```

---

confint *Confidence Intervals for Model Parameters*

Description

Computes confidence intervals for one or more parameters in a fitted model. There is a default and a method for objects inheriting from class "lm".

Usage

```
confint(object, parm, level = 0.95, ...)
```

Arguments

object	a fitted model object.
parm	a specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered.
level	the confidence level required.
...	additional argument(s) for methods.

Details

confint is a generic function. The default method assumes asymptotic normality, and needs suitable coef and vcov methods to be available. The default method can be called directly for comparison with other methods.

For objects of class "lm" the direct formulae based on t values are used.

There are stub methods for classes "glm" and "nls" which invoke those in package **MASS** which are based on profile likelihoods.

Value

A matrix (or vector) with columns giving lower and upper confidence limits for each parameter. These will be labelled as (1-level)/2 and 1 - (1-level)/2 in % (by default 2.5% and 97.5%).

See Also

confint.glm and confint.nls in package **MASS**.

Examples

```
fit <- lm(100/mpg ~ disp + hp + wt + am, data=mtcars)
confint(fit)
confint(fit, "wt")

## from example(glm) (needs MASS to be present on the system)
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9); treatment <- gl(3,3)
glm.D93 <- glm(counts ~ outcome + treatment, family=poisson())
confint(glm.D93)
confint.default(glm.D93)  # based on asymptotic normality
```

---

constrOptim *Linearly constrained optimisation*

Description

Minimise a function subject to linear inequality constraints using an adaptive barrier algorithm.

Usage

```
constrOptim(theta, f, grad, ui, ci, mu = 1e-04, control = list(),
            method = if(is.null(grad)) "Nelder-Mead" else "BFGS",
            outer.iterations = 100, outer.eps = 1e-05, ...)
```

Arguments

`theta`	Starting value: must be in the feasible region.
`f`	Function to minimise (see below).
`grad`	Gradient of `f`, or `NULL` (see below).
`ui`	Constraints (see below).
`ci`	Constraints (see below).
`mu`	(Small) tuning parameter.
`control`	Passed to `optim`.
`method`	Passed to `optim`.
`outer.iterations`	Iterations of the barrier algorithm.
`outer.eps`	Criterion for relative convergence of the barrier algorithm.
`...`	Other arguments passed to `optim`, which will pass them to `f` and `grad` if it does not use them.

Details

The feasible region is defined by `ui %*% theta - ci >= 0`. The starting value must be in the interior of the feasible region, but the minimum may be on the boundary.

A logarithmic barrier is added to enforce the constraints and then `optim` is called. The barrier function is chosen so that the objective function should decrease at each outer iteration. Minima in the interior of the feasible region are typically found quite quickly, but a substantial number of outer iterations may be needed for a minimum on the boundary.

The tuning parameter mu multiplies the barrier term. Its precise value is often relatively unimportant. As mu increases the augmented objective function becomes closer to the original objective function but also less smooth near the boundary of the feasible region.

Any optim method that permits infinite values for the objective function may be used (currently all but "L-BFGS-B").

The objective function f takes as first argument the vector of parameters over which minimisation is to take place. It should return a scalar result. Optional arguments ... will be passed to optim and then (if not used by optim) to f. As with optim, the default is to minimise, but maximisation can be performed by setting control$fnscale to a negative value.

The gradient function grad must be supplied except with method="Nelder-Mead". It should take arguments matching those of f and return a vector containing the gradient.

Value

As for optim, but with two extra components: barrier.value giving the value of the barrier function at the optimum and outer.iterations gives the number of outer iterations (calls to optim)

References

K. Lange *Numerical Analysis for Statisticians.* Springer 2001, p185ff

See Also

optim, especially method="L-BFGS-B" which does box-constrained optimisation.

Examples

```
## from optim
fr <- function(x) {   ## Rosenbrock Banana function
    x1 <- x[1]
    x2 <- x[2]
    100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
grr <- function(x) { ## Gradient of 'fr'
    x1 <- x[1]
    x2 <- x[2]
    c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
       200 *      (x2 - x1 * x1))
}

optim(c(-1.2,1), fr, grr)
#Box-constraint, optimum on the boundary
constrOptim(c(-1.2,0.9), fr, grr, ui=rbind(c(-1,0),c(0,-1)), ci=c(-1,-1))
#  x<=0.9,  y-x>0.1
constrOptim(c(.5,0), fr, grr, ui=rbind(c(-1,0),c(1,-1)), ci=c(-0.9,0.1))

## Solves linear and quadratic programming problems
## but needs a feasible starting value
#
# from example(solve.QP) in 'quadprog'
# no derivative
fQP <- function(b) {-sum(c(0,5,0)*b)+0.5*sum(b*b)}
Amat       <- matrix(c(-4,-3,0,2,1,0,0,-2,1),3,3)
bvec       <- c(-8,2,0)
```

```
constrOptim(c(2,-1,-1), fQP, NULL, ui=t(Amat),ci=bvec)
# derivative
gQP <- function(b) {-c(0,5,0)+b}
constrOptim(c(2,-1,-1), fQP, gQP, ui=t(Amat), ci=bvec)

## Now with maximisation instead of minimisation
hQP <- function(b) {sum(c(0,5,0)*b)-0.5*sum(b*b)}
constrOptim(c(2,-1,-1), hQP, NULL, ui=t(Amat), ci=bvec,
            control=list(fnscale=-1))
```

`contrast` *Contrast Matrices*

Description

Return a matrix of contrasts.

Usage

```
contr.helmert(n, contrasts = TRUE)
contr.poly(n, scores = 1:n, contrasts = TRUE)
contr.sum(n, contrasts = TRUE)
contr.treatment(n, base = 1, contrasts = TRUE)
contr.SAS(n, contrasts = TRUE)
```

Arguments

`n`	a vector of levels for a factor, or the number of levels.
`contrasts`	a logical indicating whether contrasts should be computed.
`scores`	the set of values over which orthogonal polynomials are to be computed.
`base`	an integer specifying which group is considered the baseline group. Ignored if `contrasts` is `FALSE`.

Details

These functions are used for creating contrast matrices for use in fitting analysis of variance and regression models. The columns of the resulting matrices contain contrasts which can be used for coding a factor with `n` levels. The returned value contains the computed contrasts. If the argument `contrasts` is `FALSE` a square indicator matrix (the dummy coding) is returned **except** for `contr.poly` (which include the 0-degree, i.e. constant, polynomial when `contrasts = FALSE`).

`cont.helmert` returns Helmert contrasts, which contrast the second level with the first, the third with the average of the first two, and so on. `contr.poly` returns contrasts based on orthogonal polynomials. `contr.sum` uses 'sum to zero contrasts'.

`contr.treatment` contrasts each level with the baseline level (specified by `base`): the baseline level is omitted. Note that this does not produce 'contrasts' as defined in the standard theory for linear models as they are not orthogonal to the intercept.

`contr.SAS` is a wrapper for `contr.treatment` that sets the base level to be the last level of the factor. The coefficients produced when using these contrasts should be equivalent to those produced by many (but not all) SAS procedures.

Value

A matrix with `n` rows and `k` columns, with `k=n-1` if `contrasts` is `TRUE` and `k=n` if `contrasts` is `FALSE`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`contrasts`, `C`, and `aov`, `glm`, `lm`.

Examples

```
(cH <- contr.helmert(4))
apply(cH, 2,sum) # column sums are 0!
crossprod(cH) # diagonal -- columns are orthogonal
contr.helmert(4, contrasts = FALSE) # just the 4 x 4 identity matrix

(cT <- contr.treatment(5))
all(crossprod(cT) == diag(4)) # TRUE: even orthonormal

(cT. <- contr.SAS(5))
all(crossprod(cT.) == diag(4))# TRUE

(cP <- contr.poly(3)) # Linear and Quadratic
zapsmall(crossprod(cP), digits=15) # orthonormal up to fuzz
```

`contrasts` *Get and Set Contrast Matrices*

Description

Set and view the contrasts associated with a factor.

Usage

```
contrasts(x, contrasts = TRUE)
contrasts(x, how.many) <- value
```

Arguments

`x`	a factor or a logical variable.
`contrasts`	logical. See 'Details'.
`how.many`	How many contrasts should be made. Defaults to one less than the number of levels of `x`. This need not be the same as the number of columns of `value`.
`value`	either a numeric matrix whose columns give coefficients for contrasts in the levels of `x`, or the (quoted) name of a function which computes such matrices.

Details

If contrasts are not set for a factor the default functions from `options("contrasts")` are used.

A logical vector `x` is converted into a two-level factor with levels `c(FALSE, TRUE)` (regardless of which levels occur in the variable).

The argument `contrasts` is ignored if `x` has a matrix `contrasts` attribute set. Otherwise if `contrasts = TRUE` it is passed to a contrasts function such as `contr.treatment` and if `contrasts = FALSE` an identity matrix is returned.

If `value` supplies more than `how.many` contrasts, the first `how.many` are used. If too few are supplied, a suitable contrast matrix is created by extending `value` after ensuring its columns are contrasts (orthogonal to the constant term) and not collinear.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`C`, `contr.helmert`, `contr.poly`, `contr.sum`, `contr.treatment`; `glm`, `aov`, `lm`.

Examples

```
utils::example(factor)
fff <- ff[, drop=TRUE]  # reduce to 5 levels.
contrasts(fff) # treatment contrasts by default
contrasts(C(fff, sum))
contrasts(fff, contrasts = FALSE) # the 5x5 identity matrix

contrasts(fff) <- contr.sum(5); contrasts(fff)  # set sum contrasts
contrasts(fff, 2) <- contr.sum(5); contrasts(fff)  # set 2 contrasts
# supply 2 contrasts, compute 2 more to make full set of 4.
contrasts(fff) <- contr.sum(5)[,1:2]; contrasts(fff)
```

`convolve` *Fast Convolution*

Description

Use the Fast Fourier Transform to compute the several kinds of convolutions of two sequences.

Usage

```
convolve(x, y, conj = TRUE, type = c("circular", "open", "filter"))
```

Arguments

`x,y`	numeric sequences *of the same length* to be convolved.
`conj`	logical; if `TRUE`, take the complex *conjugate* before back-transforming (default, and used for usual convolution).

`type` character; one of `"circular"`, `"open"`, `"filter"` (beginning of word is ok). For `circular`, the two sequences are treated as *circular*, i.e., periodic. For `open` and `filter`, the sequences are padded with `0`s (from left and right) first; `"filter"` returns the middle sub-vector of `"open"`, namely, the result of running a weighted mean of `x` with weights `y`.

Details

The Fast Fourier Transform, `fft`, is used for efficiency.

The input sequences `x` and `y` must have the same length if `circular` is true.

Note that the usual definition of convolution of two sequences `x` and `y` is given by `convolve(x, rev(y), type = "o")`.

Value

If `r <- convolve(x,y, type = "open")` and `n <- length(x)`, `m <- length(y)`, then

$$r_k = \sum_i x_{k-m+i} y_i$$

where the sum is over all valid indices i, for $k = 1, \ldots, n + m - 1$

If `type == "circular"`, $n = m$ is required, and the above is true for $i, k = 1, \ldots, n$ when $x_j := x_{n+j}$ for $j < 1$.

References

Brillinger, D. R. (1981) *Time Series: Data Analysis and Theory*, Second Edition. San Francisco: Holden-Day.

See Also

`fft`, `nextn`, and particularly `filter` (from the **stats** package) which may be more appropriate.

Examples

```
require(graphics)

x <- c(0,0,0,100,0,0,0)
y <- c(0,0,1, 2 ,1,0,0)/4
zapsmall(convolve(x,y))         #  *NOT* what you first thought.
zapsmall(convolve(x, y[3:5], type="f")) # rather
x <- rnorm(50)
y <- rnorm(50)
# Circular convolution *has* this symmetry:
all.equal(convolve(x,y, conj = FALSE), rev(convolve(rev(y),x)))

n <- length(x <- -20:24)
y <- (x-10)^2/1000 + rnorm(x)/8

Han <- function(y) # Hanning
       convolve(y, c(1,2,1)/4, type = "filter")

plot(x,y, main="Using  convolve(.) for Hanning filters")
lines(x[-c(1  , n)      ], Han(y), col="red")
lines(x[-c(1:2, (n-1):n)], Han(Han(y)), lwd=2, col="dark blue")
```

`cophenetic` *Cophenetic Distances for a Hierarchical Clustering*

Description

Computes the cophenetic distances for a hierarchical clustering.

Usage

```
cophenetic(x)
## Default S3 method:
cophenetic(x)
## S3 method for class 'dendrogram':
cophenetic(x)
```

Arguments

`x`	an R object representing a hierarchical clustering. For the default method, an object of class `hclust` or with a method for `as.hclust()` such as `agnes`.

Details

The cophenetic distance between two observations that have been clustered is defined to be the intergroup dissimilarity at which the two observations are first combined into a single cluster. Note that this distance has many ties and restrictions.

It can be argued that a dendrogram is an appropriate summary of some data if the correlation between the original distances and the cophenetic distances is high. Otherwise, it should simply be viewed as the description of the output of the clustering algorithm.

`cophenetic` is a generic function. Support for classes which represent hierarchical clusterings (total indexed hierarchies) can be added by providing an `as.hclust()` or, more directly, a `cophenetic()` method for such a class.

The method for objects of class `"dendrogram"` requires that all leaves of the dendrogram object have non-null labels.

Value

An object of class `dist`.

Author(s)

Robert Gentleman

References

Sneath, P.H.A. and Sokal, R.R. (1973) *Numerical Taxonomy: The Principles and Practice of Numerical Classification*, p. 278 ff; Freeman, San Francisco.

See Also

`dist`, `hclust`

Examples

```
require(graphics)

d1 <- dist(USArrests)
hc <- hclust(d1, "ave")
d2 <- cophenetic(hc)
cor(d1,d2) # 0.7659

## Example from Sneath & Sokal, Fig. 5-29, p.279
d0 <- c(1,3.8,4.4,5.1, 4,4.2,5, 2.6,5.3, 5.4)
attributes(d0) <- list(Size = 5, diag=TRUE)
class(d0) <- "dist"
names(d0) <- letters[1:5]
d0
utils::str(upgma <- hclust(d0, method = "average"))
plot(upgma, hang = -1)
#
(d.coph <- cophenetic(upgma))
cor(d0, d.coph) # 0.9911
```

---

cor *Correlation, Variance and Covariance (Matrices)*

Description

`var`, `cov` and `cor` compute the variance of `x` and the covariance or correlation of `x` and `y` if these are vectors. If `x` and `y` are matrices then the covariances (or correlations) between the columns of `x` and the columns of `y` are computed.

`cov2cor` scales a covariance matrix into the corresponding correlation matrix *efficiently*.

Usage

```
var(x, y = NULL, na.rm = FALSE, use)

cov(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))

cor(x, y = NULL, use = "everything",
     method = c("pearson", "kendall", "spearman"))

cov2cor(V)
```

Arguments

`x`	a numeric vector, matrix or data frame.
`y`	`NULL` (default) or a vector, matrix or data frame with compatible dimensions to `x`. The default is equivalent to `y = x` (but more efficient).
`na.rm`	logical. Should missing values be removed?
`use`	an optional character string giving a method for computing covariances in the presence of missing values. This must be (an abbreviation of) one of the strings `"everything"`, `"all.obs"`, `"complete.obs"`, `"na.or.complete"`, or `"pairwise.complete.obs"`.

`method`	a character string indicating which correlation coefficient (or covariance) is to be computed. One of `"pearson"` (default), `"kendall"`, or `"spearman"`, can be abbreviated.
`V`	symmetric numeric matrix, usually positive definite such as a covariance matrix.

Details

For `cov` and `cor` one must *either* give a matrix or data frame for `x` *or* give both `x` and `y`.

`var` is just another interface to `cov`, where `na.rm` is used to determine the default for `use` when that is unspecified. If `na.rm` is `TRUE` then the complete observations (rows) are used (`use = "na.or.complete"`) to compute the variance. Otherwise, by default `use = "everything"`.

If `use` is `"everything"`, `NA`s will propagate conceptually, i.e., a resulting value will be `NA` whenever one of its contributing observations is `NA`.
If `use` is `"all.obs"`, then the presence of missing observations will produce an error. If `use` is `"complete.obs"` then missing values are handled by casewise deletion (and if there are no complete cases, that gives an error).
`"na.or.complete"` is the same unless there are no complete cases, that gives `NA`. Finally, if `use` has the value `"pairwise.complete.obs"` then the correlation or covariance between each pair of variables is computed using all complete pairs of observations on those variables. This can result in covariance or correlation matrices which are not positive semi-definite, as well as `NA` entries if there are no complete pairs for that pair of variables. For `cov` and `var`, `"pairwise.complete.obs"` only works with the `"pearson"` method. Note that (the equivalent of) `var(double(0), use=*)` gives `NA` for `use = "everything"` and `"na.or.complete"`, and gives an error in the other cases.

The denominator $n - 1$ is used which gives an unbiased estimator of the (co)variance for i.i.d. observations. These functions return `NA` when there is only one observation (whereas S-PLUS has been returning `NaN`), and fail if `x` has length zero.

For `cor()`, if `method` is `"kendall"` or `"spearman"`, Kendall's τ or Spearman's ρ statistic is used to estimate a rank-based measure of association. These are more robust and have been recommended if the data do not necessarily come from a bivariate normal distribution.
For `cov()`, a non-Pearson method is unusual but available for the sake of completeness. Note that `"spearman"` basically computes `cor(R(x), R(y))` (or `cov(.,.)`) where `R(u) := rank(u, na.last="keep")`. In the case of missing values, the ranks are calculated depending on the value of `use`, either based on complete observations, or based on pairwise completeness with reranking for each pair.

Scaling a covariance matrix into a correlation one can be achieved in many ways, mathematically most appealing by multiplication with a diagonal matrix from left and right, or more efficiently by using `sweep(.., FUN = "/")` twice. The `cov2cor` function is even a bit more efficient, and provided mostly for didactical reasons.

Value

For `r <- cor(*, use = "all.obs")`, it is now guaranteed that `all(r <= 1)`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

cor.test for confidence intervals (and tests).

cov.wt for *weighted* covariance computation.

sd for standard deviation (vectors).

Examples

```
var(1:10)# 9.166667

var(1:5,1:5)# 2.5

## Two simple vectors
cor(1:10,2:11)# == 1

## Correlation Matrix of Multivariate sample:
(Cl <- cor(longley))
## Graphical Correlation Matrix:
symnum(Cl) # highly correlated

## Spearman's rho  and  Kendall's tau
symnum(clS <- cor(longley, method = "spearman"))
symnum(clK <- cor(longley, method = "kendall"))
## How much do they differ?
i <- lower.tri(Cl)
cor(cbind(P = Cl[i], S = clS[i], K = clK[i]))

## cov2cor() scales a covariance matrix by its diagonal
##           to become the correlation matrix.
cov2cor # see the function definition {and learn ..}
stopifnot(all.equal(Cl, cov2cor(cov(longley))),
          all.equal(cor(longley, method="kendall"),
            cov2cor(cov(longley, method="kendall"))))

##--- Missing value treatment:
C1 <- cov(swiss)
range(eigen(C1, only.values=TRUE)$values) # 6.19        1921
swM <- swiss
swM[1,2] <- swM[7,3] <- swM[25,5] <- NA # create 3 "missing"
try(cov(swM)) # Error: missing obs...
C2 <- cov(swM, use = "complete")
range(eigen(C2, only.values=TRUE)$values) # 6.46        1930
C3 <- cov(swM, use = "pairwise")
range(eigen(C3, only.values=TRUE)$values) # 6.19        1938

symnum(cor(swM, method = "kendall", use = "complete"))
## Kendall's tau doesn't change much:
symnum(cor(swiss, method = "kendall"))
```

---

cor.test *Test for Association/Correlation Between Paired Samples*

Description

Test for association between paired samples, using one of Pearson's product moment correlation coefficient, Kendall's τ or Spearman's ρ.

Usage

```
cor.test(x, ...)

## Default S3 method:
cor.test(x, y,
         alternative = c("two.sided", "less", "greater"),
         method = c("pearson", "kendall", "spearman"),
         exact = NULL, conf.level = 0.95, ...)

## S3 method for class 'formula':
cor.test(formula, data, subset, na.action, ...)
```

Arguments

`x, y`	numeric vectors of data values. `x` and `y` must have the same length.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter. `"greater"` corresponds to positive association, `"less"` to negative association.
`method`	a character string indicating which correlation coefficient is to be used for the test. One of `"pearson"`, `"kendall"`, or `"spearman"`, can be abbreviated.
`exact`	a logical indicating whether an exact p-value should be computed. Used for Kendall's τ and Spearman's ρ. See 'Details' for the meaning of `NULL` (the default).
`conf.level`	confidence level for the returned confidence interval. Currently only used for the Pearson product moment correlation coefficient if there are at least 4 complete pairs of observations.
`formula`	a formula of the form `~ u + v`, where each of `u` and `v` are numeric variables giving the data values for one sample. The samples must be of the same length.
`data`	an optional matrix or data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`subset`	an optional vector specifying a subset of observations to be used.
`na.action`	a function which indicates what should happen when the data contain `NA`s. Defaults to `getOption("na.action")`.
`...`	further arguments to be passed to or from methods.

Details

The three methods each estimate the association between paired samples and compute a test of the value being zero. They use different measures of association, all in the range $[-1, 1]$ with 0 indicating no association. These are sometimes referred to as tests of no *correlation*, but that term is often confined to the default method.

If `method` is `"pearson"`, the test statistic is based on Pearson's product moment correlation coefficient `cor(x, y)` and follows a t distribution with `length(x)-2` degrees of freedom if the

samples follow independent normal distributions. If there are at least 4 complete pairs of observation, an asymptotic confidence interval is given based on Fisher's Z transform.

If `method` is `"kendall"` or `"spearman"`, Kendall's τ or Spearman's ρ statistic is used to estimate a rank-based measure of association. These tests may be used if the data do not necessarily come from a bivariate normal distribution.

For Kendall's test, by default (if `exact` is NULL), an exact p-value is computed if there are less than 50 paired samples containing finite values and there are no ties. Otherwise, the test statistic is the estimate scaled to zero mean and unit variance, and is approximately normally distributed.

For Spearman's test, p-values are computed using algorithm AS 89.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic.
`parameter`	the degrees of freedom of the test statistic in the case that it follows a t distribution.
`p.value`	the p-value of the test.
`estimate`	the estimated measure of association, with name `"cor"`, `"tau"`, or `"rho"` corresponding to the method employed.
`null.value`	the value of the association measure under the null hypothesis, always `0`.
`alternative`	a character string describing the alternative hypothesis.
`method`	a character string indicating how the association was measured.
`data.name`	a character string giving the names of the data.
`conf.int`	a confidence interval for the measure of association. Currently only given for Pearson's product moment correlation coefficient in case of at least 4 complete pairs of observations.

References

D. J. Best & D. E. Roberts (1975), Algorithm AS 89: The Upper Tail Probabilities of Spearman's ρ. *Applied Statistics*, **24**, 377–379.

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric Statistical Methods.* New York: John Wiley & Sons. Pages 185–194 (Kendall and Spearman tests).

Examples

```
## Hollander & Wolfe (1973), p. 187f.
## Assessment of tuna quality.  We compare the Hunter L measure of
##  lightness to the averages of consumer panel scores (recoded as
##  integer values from 1 to 6 and averaged over 80 such values) in
##  9 lots of canned tuna.

x <- c(44.4, 45.9, 41.9, 53.3, 44.7, 44.1, 50.7, 45.2, 60.1)
y <- c( 2.6,  3.1,  2.5,  5.0,  3.6,  4.0,  5.2,  2.8,  3.8)

##  The alternative hypothesis of interest is that the
##  Hunter L value is positively associated with the panel score.

cor.test(x, y, method = "kendall", alternative = "greater")
## => p=0.05972
```

```
cor.test(x, y, method = "kendall", alternative = "greater",
         exact = FALSE) # using large sample approximation
## => p=0.04765

## Compare this to
cor.test(x, y, method = "spearm", alternative = "g")
cor.test(x, y,                    alternative = "g")

## Formula interface.
require(graphics)
pairs(USJudgeRatings)
cor.test(~ CONT + INTG, data = USJudgeRatings)
```

`cov.wt` *Weighted Covariance Matrices*

Description

Returns a list containing estimates of the weighted covariance matrix and the mean of the data, and optionally of the (weighted) correlation matrix.

Usage

```
cov.wt(x, wt = rep(1/nrow(x), nrow(x)), cor = FALSE, center = TRUE,
       method = c("unbiased", "ML"))
```

Arguments

`x`	a matrix or data frame. As usual, rows are observations and columns are variables.
`wt`	a non-negative and non-zero vector of weights for each observation. Its length must equal the number of rows of `x`.
`cor`	a logical indicating whether the estimated correlation weighted matrix will be returned as well.
`center`	either a logical or a numeric vector specifying the centers to be used when computing covariances. If `TRUE`, the (weighted) mean of each variable is used, if `FALSE`, zero is used. If `center` is numeric, its length must equal the number of columns of `x`.
`method`	string specifying how the result is scaled, see 'Details' below.

Details

By default, `method = "unbiased"`, The covariance matrix is divided by one minus the sum of squares of the weights, so if the weights are the default $(1/n)$ the conventional unbiased estimate of the covariance matrix with divisor $(n - 1)$ is obtained. This differs from the behaviour in S-PLUS which corresponds to `method = "ML"` and does not divide.

Value

A list containing the following named components:

cov	the estimated (weighted) covariance matrix
center	an estimate for the center (mean) of the data.
n.obs	the number of observations (rows) in x.
wt	the weights used in the estimation. Only returned if given as an argument.
cor	the estimated correlation matrix. Only returned if cor is TRUE.

See Also

cov and var.

Examples

```
(xy <- cbind(x = 1:10, y = c(1:3, 8:5, 8:10)))
w1 <- c(0,0,0,1,1,1,1,1,0,0)
cov.wt(xy, wt = w1) # i.e. method = "unbiased"
cov.wt(xy, wt = w1, method = "ML", cor = TRUE)
```

---

cpgram *Plot Cumulative Periodogram*

Description

Plots a cumulative periodogram.

Usage

```
cpgram(ts, taper = 0.1,
       main = paste("Series: ", deparse(substitute(ts))),
       ci.col = "blue")
```

Arguments

ts	a univariate time series
taper	proportion tapered in forming the periodogram
main	main title
ci.col	colour for confidence band.

Value

None.

Side Effects

Plots the cumulative periodogram in a square plot.

Note

From package **MASS**.

Author(s)

B.D. Ripley

Examples

```
require(graphics)

par(pty = "s", mfrow = c(1,2))
cpgram(lh)
lh.ar <- ar(lh, order.max = 9)
cpgram(lh.ar$resid, main = "AR(3) fit to lh")

cpgram(ldeaths)
```

---

cutree *Cut a tree into groups of data*

Description

Cuts a tree, e.g., as resulting from `hclust`, into several groups either by specifying the desired number(s) of groups or the cut height(s).

Usage

```
cutree(tree, k = NULL, h = NULL)
```

Arguments

`tree`	a tree as produced by `hclust`. `cutree()` only expects a list with components `merge`, `height`, and `labels`, of appropriate content each.
`k`	an integer scalar or vector with the desired number of groups
`h`	numeric scalar or vector with heights where the tree should be cut.

At least one of `k` or `h` must be specified, `k` overrides `h` if both are given.

Value

`cutree` returns a vector with group memberships if `k` or `h` are scalar, otherwise a matrix with group memberships is returned where each column corresponds to the elements of `k` or `h`, respectively (which are also used as column names).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`hclust`, `dendrogram` for cutting trees themselves.

Examples

```
hc <- hclust(dist(USArrests))

cutree(hc, k=1:5)#k = 1 is trivial
cutree(hc, h=250)

## Compare the 2 and 3 grouping:
g24 <- cutree(hc, k = c(2,4))
table(g24[,"2"], g24[,"4"])
```

---

decompose *Classical Seasonal Decomposition by Moving Averages*

Description

Decompose a time series into seasonal, trend and irregular components using moving averages. Deals with additive or multiplicative seasonal component.

Usage

```
decompose(x, type = c("additive", "multiplicative"), filter = NULL)
```

Arguments

x	A time series.
type	The type of seasonal component. Can be abbreviated.
filter	A vector of filter coefficients in reverse time order (as for AR or MA coefficients), used for filtering out the seasonal component. If NULL, a moving average with symmetric window is performed.

Details

The additive model used is:

$$Y_t = T_t + S_t + e_t$$

The multiplicative model used is:

$$Y_t = T_t \, S_t \, e_t$$

The function first determines the trend component using a moving average (if filter is NULL, a symmetric window with equal weights is used), and removes it from the time series. Then, the seasonal figure is computed by averaging, for each time unit, over all periods. The seasonal figure is then centered. Finally, the error component is determined by removing trend and seasonal figure (recycled as needed) from the orginal time series.

Value

An object of class "decomposed.ts" with following components:

seasonal	The seasonal component (i.e., the repeated seasonal figure)
figure	The estimated seasonal figure only
trend	The trend component
random	The remainder part
type	The value of type

Note

The function stl provides a much more sophisticated decomposition.

Author(s)

David Meyer ⟨David.Meyer@wu-wien.ac.at⟩

See Also

stl

Examples

```
require(graphics)

m <- decompose(co2)
m$figure
plot(m)
```

---

delete.response *Modify Terms Objects*

Description

delete.response returns a terms object for the same model but with no response variable.

drop.terms removes variables from the right-hand side of the model. There is also a "[.terms" method to perform the same function (with keep.response=TRUE).

reformulate creates a formula from a character vector.

Usage

```
delete.response(termobj)

reformulate(termlabels, response = NULL)

drop.terms(termobj, dropx = NULL, keep.response = FALSE)
```

Arguments

termobj	A terms object
termlabels	character vector giving the right-hand side of a model formula. Cannot be zero-length.
response	character string, symbol or call giving the left-hand side of a model formula, or NULL.
dropx	vector of positions of variables to drop from the right-hand side of the model.
keep.response	Keep the response in the resulting object?

Value

`delete.response` and `drop.terms` return a `terms` object.

`reformulate` returns a `formula`.

See Also

`terms`

Examples

```
ff <- y ~ z + x + w
tt <- terms(ff)
tt
delete.response(tt)
drop.terms(tt, 2:3, keep.response = TRUE)
tt[-1]
tt[2:3]
reformulate(attr(tt, "term.labels"))

## keep LHS :
reformulate("x*w", ff[[2]])
fS <- surv(ft, case) ~ a + b
reformulate(c("a", "b*f"), fS[[2]])

stopifnot(identical(      ~ var, reformulate("var")),
          identical(~ a + b + c, reformulate(letters[1:3])),
          identical(  y ~ a + b, reformulate(letters[1:2], "y"))
         )
```

---

dendrapply *Apply a Function to All Nodes of a Dendrogram*

Description

Apply function `FUN` to each node of a `dendrogram` recursively. When `y <- dendrapply(x, fn)`, then `y` is a dendrogram of the same graph structure as `x` and for each node, `y.node[j] <- FUN( x.node[j], ...)` (where `y.node[j]` is an (invalid!) notation for the j-th node of y.

Usage

```
dendrapply(X, FUN, ...)
```

Arguments

`X`	an object of class `"dendrogram"`.
`FUN`	an R function to be applied to each dendrogram node, typically working on its `attributes` alone, returning an altered version of the same node.
`...`	potential further arguments passed to `FUN`.

Value

Usually a dendrogram of the same (graph) structure as `X`. For that, the function must be conceptually of the form `FUN <- function(X) { attributes(X) <- .....; X }`, i.e. returning the node with some attributes added or changed.

Note

this is still somewhat experimental, and suggestions for enhancements (or nice examples of usage) are very welcome.

Author(s)

Martin Maechler

See Also

`as.dendrogram`, `lapply` for applying a function to each component of a `list`, `rapply` for doing so to each non-list component of a nested list.

Examples

```
require(graphics)

## a smallish simple dendrogram
dhc <- as.dendrogram(hc <- hclust(dist(USArrests), "ave"))
(dhc21 <- dhc[[2]][[1]])

## too simple:
dendrapply(dhc21, function(n) utils::str(attributes(n)))

## toy example to set colored leaf labels :
local({
  colLab <<- function(n) {
      if(is.leaf(n)) {
        a <- attributes(n)
        i <<- i+1
        attr(n, "nodePar") <-
            c(a$nodePar, list(lab.col = mycols[i], lab.font= i%%3))
      }
      n
  }
  mycols <- grDevices::rainbow(attr(dhc21,"members"))
  i <- 0
 })
dL <- dendrapply(dhc21, colLab)
op <- par(mfrow=2:1)
 plot(dhc21)
 plot(dL) ## --> colored labels!
par(op)
```

dendrogram *General Tree Structures*

Description

Class "dendrogram" provides general functions for handling tree-like structures. It is intended as a replacement for similar functions in hierarchical clustering and classification/regression trees, such that all of these can use the same engine for plotting or cutting trees.

The code is still in testing stage and the API may change in the future.

Usage

```
as.dendrogram(object, ...)
## S3 method for class 'hclust':
as.dendrogram(object, hang = -1, ...)

## S3 method for class 'dendrogram':
plot(x, type = c("rectangle", "triangle"),
      center = FALSE,
      edge.root = is.leaf(x) || !is.null(attr(x,"edgetext")),
      nodePar = NULL, edgePar = list(),
      leaflab = c("perpendicular", "textlike", "none"),
      dLeaf = NULL, xlab = "", ylab = "", xaxt = "n", yaxt = "s",
      horiz = FALSE, frame.plot = FALSE, xlim, ylim, ...)

## S3 method for class 'dendrogram':
cut(x, h, ...)

## S3 method for class 'dendrogram':
print(x, digits, ...)

## S3 method for class 'dendrogram':
rev(x)

## S3 method for class 'dendrogram':
str(object, max.level = NA, digits.d = 3,
    give.attr = FALSE, wid = getOption("width"),
    nest.lev = 0, indent.str = "", stem = "--", ...)

is.leaf(object)
```

Arguments

object	any R object that can be made into one of class "dendrogram".
x	object of class "dendrogram".
hang	numeric scalar indicating how the *height* of leaves should be computed from the heights of their parents; see plot.hclust.
type	type of plot.
center	logical; if TRUE, nodes are plotted centered with respect to the leaves in the branch. Otherwise (default), plot them in the middle of all direct child nodes.

edge.root	logical; if true, draw an edge to the root node.
nodePar	a list of plotting parameters to use for the nodes (see points) or NULL by default which does not draw symbols at the nodes. The list may contain components named pch, cex, col, and/or bg each of which can have length two for specifying separate attributes for *inner* nodes and *leaves*.
edgePar	a list of plotting parameters to use for the edge segments and labels (if there's an edgetext). The list may contain components named col, lty and lwd (for the segments), p.col, p.lwd, and p.lty (for the polygon around the text) and t.col for the text color. As with nodePar, each can have length two for differentiating leaves and inner nodes.
leaflab	a string specifying how leaves are labeled. The default "perpendicular" write text vertically (by default). "textlike" writes text horizontally (in a rectangle), and "none" suppresses leaf labels.
dLeaf	a number specifying the **d**istance in user coordinates between the tip of a leaf and its label. If NULL as per default, 3/4 of a letter width or height is used.
horiz	logical indicating if the dendrogram should be drawn *horizontally* or not.
frame.plot	logical indicating if a box around the plot should be drawn, see plot.default.
h	height at which the tree is cut.
xlim, ylim	optional x- and y-limits of the plot, passed to plot.default. The defaults for these show the full dendrogram.
..., xlab, ylab, xaxt, yaxt	graphical parameters, or arguments for other methods.
digits	integer specifying the precision for printing, see print.default.
max.level, digits.d, give.attr, wid, nest.lev, indent.str	arguments to str, see str.default(). Note that give.attr = FALSE still shows height and members attributes for each node.
stem	a string used for str() specifying the *stem* to use for each dendrogram branch.

Details

Warning: This documentation is preliminary.

The dendrogram is directly represented as a nested list where each component corresponds to a branch of the tree. Hence, the first branch of tree z is z[[1]], the second branch of the corresponding subtree is z[[1]][[2]] etc.. Each node of the tree carries some information needed for efficient plotting or cutting as attributes, of which only members, height and leaf for leaves are compulsory:

members total number of leaves in the branch

height numeric non-negative height at which the node is plotted.

midpoint numeric horizontal distance of the node from the left border (the leftmost leaf) of the branch (unit 1 between all leaves). This is used for plot(*, center=FALSE).

label character; the label of the node

x.member for cut()$upper, the number of *former* members; more generally a substitute for the members component used for 'horizontal' (when horiz = FALSE, else 'vertical') alignment.

edgetext character; the label for the edge leading to the node

nodePar a named list (of length-1 components) specifying node-specific attributes for points plotting, see the nodePar argument above.

edgePar a named list (of length-1 components) specifying attributes for segments plotting of the edge leading to the node, and drawing of the edgetext if available, see the edgePar argument above.

leaf logical, if TRUE, the node is a leaf of the tree.

cut.dendrogram() returns a list with components $upper and $lower, the first is a truncated version of the original tree, also of class dendrogram, the latter a list with the branches obtained from cutting the tree, each a dendrogram.

There are [[, print, and str methods for "dendrogram" objects where the first one (extraction) ensures that selecting sub-branches keeps the class.

Objects of class "hclust" can be converted to class "dendrogram" using method as.dendrogram.

rev.dendrogram simply returns the dendrogram x with reversed nodes, see also reorder.dendrogram.

is.leaf(object) is logical indicating if object is a leaf (the most simple dendrogram). plotNode() and plotNodeLimit() are helper functions.

Warning

Some operations on dendrograms (including plotting) make use of recursion. For very deep trees It may be necessary to increase options("expressions"): if you do you are likely to need to set the C stack size larger than the OS default if possible (which it is not on Windows).

Note

When using type = "triangle", center = TRUE often looks better.

See Also

order.dendrogram also on the labels method for dendrograms.

Examples

```
require(graphics); require(utils)

hc <- hclust(dist(USArrests), "ave")
(dend1 <- as.dendrogram(hc)) # "print()" method
str(dend1)          # "str()" method
str(dend1, max = 2) # only the first two sub-levels

op <- par(mfrow= c(2,2), mar = c(5,2,1,4))
plot(dend1)
## "triangle" type and show inner nodes:
plot(dend1, nodePar=list(pch = c(1,NA), cex=0.8, lab.cex = 0.8),
      type = "t", center=TRUE)
plot(dend1, edgePar=list(col = 1:2, lty = 2:3),
     dLeaf=1, edge.root = TRUE)
plot(dend1, nodePar=list(pch = 2:1,cex=.4*2:1, col = 2:3),
     horiz=TRUE)

dend2 <- cut(dend1, h=70)
```

```
plot(dend2$upper)
## leafs are wrong horizontally:
plot(dend2$upper, nodePar=list(pch = c(1,7), col = 2:1))
##  dend2$lower is *NOT* a dendrogram, but a list of .. :
plot(dend2$lower[[3]], nodePar=list(col=4), horiz = TRUE, type = "tr")
## "inner" and "leaf" edges in different type & color :
plot(dend2$lower[[2]], nodePar=list(col=1),# non empty list
     edgePar = list(lty=1:2, col=2:1), edge.root=TRUE)
par(op)
str(d3 <- dend2$lower[[2]][[2]][[1]])

## "Zoom" in to the first dendrogram :
plot(dend1, xlim = c(1,20), ylim = c(1,50))

nP <- list(col=3:2, cex=c(2.0, 0.75), pch= 21:22,
           bg= c("light blue", "pink"),
           lab.cex = 0.75, lab.col = "tomato")
plot(d3, nodePar= nP, edgePar = list(col="gray", lwd=2), horiz = TRUE)
addE <- function(n) {
      if(!is.leaf(n)) {
        attr(n, "edgePar") <- list(p.col="plum")
        attr(n, "edgetext") <- paste(attr(n,"members"),"members")
      }
      n
}
d3e <- dendrapply(d3, addE)
plot(d3e, nodePar= nP)
plot(d3e, nodePar= nP, leaflab = "textlike")
```

`density` *Kernel Density Estimation*

Description

The (S3) generic function `density` computes kernel density estimates. Its default method does so with the given kernel and bandwidth for univariate observations.

Usage

```
density(x, ...)
## Default S3 method:
density(x, bw = "nrd0", adjust = 1,
        kernel = c("gaussian", "epanechnikov", "rectangular",
                   "triangular", "biweight",
                   "cosine", "optcosine"),
        weights = NULL, window = kernel, width,
        give.Rkern = FALSE,
        n = 512, from, to, cut = 3, na.rm = FALSE, ...)
```

Arguments

`x`	the data from which the estimate is to be computed.

`bw` the smoothing bandwidth to be used. The kernels are scaled such that this is the standard deviation of the smoothing kernel. (Note this differs from the reference books cited below, and from S-PLUS.)

`bw` can also be a character string giving a rule to choose the bandwidth. See `bw.nrd`.

The specified (or computed) value of `bw` is multiplied by `adjust`.

`adjust` the bandwidth used is actually `adjust*bw`. This makes it easy to specify values like 'half the default' bandwidth.

`kernel, window`
a character string giving the smoothing kernel to be used. This must be one of `"gaussian"`, `"rectangular"`, `"triangular"`, `"epanechnikov"`, `"biweight"`, `"cosine"` or `"optcosine"`, with default `"gaussian"`, and may be abbreviated to a unique prefix (single letter).

`"cosine"` is smoother than `"optcosine"`, which is the usual 'cosine' kernel in the literature and almost MSE-efficient. However, `"cosine"` is the version used by S.

`weights` numeric vector of non-negative observation weights, hence of same length as `x`. The default `NULL` is equivalent to `weights = rep(1/nx, nx)` where `nx` is the length of (the finite entries of) `x[]`.

`width` this exists for compatibility with S; if given, and `bw` is not, will set `bw` to `width` if this is a character string, or to a kernel-dependent multiple of `width` if this is numeric.

`give.Rkern` logical; if true, *no* density is estimated, and the 'canonical bandwidth' of the chosen `kernel` is returned instead.

`n` the number of equally spaced points at which the density is to be estimated. When `n > 512`, it is rounded up to the next power of 2 for efficiency reasons (`fft`).

`from,to` the left and right-most points of the grid at which the density is to be estimated; the defaults are `cut * bw` outside of `range(x)`.

`cut` by default, the values of `from` and `to` are `cut` bandwidths beyond the extremes of the data. This allows the estimated density to drop to approximately zero at the extremes.

`na.rm` logical; if `TRUE`, missing values are removed from `x`. If `FALSE` any missing values cause an error.

`...` further arguments for (non-default) methods.

Details

The algorithm used in `density.default` disperses the mass of the empirical distribution function over a regular grid of at least 512 points and then uses the fast Fourier transform to convolve this approximation with a discretized version of the kernel and then uses linear approximation to evaluate the density at the specified points.

The statistical properties of a kernel are determined by $\sigma_K^2 = \int t^2 K(t)dt$ which is always $= 1$ for our kernels (and hence the bandwidth `bw` is the standard deviation of the kernel) and $R(K) = \int K^2(t)dt$.
MSE-equivalent bandwidths (for different kernels) are proportional to $\sigma_K R(K)$ which is scale invariant and for our kernels equal to $R(K)$. This value is returned when `give.Rkern = TRUE`. See the examples for using exact equivalent bandwidths.

Infinite values in `x` are assumed to correspond to a point mass at `+/-Inf` and the density estimate is of the sub-density on `(-Inf, +Inf)`.

Value

If `give.Rkern` is true, the number $R(K)$, otherwise an object with class `"density"` whose underlying structure is a list containing the following components.

`x`	the `n` coordinates of the points where the density is estimated.
`y`	the estimated density values. These will be non-negative, but can be zero.
`bw`	the bandwidth used.
`n`	the sample size after elimination of missing values.
`call`	the call which produced the result.
`data.name`	the deparsed name of the `x` argument.
`has.na`	logical, for compatibility (always `FALSE`).

The `print` method reports `summary` values on the `x` and `y` components.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (for S version).

Scott, D. W. (1992) *Multivariate Density Estimation. Theory, Practice and Visualization*. New York: Wiley.

Sheather, S. J. and Jones M. C. (1991) A reliable data-based bandwidth selection method for kernel density estimation. *J. Roy. Statist. Soc.* **B**, 683–690.

Silverman, B. W. (1986) *Density Estimation*. London: Chapman and Hall.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. New York: Springer.

See Also

`bw.nrd`, `plot.density`, `hist`.

Examples

```
require(graphics)

plot(density(c(-20,rep(0,98),20)), xlim = c(-4,4))# IQR = 0

# The Old Faithful geyser data
d <- density(faithful$eruptions, bw = "sj")
d
plot(d)

plot(d, type = "n")
polygon(d, col = "wheat")

## Missing values:
x <- xx <- faithful$eruptions
x[i.out <- sample(length(x), 10)] <- NA
doR <- density(x, bw = 0.15, na.rm = TRUE)
lines(doR, col = "blue")
points(xx[i.out], rep(0.01, 10))

## Weighted observations:
fe <- sort(faithful$eruptions) # has quite a few non-unique values
```

```
## use 'counts / n' as weights:
dw <- density(unique(fe), weights = table(fe)/length(fe), bw = d$bw)
utils::str(dw) ## smaller n: only 126, but identical estimate:
stopifnot(all.equal(d[1:3], dw[1:3]))

## simulation from a density() fit:
# a kernel density fit is an equally-weighted mixture.
fit <- density(xx)
N <- 1e6
x.new <- rnorm(N, sample(xx, size = N, replace = TRUE), fit$bw)
plot(fit)
lines(density(x.new), col="blue")

(kernels <- eval(formals(density.default)$kernel))

## show the kernels in the R parametrization
plot (density(0, bw = 1), xlab = "",
      main="R's density() kernels with bw = 1")
for(i in 2:length(kernels))
   lines(density(0, bw = 1, kernel =  kernels[i]), col = i)
legend(1.5,.4, legend = kernels, col = seq(kernels),
       lty = 1, cex = .8, y.intersp = 1)

## show the kernels in the S parametrization
plot(density(0, from=-1.2, to=1.2, width=2, kernel="gaussian"), type="l",
     ylim = c(0, 1), xlab="", main="R's density() kernels with width = 1")
for(i in 2:length(kernels))
   lines(density(0, width = 2, kernel =  kernels[i]), col = i)
legend(0.6, 1.0, legend = kernels, col = seq(kernels), lty = 1)

##-------- Semi-advanced theoretic from here on -------------

(RKs <- cbind(sapply(kernels,
                     function(k) density(kernel = k, give.Rkern = TRUE))))
100*round(RKs["epanechnikov",]/RKs, 4) ## Efficiencies

bw <- bw.SJ(precip) ## sensible automatic choice
plot(density(precip, bw = bw),
     main = "same sd bandwidths, 7 different kernels")
for(i in 2:length(kernels))
   lines(density(precip, bw = bw, kernel = kernels[i]), col = i)

## Bandwidth Adjustment for "Exactly Equivalent Kernels"
h.f <- sapply(kernels, function(k)density(kernel = k, give.Rkern = TRUE))
(h.f <- (h.f["gaussian"] / h.f)^ .2)
## -> 1, 1.01, .995, 1.007,... close to 1 => adjustment barely visible..

plot(density(precip, bw = bw),
     main = "equivalent bandwidths, 7 different kernels")
for(i in 2:length(kernels))
   lines(density(precip, bw = bw, adjust = h.f[i], kernel = kernels[i]),
         col = i)
legend(55, 0.035, legend = kernels, col = seq(kernels), lty = 1)
```

`deriv` *Symbolic and Algorithmic Derivatives of Simple Expressions*

Description

Compute derivatives of simple expressions, symbolically.

Usage

```
    D (expr, name)
 deriv(expr, ...)
deriv3(expr, ...)

 ## Default S3 method:
 deriv(expr, namevec, function.arg = NULL, tag = ".expr",
       hessian = FALSE, ...)
 ## S3 method for class 'formula':
 deriv(expr, namevec, function.arg = NULL, tag = ".expr",
       hessian = FALSE, ...)

## Default S3 method:
deriv3(expr, namevec, function.arg = NULL, tag = ".expr",
       hessian = TRUE, ...)
## S3 method for class 'formula':
deriv3(expr, namevec, function.arg = NULL, tag = ".expr",
       hessian = TRUE, ...)
```

Arguments

`expr`	A `expression` or `call` or (except `D`) a formula with no lhs.
`name, namevec`	character vector, giving the variable names (only one for `D()`) with respect to which derivatives will be computed.
`function.arg`	If specified and non-`NULL`, a character vector of arguments for a function return, or a function (with empty body) or `TRUE`, the latter indicating that a function with argument names `namevec` should be used.
`tag`	character; the prefix to be used for the locally created variables in result.
`hessian`	a logical value indicating whether the second derivatives should be calculated and incorporated in the return value.
`...`	arguments to tbe passed to or from methods.

Details

`D` is modelled after its S namesake for taking simple symbolic derivatives.

`deriv` is a *generic* function with a default and a `formula` method. It returns a `call` for computing the `expr` and its (partial) derivatives, simultaneously. It uses so-called *algorithmic derivatives*. If `function.arg` is a function, its arguments can have default values, see the `fx` example below.

Currently, `deriv.formula` just calls `deriv.default` after extracting the expression to the right of `~`.

`deriv3` and its methods are equivalent to `deriv` and its methods except that `hessian` defaults to `TRUE` for `deriv3`.

The internal code knows about the arithmetic operators `+`, `-`, `*`, `/` and `^`, and the single-variable functions `exp`, `log`, `sin`, `cos`, `tan`, `sinh`, `cosh`, `sqrt`, `pnorm`, `dnorm`, `asin`, `acos`, `atan`, `gamma`, `lgamma`, `digamma` and `trigamma`, as well as `psigamma` for one or two arguments (but derivative only with respect to the first). (Note that only the standard normal distribution is considered.)

Value

`D` returns a call and therefore can easily be iterated for higher derivatives.

`deriv` and `deriv3` normally return an `expression` object whose evaluation returns the function values with a `"gradient"` attribute containing the gradient matrix. If `hessian` is `TRUE` the evaluation also returns a `"hessian"` attribute containing the Hessian array.

If `function.arg` is not `NULL`, `deriv` and `deriv3` return a function with those arguments rather than an expression.

References

Griewank, A. and Corliss, G. F. (1991) *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM proceedings, Philadelphia.

Bates, D. M. and Chambers, J. M. (1992) *Nonlinear models.* Chapter 10 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`nlm` and `optim` for numeric minimization which could make use of derivatives,

Examples

```
## formula argument :
dx2x <- deriv(~ x^2, "x") ; dx2x
## Not run:
expression({
         .value <- x^2
         .grad <- array(0, c(length(.value), 1), list(NULL, c("x")))
         .grad[, "x"] <- 2 * x
         attr(.value, "gradient") <- .grad
         .value
})
## End(Not run)
mode(dx2x)
x <- -1:2
eval(dx2x)

## Something 'tougher':
trig.exp <- expression(sin(cos(x + y^2)))
( D.sc <- D(trig.exp, "x") )
all.equal(D(trig.exp[[1]], "x"), D.sc)

( dxy <- deriv(trig.exp, c("x", "y")) )
y <- 1
eval(dxy)
eval(D.sc)

## function returned:
deriv((y ~ sin(cos(x) * y)), c("x","y"), func = TRUE)

## function with defaulted arguments:
(fx <- deriv(y ~ b0 + b1 * 2^(-x/th), c("b0", "b1", "th"),
             function(b0, b1, th, x = 1:7){} ) )
fx(2,3,4)
```

```
## Higher derivatives
deriv3(y ~ b0 + b1 * 2^(-x/th), c("b0", "b1", "th"),
     c("b0", "b1", "th", "x") )

## Higher derivatives:
DD <- function(expr,name, order = 1) {
   if(order < 1) stop("'order' must be >= 1")
   if(order == 1) D(expr,name)
   else DD(D(expr, name), name, order - 1)
}
DD(expression(sin(x^2)), "x", 3)
## showing the limits of the internal "simplify()" :
## Not run:
-sin(x^2) * (2 * x) * 2 + ((cos(x^2) * (2 * x) * (2 * x) + sin(x^2) *
    2) * (2 * x) + sin(x^2) * (2 * x) * 2)
## End(Not run)
```

`deviance` *Model Deviance*

Description

Returns the deviance of a fitted model object.

Usage

```
deviance(object, ...)
```

Arguments

`object`	an object for which the deviance is desired.
`...`	additional optional argument.

Details

This is a generic function which can be used to extract deviances for fitted models. Consult the individual modeling functions for details on how to use this function.

Value

The value of the deviance extracted from the object `object`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`df.residual`, `extractAIC`, `glm`, `lm`.

df.residual *Residual Degrees-of-Freedom*

Description

Returns the residual degrees-of-freedom extracted from a fitted model object.

Usage

```
df.residual(object, ...)
```

Arguments

object	an object for which the degrees-of-freedom are desired.
...	additional optional arguments.

Details

This is a generic function which can be used to extract residual degrees-of-freedom for fitted models. Consult the individual modeling functions for details on how to use this function.

The default method just extracts the `df.residual` component.

Value

The value of the residual degrees-of-freedom extracted from the object `x`.

See Also

`deviance`, `glm`, `lm`.

diffinv *Discrete Integration: Inverse of Differencing*

Description

Computes the inverse function of the lagged differences function `diff`.

Usage

```
diffinv(x, ...)

## Default S3 method:
diffinv(x, lag = 1, differences = 1, xi, ...)
## S3 method for class 'ts':
diffinv(x, lag = 1, differences = 1, xi, ...)
```

Arguments

`x`	a numeric vector, matrix, or time series.
`lag`	a scalar lag parameter.
`differences`	an integer representing the order of the difference.
`xi`	a numeric vector, matrix, or time series containing the initial values for the integrals. If missing, zeros are used.
`...`	arguments passed to or from other methods.

Details

`diffinv` is a generic function with methods for class `"ts"` and `default` for vectors and matrices.

Missing values are not handled.

Value

A numeric vector, matrix, or time series (the latter for the `"ts"` method) representing the discrete integral of `x`.

Author(s)

A. Trapletti

See Also

`diff`

Examples

```
s <- 1:10
d <- diff(s)
diffinv(d, xi = 1)
```

`dist` *Distance Matrix Computation*

Description

This function computes and returns the distance matrix computed by using the specified distance measure to compute the distances between the rows of a data matrix.

Usage

```
dist(x, method = "euclidean", diag = FALSE, upper = FALSE, p = 2)

as.dist(m, diag = FALSE, upper = FALSE)
## Default S3 method:
as.dist(m, diag = FALSE, upper = FALSE)

## S3 method for class 'dist':
```

```
print(x, diag = NULL, upper = NULL,
      digits = getOption("digits"), justify = "none",
      right = TRUE, ...)

## S3 method for class 'dist':
as.matrix(x, ...)
```

Arguments

x	a numeric matrix, data frame or "dist" object.
method	the distance measure to be used. This must be one of "euclidean", "maximum", "manhattan", "canberra", "binary" or "minkowski". Any unambiguous substring can be given.
diag	logical value indicating whether the diagonal of the distance matrix should be printed by print.dist.
upper	logical value indicating whether the upper triangle of the distance matrix should be printed by print.dist.
p	The power of the Minkowski distance.
m	An object with distance information to be converted to a "dist" object. For the default method, a "dist" object, or a matrix (of distances) or an object which can be coerced to such a matrix using as.matrix(). (Only the lower triangle of the matrix is used, the rest is ignored).
digits, justify	passed to format inside of print().
right, ...	further arguments, passed to other methods.

Details

Available distance measures are (written for two vectors x and y):

euclidean: Usual square distance between the two vectors (2 norm).

maximum: Maximum distance between two components of x and y (supremum norm)

manhattan: Absolute distance between the two vectors (1 norm).

canberra: $\sum_i |x_i - y_i| / |x_i + y_i|$. Terms with zero numerator and denominator are omitted from the sum and treated as if the values were missing.

binary: (aka *asymmetric binary*): The vectors are regarded as binary bits, so non-zero elements are 'on' and zero elements are 'off'. The distance is the *proportion* of bits in which only one is on amongst those in which at least one is on.

minkowski: The p norm, the pth root of the sum of the pth powers of the differences of the components.

Missing values are allowed, and are excluded from all computations involving the rows within which they occur. Further, when Inf values are involved, all pairs of values are excluded when their contribution to the distance gave NaN or NA.
If some columns are excluded in calculating a Euclidean, Manhattan, Canberra or Minkowski distance, the sum is scaled up proportionally to the number of columns used. If all pairs are excluded when calculating a particular distance, the value is NA.

The "dist" method of as.matrix() and as.dist() can be used for conversion between objects of class "dist" and conventional distance matrices.

`as.dist()` is a generic function. Its default method handles objects inheriting from class `"dist"`, or coercible to matrices using `as.matrix()`. Support for classes representing distances (also known as dissimilarities) can be added by providing an `as.matrix()` or, more directly, an `as.dist` method for such a class.

Value

`dist` returns an object of class `"dist"`.

The lower triangle of the distance matrix stored by columns in a vector, say `do`. If `n` is the number of observations, i.e., `n <- attr(do, "Size")`, then for $i < j <= n$, the dissimilarity between (row) i and j is `do[n*(i-1) - i*(i-1)/2 + j-i]`. The length of the vector is $n*(n-1)/2$, i.e., of order n^2.

The object has the following attributes (besides `"class"` equal to `"dist"`):

`Size`	integer, the number of observations in the dataset.
`Labels`	optionally, contains the labels, if any, of the observations of the dataset.
`Diag, Upper`	logicals corresponding to the arguments `diag` and `upper` above, specifying how the object should be printed.
`call`	optionally, the `call` used to create the object.
`method`	optionally, the distance method used; resulting from `dist()`, the (`match.arg()`ed) `method` argument.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Mardia, K. V., Kent, J. T. and Bibby, J. M. (1979) *Multivariate Analysis*. Academic Press.

Borg, I. and Groenen, P. (1997) *Modern Multidimensional Scaling. Theory and Applications*. Springer.

See Also

`daisy` in the **cluster** package with more possibilities in the case of *mixed* (continuous / categorical) variables. `hclust`.

Examples

```
require(graphics)

x <- matrix(rnorm(100), nrow=5)
dist(x)
dist(x, diag = TRUE)
dist(x, upper = TRUE)
m <- as.matrix(dist(x))
d <- as.dist(m)
stopifnot(d == dist(x))

## Use correlations between variables "as distance"
dd <- as.dist((1 - cor(USJudgeRatings))/2)
round(1000 * dd) # (prints more nicely)
plot(hclust(dd)) # to see a dendrogram of clustered variables
```

```
## example of binary and canberra distances.
x <- c(0, 0, 1, 1, 1, 1)
y <- c(1, 0, 1, 1, 0, 1)
dist(rbind(x,y), method= "binary")
## answer 0.4 = 2/5
dist(rbind(x,y), method= "canberra")
## answer 2 * (6/5)

## To find the names
labels(eurodist)

## Examples involving "Inf" :
## 1)
x[6] <- Inf
(m2 <- rbind(x,y))
dist(m2, method="binary")# warning, answer 0.5 = 2/4
## These all give "Inf":
stopifnot(Inf == dist(m2, method= "euclidean"),
          Inf == dist(m2, method= "maximum"),
          Inf == dist(m2, method= "manhattan"))
##  "Inf" is same as very large number:
x1 <- x; x1[6] <- 1e100
stopifnot(dist(cbind(x ,y), method="canberra") ==
    print(dist(cbind(x1,y), method="canberra")))

## 2)
y[6] <- Inf #-> 6-th pair is excluded
dist(rbind(x,y), method="binary")   # warning; 0.5
dist(rbind(x,y), method="canberra") # 3
dist(rbind(x,y), method="maximum")  # 1
dist(rbind(x,y), method="manhattan")# 2.4
```

`dummy.coef` *Extract Coefficients in Original Coding*

Description

This extracts coefficients in terms of the original levels of the coefficients rather than the coded variables.

Usage

```
dummy.coef(object, ...)

## S3 method for class 'lm':
dummy.coef(object, use.na = FALSE, ...)

## S3 method for class 'aovlist':
dummy.coef(object, use.na = FALSE, ...)
```

Arguments

`object`	a linear model fit.

use.na	logical flag for coefficients in a singular model. If use.na is true, undetermined coefficients will be missing; if false they will get one possible value.
...	arguments passed to or from other methods.

Details

A fitted linear model has coefficients for the contrasts of the factor terms, usually one less in number than the number of levels. This function re-expresses the coefficients in the original coding; as the coefficients will have been fitted in the reduced basis, any implied constraints (e.g., zero sum for contr.helmert or contr.sum will be respected. There will be little point in using dummy.coef for contr.treatment contrasts, as the missing coefficients are by definition zero.

The method used has some limitations, and will give incomplete results for terms such as poly(x, 2)). However, it is adequate for its main purpose, aov models.

Value

A list giving for each term the values of the coefficients. For a multistratum aov model, such a list for each stratum.

Warning

This function is intended for human inspection of the output: it should not be used for calculations. Use coded variables for all calculations.

The results differ from S for singular values, where S can be incorrect.

See Also

aov, model.tables

Examples

```
options(contrasts=c("contr.helmert", "contr.poly"))
## From Venables and Ripley (2002) p.165.
utils::data(npk, package="MASS")
npk.aov <- aov(yield ~ block + N*P*K, npk)
dummy.coef(npk.aov)

npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
dummy.coef(npk.aovE)
```

---

ecdf *Empirical Cumulative Distribution Function*

Description

Compute or plot an empirical cumulative distribution function.

Usage

```
ecdf(x)

## S3 method for class 'ecdf':
plot(x, ..., ylab="Fn(x)", verticals = FALSE,
     col.01line = "gray70", pch = 19)

## S3 method for class 'ecdf':
print(x, digits= getOption("digits") - 2, ...)

## S3 method for class 'ecdf':
summary(object, ...)
```

Arguments

`x, object`	numeric vector of the observations for `ecdf`; for the methods, an object inheriting from class `"ecdf"`.
`...`	arguments to be passed to subsequent methods, e.g., `plot.stepfun` for the `plot` method.
`ylab`	label for the y-axis.
`verticals`	see `plot.stepfun`.
`col.01line`	numeric or character specifying the color of the horizontal lines at y = 0 and 1, see `colors`.
`pch`	plotting character.
`digits`	number of significant digits to use, see `print`.

Details

The e.c.d.f. (empirical cumulative distribution function) F_n is a step function with jumps i/n at observation values, where i is the number of tied observations at that value. Missing values are ignored.

For observations `x`$= (x_1, x_2, \ldots x_n)$, F_n is the fraction of observations less or equal to t, i.e.,

$$F_n(t) = \#\{x_i \le t\} \,/n = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{[x_i \le t]}.$$

The function `plot.ecdf` which implements the `plot` method for `ecdf` objects, is implemented via a call to `plot.stepfun`; see its documentation.

Value

For `ecdf`, a function of class `"ecdf"`, inheriting from the `"stepfun"` class.

For the `summary` method, a summary of the knots of `object` with a `"header"` attribute.

Author(s)

Martin Maechler, ⟨maechler@stat.math.ethz.ch⟩.
Corrections by R-core.

See Also

`stepfun`, the more general class of step functions, `approxfun` and `splinefun`.

Examples

```
##-- Simple didactical  ecdf  example :
x <- rnorm(12)
Fn <- ecdf(x)
Fn     # a *function*
Fn(x)  # returns the percentiles for x
tt <- seq(-2,2, by = 0.1)
12 * Fn(tt) # Fn is a 'simple' function {with values k/12}
summary(Fn)
##--> see below for graphics
knots(Fn)# the unique data values {12 of them if there were no ties}

y <- round(rnorm(12),1); y[3] <- y[1]
Fn12 <- ecdf(y)
Fn12
knots(Fn12)# unique values (always less than 12!)
summary(Fn12)
summary.stepfun(Fn12)

## Advanced: What's inside the function closure?
print(ls.Fn12 <- ls(environment(Fn12)))
##[1] "f"  "method"  "n"  "x"  "y"  "yleft"  "yright"
utils::ls.str(environment(Fn12))

###----------------- Plotting --------------------------
require(graphics)

op <- par(mfrow=c(3,1), mgp=c(1.5, 0.8,0), mar= .1+c(3,3,2,1))

F10 <- ecdf(rnorm(10))
summary(F10)

plot(F10)
plot(F10, verticals= TRUE, do.points = FALSE)

plot(Fn12 , lwd = 2) ; mtext("lwd = 2", adj=1)
xx <- unique(sort(c(seq(-3, 2, length=201), knots(Fn12))))
lines(xx, Fn12(xx), col='blue')
abline(v=knots(Fn12),lty=2,col='gray70')

plot(xx, Fn12(xx), type='o', cex=.1)#- plot.default {ugly}
plot(Fn12, col.hor='red', add= TRUE)  #- plot method
abline(v=knots(Fn12),lty=2,col='gray70')
## luxury plot
plot(Fn12, verticals=TRUE, col.points='blue',
     col.hor='red', col.vert='bisque')

##-- this works too (automatic call to  ecdf(.)):
plot.ecdf(rnorm(24))
title("via  simple  plot.ecdf(x)", adj=1)

par(op)
```

`eff.aovlist` *Compute Efficiencies of Multistratum Analysis of Variance*

Description

Computes the efficiencies of fixed-effect terms in an analysis of variance model with multiple strata.

Usage

```
eff.aovlist(aovlist)
```

Arguments

`aovlist`	The result of a call to `aov` with an `Error` term.

Details

Fixed-effect terms in an analysis of variance model with multiple strata may be estimable in more than one stratum, in which case there is less than complete information in each. The efficiency for a term is the fraction of the maximum possible precision (inverse variance) obtainable by estimating in just that stratum. Under the assumption of balance, this is the same for all contrasts involving that term.

This function is used to pick strata in which to estimate terms in `model.tables.aovlist` and `se.contrast.aovlist`.

In many cases terms will only occur in one stratum, when all the efficiencies will be one: this is detected and no further calculations are done.

The calculation used requires orthogonal contrasts for each term, and will throw an error if non-orthogonal contrasts (e.g. treatment contrasts or an unbalanced design) are detected.

Value

A matrix giving for each non-pure-error stratum (row) the efficiencies for each fixed-effect term in the model.

References

Heiberger, R. M. (1989) *Computation for the Analysis of Designed Experiments*. Wiley.

See Also

`aov`, `model.tables.aovlist`, `se.contrast.aovlist`

Examples

```
## An example from Yates (1932),
## a 2^3 design in 2 blocks replicated 4 times

Block <- gl(8, 4)
A <- factor(c(0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,
              0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1))
B <- factor(c(0,0,1,1,0,0,1,1,0,1,0,1,1,0,1,0,0,0,1,1,
              0,0,1,1,0,0,1,1,0,0,1,1))
```

```
C <- factor(c(0,1,1,0,1,0,0,1,0,0,1,1,0,0,1,1,0,1,0,1,
              1,0,1,0,0,0,1,1,1,1,0,0))
Yield <- c(101, 373, 398, 291, 312, 106, 265, 450, 106, 306, 324, 449,
           272, 89, 407, 338, 87, 324, 279, 471, 323, 128, 423, 334,
           131, 103, 445, 437, 324, 361, 302, 272)
aovdat <- data.frame(Block, A, B, C, Yield)

old <- getOption("contrasts")
options(contrasts=c("contr.helmert", "contr.poly"))
(fit <- aov(Yield ~ A*B*C + Error(Block), data = aovdat))
eff.aovlist(fit)
options(contrasts = old)
```

`effects` *Effects from Fitted Model*

Description

Returns (orthogonal) effects from a fitted model, usually a linear model. This is a generic function, but currently only has a methods for objects inheriting from classes `"lm"` and `"glm"`.

Usage

```
effects(object, ...)

## S3 method for class 'lm':
effects(object, set.sign = FALSE, ...)
```

Arguments

`object`	an R object; typically, the result of a model fitting function such as `lm`.
`set.sign`	logical. If `TRUE`, the sign of the effects corresponding to coefficients in the model will be set to agree with the signs of the corresponding coefficients, otherwise the sign is arbitrary.
`...`	arguments passed to or from other methods.

Details

For a linear model fitted by `lm` or `aov`, the effects are the uncorrelated single-degree-of-freedom values obtained by projecting the data onto the successive orthogonal subspaces generated by the QR decomposition during the fitting process. The first r (the rank of the model) are associated with coefficients and the remainder span the space of residuals (but are not associated with particular residuals).

Empty models do not have effects.

Value

A (named) numeric vector of the same length as `residuals`, or a matrix if there were multiple responses in the fitted model, in either case of class `"coef"`.

The first r rows are labelled by the corresponding coefficients, and the remaining rows are unlabelled. Note that in rank-deficient models the corresponding coefficients will be in a different order if pivoting occurred.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

coef

Examples

```
y <- c(1:3,7,5)
x <- c(1:3,6:7)
( ee <- effects(lm(y ~ x)) )
c( round(ee - effects(lm(y+10 ~ I(x-3.8))), 3) )
# just the first is different
```

---

embed — *Embedding a Time Series*

Description

Embeds the time series `x` into a low-dimensional Euclidean space.

Usage

```
embed (x, dimension = 1)
```

Arguments

`x`	a numeric vector, matrix, or time series.
`dimension`	a scalar representing the embedding dimension.

Details

Each row of the resulting matrix consists of sequences `x[t]`, `x[t-1]`, ..., `x[t-dimension+1]`, where `t` is the original index of `x`. If `x` is a matrix, i.e., `x` contains more than one variable, then `x[t]` consists of the `t`th observation on each variable.

Value

A matrix containing the embedded time series `x`.

Author(s)

A. Trapletti, B.D. Ripley

Examples

```
x <- 1:10
embed (x, 3)
```

`expand.model.frame` *Add new variables to a model frame*

Description

Evaluates new variables as if they had been part of the formula of the specified model. This ensures that the same `na.action` and `subset` arguments are applied and allows, for example, `x` to be recovered for a model using `sin(x)` as a predictor.

Usage

```
expand.model.frame(model, extras,
                   envir = environment(formula(model)),
                   na.expand = FALSE)
```

Arguments

`model`	a fitted model
`extras`	one-sided formula or vector of character strings describing new variables to be added
`envir`	an environment to evaluate things in
`na.expand`	logical; see below

Details

If `na.expand=FALSE` then `NA` values in the extra variables will be passed to the `na.action` function used in `model`. This may result in a shorter data frame (with `na.omit`) or an error (with `na.fail`). If `na.expand=TRUE` the returned data frame will have precisely the same rows as `model.frame(model)`, but the columns corresponding to the extra variables may contain `NA`.

Value

A data frame.

See Also

`model.frame`,`predict`

Examples

```
model <- lm(log(Volume) ~ log(Girth) + log(Height), data=trees)
expand.model.frame(model, ~ Girth) # prints data.frame like

dd <- data.frame(x=1:5, y=rnorm(5), z=c(1,2,NA,4,5))
model <- glm(y ~ x, data=dd, subset=1:4, na.action=na.omit)
expand.model.frame(model, "z", na.expand=FALSE) # = default
expand.model.frame(model, "z", na.expand=TRUE)
```

Exponential *The Exponential Distribution*

Description

Density, distribution function, quantile function and random generation for the exponential distribution with rate `rate` (i.e., mean `1/rate`).

Usage

```
dexp(x, rate = 1, log = FALSE)
pexp(q, rate = 1, lower.tail = TRUE, log.p = FALSE)
qexp(p, rate = 1, lower.tail = TRUE, log.p = FALSE)
rexp(n, rate = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`rate`	vector of rates.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `rate` is not specified, it assumes the default value of `1`.

The exponential distribution with rate λ has density

$$f(x) = \lambda e^{-\lambda x}$$

for $x \geq 0$.

Value

`dexp` gives the density, `pexp` gives the distribution function, `qexp` gives the quantile function, and `rexp` generates random deviates.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-pexp(t, r, lower = FALSE, log = TRUE)`.

Source

`dexp`, `pexp` and `qexp` are all calculated from numerically stable versions of the definitions.

`rexp` uses

Ahrens, J. H. and Dieter, U. (1972). Computer methods for sampling from the exponential and normal distributions. *Communications of the ACM*, **15**, 873–882.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 19. Wiley, New York.

See Also

`exp` for the exponential function, `dgamma` for the gamma distribution and `dweibull` for the Weibull distribution, both of which generalize the exponential.

Examples

```
dexp(1) - exp(-1) #-> 0
```

`extractAIC` *Extract AIC from a Fitted Model*

Description

Computes the (generalized) Akaike **A**n **I**nformation **C**riterion for a fitted parametric model.

Usage

```
extractAIC(fit, scale, k = 2, ...)
```

Arguments

`fit`	fitted model, usually the result of a fitter like `lm`.
`scale`	optional numeric specifying the scale parameter of the model, see `scale` in `step`. Currently only used in the `"lm"` method, where `scale` specifies the estimate of the error variance, and `scale = 0` indicates that it is to be estimated by maximum likelihood.
`k`	numeric specifying the 'weight' of the *equivalent degrees of freedom* ($\equiv$ `edf`) part in the AIC formula.
`...`	further arguments (currently unused in base R).

Details

This is a generic function, with methods in base R for `"aov"`, `"coxph"`, `"glm"`, `"lm"`, `"negbin"` and `"survreg"` classes.

The criterion used is

$$AIC = -2 \log L + k \times \text{edf},$$

where L is the likelihood and `edf` the equivalent degrees of freedom (i.e., the number of free parameters for usual parametric models) of `fit`.

For linear models with unknown scale (i.e., for `lm` and `aov`), $-2 \log L$ is computed from the *deviance* and uses a different additive constant to `logLik` and hence `AIC`. If RSS denotes the (weighted) residual sum of squares then `extractAIC` uses for $-2 \log L$ the formulae $RSS/s - n$ (corresponding to Mallows' C_p) in the case of known scale s and $n \log(RSS/n)$ for unknown scale.

`AIC` only handles unknown scale and uses the formula $n \log(RSS/n) - n + n \log 2\pi - \sum \log w$ where w are the weights.

For `glm` fits the family's `aic()` function to compute the AIC: see the note under `logLik` about the assumptions this makes.

`k = 2` corresponds to the traditional AIC, using `k = log(n)` provides the BIC (Bayesian IC) instead.

Value

A numeric vector of length 2, giving

`edf`	the '**e**quivalent **d**egrees of **f**reedom' for the fitted model `fit`.
`AIC`	the (generalized) Akaike Information Criterion for `fit`.

Note

This function is used in `add1`, `drop1` and `step` and similar functions in package **MASS** from which it was adopted.

Author(s)

B. D. Ripley

References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. New York: Springer (4th ed).

See Also

`AIC`, `deviance`, `add1`, `step`

Examples

```
utils::example(glm)
extractAIC(glm.D93)#>>  5  15.129
```

---

factanal *Factor Analysis*

Description

Perform maximum-likelihood factor analysis on a covariance matrix or data matrix.

Usage

```
factanal(x, factors, data = NULL, covmat = NULL, n.obs = NA,
         subset, na.action, start = NULL,
         scores = c("none", "regression", "Bartlett"),
         rotation = "varimax", control = NULL, ...)
```

Arguments

`x`	A formula or a numeric matrix or an object that can be coerced to a numeric matrix.
`factors`	The number of factors to be fitted.
`data`	An optional data frame (or similar: see `model.frame`), used only if `x` is a formula. By default the variables are taken from `environment(formula)`.
`covmat`	A covariance matrix, or a covariance list as returned by `cov.wt`. Of course, correlation matrices are covariance matrices.
`n.obs`	The number of observations, used if `covmat` is a covariance matrix.
`subset`	A specification of the cases to be used, if `x` is used as a matrix or formula.
`na.action`	The `na.action` to be used if `x` is used as a formula.
`start`	`NULL` or a matrix of starting values, each column giving an initial set of uniquenesses.
`scores`	Type of scores to produce, if any. The default is none, `"regression"` gives Thompson's scores, `"Bartlett"` given Bartlett's weighted least-squares scores. Partial matching allows these names to be abbreviated.
`rotation`	character. `"none"` or the name of a function to be used to rotate the factors: it will be called with first argument the loadings matrix, and should return a list with component `loadings` giving the rotated loadings, or just the rotated loadings.
`control`	A list of control values,
	nstart The number of starting values to be tried if `start = NULL`. Default 1.
	trace logical. Output tracing information? Default `FALSE`.
	lower The lower bound for uniquenesses during optimization. Should be > 0. Default 0.005.
	opt A list of control values to be passed to `optim`'s `control` argument.
	rotate a list of additional arguments for the rotation function.
`...`	Components of `control` can also be supplied as named arguments to `factanal`.

Details

The factor analysis model is

$$x = \Lambda f + e$$

for a p–element row-vector x, a $p \times k$ matrix of *loadings*, a k–element vector of *scores* and a p–element vector of errors. None of the components other than x is observed, but the major restriction is that the scores be uncorrelated and of unit variance, and that the errors be independent with variances Φ, the *uniquenesses*. Thus factor analysis is in essence a model for the covariance matrix of x,

$$\Sigma = \Lambda'\Lambda + \Psi$$

There is still some indeterminacy in the model for it is unchanged if Λ is replaced by $G\Lambda$ for any orthogonal matrix G. Such matrices G are known as *rotations* (although the term is applied also to non-orthogonal invertible matrices).

If `covmat` is supplied it is used. Otherwise `x` is used if it is a matrix, or a formula `x` is used with `data` to construct a model matrix, and that is used to construct a covariance matrix. (It makes no sense for the formula to have a response, and all the variables must be numeric.) Once a covariance

matrix is found or calculated from `x`, it is converted to a correlation matrix for analysis. The correlation matrix is returned as component `correlation` of the result.

The fit is done by optimizing the log likelihood assuming multivariate normality over the uniquenesses. (The maximizing loadings for given uniquenesses can be found analytically: Lawley & Maxwell (1971, p. 27).) All the starting values supplied in `start` are tried in turn and the best fit obtained is used. If `start = NULL` then the first fit is started at the value suggested by Jöreskog (1963) and given by Lawley & Maxwell (1971, p. 31), and then `control$nstart - 1` other values are tried, randomly selected as equal values of the uniquenesses.

The uniquenesses are technically constrained to lie in $[0, 1]$, but near-zero values are problematical, and the optimization is done with a lower bound of `control$lower`, default 0.005 (Lawley & Maxwell, 1971, p. 32).

Scores can only be produced if a data matrix is supplied and used. The first method is the regression method of Thomson (1951), the second the weighted least squares method of Bartlett (1937, 8). Both are estimates of the unobserved scores f. Thomson's method regresses (in the population) the unknown f on x to yield

$$\hat{f} = \Lambda' \Sigma^{-1} x$$

and then substitutes the sample estimates of the quantities on the right-hand side. Bartlett's method minimizes the sum of squares of standardized errors over the choice of f, given (the fitted) Λ.

If `x` is a formula then the standard NA-handling is applied to the scores (if requested): see `napredict`.

Value

An object of class `"factanal"` with components

`loadings`	A matrix of loadings, one column for each factor. The factors are ordered in decreasing order of sums of squares of loadings, and given the sign that will make the sum of the loadings positive.
`uniquenesses`	The uniquenesses computed.
`correlation`	The correlation matrix used.
`criteria`	The results of the optimization: the value of the negative log-likelihood and information on the iterations used.
`factors`	The argument `factors`.
`dof`	The number of degrees of freedom of the factor analysis model.
`method`	The method: always `"mle"`.
`scores`	If requested, a matrix of scores. `napredict` is applied to handle the treatment of values omitted by the `na.action`.
`n.obs`	The number of observations if available, or `NA`.
`call`	The matched call.
`na.action`	If relevant.
`STATISTIC, PVAL`	The significance-test statistic and P value, if if can be computed.

Note

There are so many variations on factor analysis that it is hard to compare output from different programs. Further, the optimization in maximum likelihood factor analysis is hard, and many other examples we compared had less good fits than produced by this function. In particular, solutions which are Heywood cases (with one or more uniquenesses essentially zero) are much often common than most texts and some other programs would lead one to believe.

References

Bartlett, M. S. (1937) The statistical conception of mental factors. *British Journal of Psychology*, **28**, 97–104.

Bartlett, M. S. (1938) Methods of estimating mental factors. *Nature*, **141**, 609–610.

Jöreskog, K. G. (1963) *Statistical Estimation in Factor Analysis.* Almqvist and Wicksell.

Lawley, D. N. and Maxwell, A. E. (1971) *Factor Analysis as a Statistical Method.* Second edition. Butterworths.

Thomson, G. H. (1951) *The Factorial Analysis of Human Ability.* London University Press.

See Also

`print.loadings`, `varimax`, `princomp`, `ability.cov`, `Harman23.cor`, `Harman74.cor`

Examples

```
# A little demonstration, v2 is just v1 with noise,
# and same for v4 vs. v3 and v6 vs. v5
# Last four cases are there to add noise
# and introduce a positive manifold (g factor)
v1 <- c(1,1,1,1,1,1,1,1,1,1,3,3,3,3,3,4,5,6)
v2 <- c(1,2,1,1,1,1,2,1,2,1,3,4,3,3,3,4,6,5)
v3 <- c(3,3,3,3,3,1,1,1,1,1,1,1,1,1,1,5,4,6)
v4 <- c(3,3,4,3,3,1,1,2,1,1,1,1,2,1,1,5,6,4)
v5 <- c(1,1,1,1,1,3,3,3,3,3,1,1,1,1,1,6,4,5)
v6 <- c(1,1,1,2,1,3,3,3,4,3,1,1,1,2,1,6,5,4)
m1 <- cbind(v1,v2,v3,v4,v5,v6)
cor(m1)
factanal(m1, factors=3) # varimax is the default
factanal(m1, factors=3, rotation="promax")
# The following shows the g factor as PC1
prcomp(m1)

## formula interface
factanal(~v1+v2+v3+v4+v5+v6, factors = 3,
         scores = "Bartlett")$scores

## a realistic example from Bartholomew (1987, pp. 61-65)
utils::example(ability.cov)
```

`factor.scope` *Compute Allowed Changes in Adding to or Dropping from a Formula*

Description

`add.scope` and `drop.scope` compute those terms that can be individually added to or dropped from a model while respecting the hierarchy of terms.

Usage

```
add.scope(terms1, terms2)

drop.scope(terms1, terms2)

factor.scope(factor, scope)
```

Arguments

terms1	the terms or formula for the base model.
terms2	the terms or formula for the upper (add.scope) or lower (drop.scope) scope. If missing for drop.scope it is taken to be the null formula, so all terms (except any intercept) are candidates to be dropped.
factor	the "factor" attribute of the terms of the base object.
scope	a list with one or both components drop and add giving the "factor" attribute of the lower and upper scopes respectively.

Details

factor.scope is not intended to be called directly by users.

Value

For add.scope and drop.scope a character vector of terms labels. For factor.scope, a list with components drop and add, character vectors of terms labels.

See Also

add1, drop1, aov, lm

Examples

```
add.scope( ~ a + b + c + a:b,  ~ (a + b + c)^3)
# [1] "a:c" "b:c"
drop.scope( ~ a + b + c + a:b)
# [1] "c"   "a:b"
```

---

family *Family Objects for Models*

Description

Family objects provide a convenient way to specify the details of the models used by functions such as glm. See the documentation for glm for the details on how such model fitting takes place.

Usage

```
family(object, ...)

binomial(link = "logit")
gaussian(link = "identity")
Gamma(link = "inverse")
inverse.gaussian(link = "1/mu^2")
poisson(link = "log")
quasi(link = "identity", variance = "constant")
quasibinomial(link = "logit")
quasipoisson(link = "log")
```

Arguments

link	a specification for the model link function. This can be a name/expression, a literal character string, a length-one character vector or an object of class "link-glm" (such as generated by make.link) provided it is not specified *via* one of the standard names given next. The gaussian family accepts the links (as names) identity, log and inverse; the binomial family the links logit, probit, cauchit, (corresponding to logistic, normal and Cauchy CDFs respectively) log and cloglog (complementary log-log); the Gamma family the links inverse, identity and log; the poisson family the links log, identity, and sqrt and the inverse.gaussian family the links 1/mu^2, inverse, identity and log. The quasi family accepts the links logit, probit, cloglog, identity, inverse, log, 1/mu^2 and sqrt, and the function power can be used to create a power link function.
variance	for all families other than quasi, the variance function is determined by the family. The quasi family will accept the literal character string (or unquoted as a name/expression) specifications "constant", "mu(1-mu)", "mu", "mu^2" and "mu^3", a length-one character vector taking one of those values, or a list containing components varfun, validmu, dev.resids, initialize and name.
object	the function family accesses the family objects which are stored within objects created by modelling functions (e.g., glm).
...	further arguments passed to methods.

Details

family is a generic function with methods for classes "glm" and "lm" (the latter returning gaussian()).

The quasibinomial and quasipoisson families differ from the binomial and poisson families only in that the dispersion parameter is not fixed at one, so they can model over-dispersion. For the binomial case see McCullagh and Nelder (1989, pp. 124–8). Although they show that there is (under some restrictions) a model with variance proportional to mean as in the quasi-binomial model, note that glm does not compute maximum-likelihood estimates in that model. The behaviour of S is closer to the quasi- variants.

Value

An object of class "family" (which has a concise print method). This is a list with elements

family	character: the family name.
link	character: the link name.
linkfun	function: the link.
linkinv	function: the inverse of the link function.
variance	function: the variance as a function of the mean.
dev.resids	function giving the deviance residuals as a function of (y, mu, wt).
aic	function giving the AIC value if appropriate (but NA for the quasi- families). See logLik for the assumptions made about the dispersion parameter.
mu.eta	function: derivative function(eta) $d\mu/d\eta$.
initialize	expression. This needs to set up whatever data objects are needed for the family as well as n (needed for AIC in the binomial family) and mustart (see glm.
valid.mu	logical function. Returns TRUE if a mean vector mu is within the domain of variance.
valid.eta	logical function. Returns TRUE if a linear predictor eta is within the domain of linkinv.

Note

The link and variance arguments have rather awkward semantics for back-compatibility. The recommended way is to supply them is as quoted character strings, but they can also be supplied unquoted (as names or expressions). In addition, they can also be supplied as a length-one character vector giving the name of one of the options, or as a list (for link, of class "link-glm"). The restrictions apply only to links given as names: when given as a character string all the links known to make.link are accepted.

This is potentially ambiguous: supplying link=logit could mean the unquoted name of a link or the value of object logit. It is interpreted if possible as the name of an allowed link, then as an object. (You can force the interpretation to always be the value of an object via logit[1].)

Author(s)

The design was inspired by S functions of the same names described in Hastie & Pregibon (1992) (except quasibinomial and quasipoisson).

References

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

Dobson, A. J. (1983) *An Introduction to Statistical Modelling.* London: Chapman and Hall.

Cox, D. R. and Snell, E. J. (1981). *Applied Statistics; Principles and Examples.* London: Chapman and Hall.

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

glm, power, make.link.

Examples

```
require(utils) # for str

nf <- gaussian()# Normal family
nf
str(nf)# internal STRucture

gf <- Gamma()
gf
str(gf)
gf$linkinv
gf$variance(-3:4) #- == (.)^2

## quasipoisson. compare with example(glm)
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
d.AD <- data.frame(treatment, outcome, counts)
glm.qD93 <- glm(counts ~ outcome + treatment, family=quasipoisson())
glm.qD93
anova(glm.qD93, test="F")
summary(glm.qD93)
## for Poisson results use
anova(glm.qD93, dispersion = 1, test="Chisq")
summary(glm.qD93, dispersion = 1)

## Example of user-specified link, a logit model for p^days
## See Shaffer, T.  2004. Auk 121(2): 526-540.
logexp <- function(days = 1)
{
    linkfun <- function(mu) qlogis(mu^(1/days))
    linkinv <- function(eta) plogis(eta)^days
    mu.eta <- function(eta) days * plogis(eta)^(days-1) *
      .Call("logit_mu_eta", eta, PACKAGE = "stats")
    valideta <- function(eta) TRUE
    link <- paste("logexp(", days, ")", sep="")
    structure(list(linkfun = linkfun, linkinv = linkinv,
                   mu.eta = mu.eta, valideta = valideta, name = link),
              class = "link-glm")
}
binomial(logexp(3))
## in practice this would be used with a vector of 'days', in
## which case use an offset of 0 in the corresponding formula
## to get the null deviance right.

## Binomial with identity link: often not a good idea.
## Not run: binomial(link=make.link("identity"))

## tests of quasi
x <- rnorm(100)
y <- rpois(100, exp(1+x))
glm(y ~x, family=quasi(variance="mu", link="log"))
# which is the same as
glm(y ~x, family=poisson)
glm(y ~x, family=quasi(variance="mu^2", link="log"))
## Not run: glm(y ~x, family=quasi(variance="mu^3", link="log")) # fails
```

```
y <- rbinom(100, 1, plogis(x))
# needs to set a starting value for the next fit
glm(y ~x, family=quasi(variance="mu(1-mu)", link="logit"), start=c(0,1))
```

`FDist` *The F Distribution*

Description

Density, distribution function, quantile function and random generation for the F distribution with `df1` and `df2` degrees of freedom (and optional non-centrality parameter `ncp`).

Usage

```
df(x, df1, df2, ncp, log = FALSE)
pf(q, df1, df2, ncp, lower.tail = TRUE, log.p = FALSE)
qf(p, df1, df2, ncp, lower.tail = TRUE, log.p = FALSE)
rf(n, df1, df2, ncp)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`df1, df2`	degrees of freedom. `Inf` is allowed.
`ncp`	non-centrality parameter. If omitted the central F is assumed.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

The F distribution with `df1` $= n_1$ and `df2` $= n_2$ degrees of freedom has density

$$f(x) = \frac{\Gamma(n_1/2 + n_2/2)}{\Gamma(n_1/2)\Gamma(n_2/2)} \left(\frac{n_1}{n_2}\right)^{n_1/2} x^{n_1/2-1} \left(1 + \frac{n_1 x}{n_2}\right)^{-(n_1+n_2)/2}$$

for $x > 0$.

It is the distribution of the ratio of the mean squares of n_1 and n_2 independent standard normals, and hence of the ratio of two independent chi-squared variates each divided by its degrees of freedom. Since the ratio of a normal and the root mean-square of m independent normals has a Student's t_m distribution, the square of a t_m variate has a F distribution on 1 and m degrees of freedom.

The non-central F distribution is again the ratio of mean squares of independent normals of unit variance, but those in the numerator are allowed to have non-zero means and `ncp` is the sum of squares of the means. See Chisquare for further details on non-central distributions.

Value

`df` gives the density, `pf` gives the distribution function `qf` gives the quantile function, and `rf` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Source

For `df`, and `ncp == 0`, computed via a binomial probability, code contributed by Catherine Loader (see `dbinom`); for `ncp != 0`, computed via a `dbeta`, code contributed by Peter Ruckdeschel.

For `pf`, via `pbeta` (or for large `df2`, via `pchisq`).

For `qf`, via `qchisq` for large `df2`, else via `qbeta`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 2, chapters 27 and 30. Wiley, New York.

See Also

`dchisq` for chi-squared and `dt` for Student's t distributions.

Examples

```
## the density of the square of a t_m is 2*dt(x, m)/(2*x)
# check this is the same as the density of F_{1,m}
x <- seq(0.001, 5, len=100)
all.equal(df(x^2, 1, 5), dt(x, 5)/x)

## Identity:  qf(2*p - 1, 1, df)) == qt(p, df)^2)  for  p >= 1/2
p <- seq(1/2, .99, length=50); df <- 10
rel.err <- function(x,y) ifelse(x==y,0, abs(x-y)/mean(abs(c(x,y))))
quantile(rel.err(qf(2*p - 1, df1=1, df2=df), qt(p, df)^2), .90)# ~= 7e-9
```

`fft` *Fast Discrete Fourier Transform*

Description

Performs the Fast Fourier Transform of an array.

Usage

```
fft(z, inverse = FALSE)
mvfft(z, inverse = FALSE)
```

Arguments

`z`	a real or complex array containing the values to be transformed.
`inverse`	if `TRUE`, the unnormalized inverse transform is computed (the inverse has a + in the exponent of e, but here, we do *not* divide by `1/length(x)`).

Value

When z is a vector, the value computed and returned by fft is the unnormalized univariate Fourier transform of the sequence of values in z. When z contains an array, fft computes and returns the multivariate (spatial) transform. If inverse is TRUE, the (unnormalized) inverse Fourier transform is returned, i.e., if y <- fft(z), then z is fft(y, inverse = TRUE) / length(y).

By contrast, mvfft takes a real or complex matrix as argument, and returns a similar shaped matrix, but with each column replaced by its discrete Fourier transform. This is useful for analyzing vector-valued series.

The FFT is fastest when the length of the series being transformed is highly composite (i.e., has many factors). If this is not the case, the transform may take a long time to compute and will use a large amount of memory.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Singleton, R. C. (1979) Mixed Radix Fast Fourier Transforms, in *Programs for Digital Signal Processing*, IEEE Digital Signal Processing Committee eds. IEEE Press.

See Also

convolve, nextn.

Examples

```
x <- 1:4
fft(x)
fft(fft(x), inverse = TRUE)/length(x)
```

---

filter *Linear Filtering on a Time Series*

Description

Applies linear filtering to a univariate time series or to each series separately of a multivariate time series.

Usage

```
filter(x, filter, method = c("convolution", "recursive"),
       sides = 2, circular = FALSE, init)
```

Arguments

x	a univariate or multivariate time series.
filter	a vector of filter coefficients in reverse time order (as for AR or MA coefficients).
method	Either "convolution" or "recursive" (and can be abbreviated). If "convolution" a moving average is used: if "recursive" an autoregression is used.

sides	for convolution filters only. If sides=1 the filter coefficients are for past values only; if sides=2 they are centred around lag 0. In this case the length of the filter should be odd, but if it is even, more of the filter is forward in time than backward.
circular	for convolution filters only. If TRUE, wrap the filter around the ends of the series, otherwise assume external values are missing (NA).
init	for recursive filters only. Specifies the initial values of the time series just prior to the start value, in reverse time order. The default is a set of zeros.

Details

Missing values are allowed in x but not in filter (where they would lead to missing values everywhere in the output).

Note that there is an implied coefficient 1 at lag 0 in the recursive filter, which gives

$$y_i = x_i + f_1 y_{i-1} + \cdots + f_p y_{i-p}$$

No check is made to see if recursive filter is invertible: the output may diverge if it is not.

The convolution filter is

$$y_i = f_1 x_{i+o} + \cdots + f_p x_{i+o-(p-1)}$$

where o is the offset: see sides for how it is determined.

Value

A time series object.

Note

convolve(, type="filter") uses the FFT for computations and so *may* be faster for long filters on univariate series, but it does not return a time series (and so the time alignment is unclear), nor does it handle missing values. filter is faster for a filter of length 100 on a series of length 1000, for example.

See Also

convolve, arima.sim

Examples

```
x <- 1:100
filter(x, rep(1, 3))
filter(x, rep(1, 3), sides = 1)
filter(x, rep(1, 3), sides = 1, circular = TRUE)

filter(presidents, rep(1,3))
```

`fisher.test` *Fisher's Exact Test for Count Data*

Description

Performs Fisher's exact test for testing the null of independence of rows and columns in a contingency table with fixed marginals.

Usage

```
fisher.test(x, y = NULL, workspace = 200000, hybrid = FALSE,
            control = list(), or = 1, alternative = "two.sided",
            conf.int = TRUE, conf.level = 0.95,
            simulate.p.value = FALSE, B = 2000)
```

Arguments

`x`	either a two-dimensional contingency table in matrix form, or a factor object.
`y`	a factor object; ignored if `x` is a matrix.
`workspace`	an integer specifying the size of the workspace used in the network algorithm. In units of 4 bytes. Only used for non-simulated p-values larger than 2×2 tables.
`hybrid`	a logical. Only used for larger than 2×2 tables, in which cases it indicated whether the exact probabilities (default) or a hybrid approximation thereof should be computed. See 'Details'.
`control`	a list with named components for low level algorithm control. At present the only one used is `"mult"`, a positive integer $>= 2$ with default 30 used only for larger than 2×2 tables. This says how many times as much space should be allocated to paths as to keys: see file 'fexact.c' in the sources of this package.
`or`	the hypothesized odds ratio. Only used in the 2×2 case.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter. Only used in the 2×2 case.
`conf.int`	logical indicating if a confidence interval should be computed (and returned).
`conf.level`	confidence level for the returned confidence interval. Only used in the 2×2 case if `conf.int = TRUE`.
`simulate.p.value`	a logical indicating whether to compute p-values by Monte Carlo simulation, in larger than 2×2 tables.
`B`	an integer specifying the number of replicates used in the Monte Carlo test.

Details

If `x` is a matrix, it is taken as a two-dimensional contingency table, and hence its entries should be nonnegative integers. Otherwise, both `x` and `y` must be vectors of the same length. Incomplete cases are removed, the vectors are coerced into factor objects, and the contingency table is computed from these.

For 2×2 cases, p-values are obtained directly using the (central or non-central) hypergeometric distribution. Otherwise, computations are based on a C version of the FORTRAN subroutine FEXACT which implements the network developed by Mehta and Patel (1986) and improved by Clarkson, Fan and Joe (1993). The FORTRAN code can be obtained from `http://www.netlib.org/toms/643`. Note this fails (with an error message) when the entries of the table are too large. (It transposes the table if necessary so it has no more rows than columns. One constraint is that the product of the row marginals be less than $2^{31} - 1$.)

For 2×2 tables, the null of conditional independence is equivalent to the hypothesis that the odds ratio equals one. 'Exact' inference can be based on observing that in general, given all marginal totals fixed, the first element of the contingency table has a non-central hypergeometric distribution with non-centrality parameter given by the odds ratio (Fisher, 1935). The alternative for a one-sided test is based on the odds ratio, so `alternative = "greater"` is a test of the odds ratio being bigger than `or`.

Two-sided tests are based on the probabilities of the tables, and take as 'more extreme' all tables with probabilities less than or equal to that of the observed table, the p-value being the sum of such probabilities.

For larger than 2×2 tables and `hybrid = TRUE`, asymptotic chi-squared probabilities are only used if the 'Cochran conditions' are satisfied, that is if no cell has count zero, and more than 80% of the cells have counts at least 5.

Simulation is done conditional on the row and column marginals, and works only if the marginals are strictly positive. (A C translation of the algorithm of Patefield (1981) is used.)

Value

A list with class `"htest"` containing the following components:

`p.value`	the p-value of the test.
`conf.int`	a confidence interval for the odds ratio. Only present in the 2×2 case if argument `conf.int = TRUE`.
`estimate`	an estimate of the odds ratio. Note that the *conditional* Maximum Likelihood Estimate (MLE) rather than the unconditional MLE (the sample odds ratio) is used. Only present in the 2×2 case.
`null.value`	the odds ratio under the null, `or`. Only present in the 2×2 case.
`alternative`	a character string describing the alternative hypothesis.
`method`	the character string `"Fisher's Exact Test for Count Data"`.
`data.name`	a character string giving the names of the data.

References

Agresti, A. (1990) *Categorical data analysis*. New York: Wiley. Pages 59–66.

Fisher, R. A. (1935) The logic of inductive inference. *Journal of the Royal Statistical Society Series A* **98**, 39–54.

Fisher, R. A. (1962) Confidence limits for a cross-product ratio. *Australian Journal of Statistics* **4**, 41.

Fisher, R. A. (1970) *Statistical Methods for Research Workers.* Oliver & Boyd.

Mehta, C. R. and Patel, N. R. (1986) Algorithm 643. FEXACT: A Fortran subroutine for Fisher's exact test on unordered $r * c$ contingency tables. *ACM Transactions on Mathematical Software*, **12**, 154–161.

Clarkson, D. B., Fan, Y. and Joe, H. (1993) A Remark on Algorithm 643: FEXACT: An Algorithm for Performing Fisher's Exact Test in $r \times c$ Contingency Tables. *ACM Transactions on Mathematical Software*, **19**, 484–488.

Patefield, W. M. (1981) Algorithm AS159. An efficient method of generating r x c tables with given row and column totals. *Applied Statistics* **30**, 91–97.

See Also

chisq.test

Examples

```
## Agresti (1990), p. 61f, Fisher's Tea Drinker
## A British woman claimed to be able to distinguish whether milk or
##  tea was added to the cup first.  To test, she was given 8 cups of
##  tea, in four of which milk was added first.  The null hypothesis
##  is that there is no association between the true order of pouring
##  and the woman's guess, the alternative that there is a positive
##  association (that the odds ratio is greater than 1).
TeaTasting <-
matrix(c(3, 1, 1, 3),
       nrow = 2,
       dimnames = list(Guess = c("Milk", "Tea"),
                       Truth = c("Milk", "Tea")))
fisher.test(TeaTasting, alternative = "greater")
## => p=0.2429, association could not be established

## Fisher (1962, 1970), Criminal convictions of like-sex twins
Convictions <-
matrix(c(2, 10, 15, 3),
       nrow = 2,
       dimnames =
       list(c("Dizygotic", "Monozygotic"),
            c("Convicted", "Not convicted")))
Convictions
fisher.test(Convictions, alternative = "less")
fisher.test(Convictions, conf.int = FALSE)
fisher.test(Convictions, conf.level = 0.95)$conf.int
fisher.test(Convictions, conf.level = 0.99)$conf.int

## A r x c table  Agresti (2002, p. 57) Job Satisfaction
Job <- matrix(c(1,2,1,0, 3,3,6,1, 10,10,14,9, 6,7,12,11), 4, 4,
dimnames = list(income=c("< 15k", "15-25k", "25-40k", "> 40k"),
                satisfaction=c("VeryD", "LittleD", "ModerateS", "VeryS")))
fisher.test(Job)
fisher.test(Job, simulate.p.value=TRUE, B=1e5)
```

`fitted` *Extract Model Fitted Values*

Description

`fitted` is a generic function which extracts fitted values from objects returned by modeling functions. `fitted.values` is an alias for it.

All object classes which are returned by model fitting functions should provide a `fitted` method. (Note that the generic is `fitted` and not `fitted.values`.)

Methods can make use of `napredict` methods to compensate for the omission of missing values. The default and `nls` methods do.

Usage

```
fitted(object, ...)
fitted.values(object, ...)
```

Arguments

`object`	an object for which the extraction of model fitted values is meaningful.
`...`	other arguments.

Value

Fitted values extracted from the object `x`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`coefficients`, `glm`, `lm`, `residuals`.

`fivenum` *Tukey Five-Number Summaries*

Description

Returns Tukey's five number summary (minimum, lower-hinge, median, upper-hinge, maximum) for the input data.

Usage

```
fivenum(x, na.rm = TRUE)
```

Arguments

`x`	numeric, maybe including `NA`s and $\pm$`Inf`s.
`na.rm`	logical; if `TRUE`, all `NA` and `NaN`s are dropped, before the statistics are computed.

Value

A numeric vector of length 5 containing the summary information. See `boxplot.stats` for more details.

See Also

`IQR`, `boxplot.stats`, `median`, `quantile`, `range`.

Examples

```
fivenum(c(rnorm(100),-1:1/0))
```

`fligner.test` *Fligner-Killeen Test of Homogeneity of Variances*

Description

Performs a Fligner-Killeen (median) test of the null that the variances in each of the groups (samples) are the same.

Usage

```
fligner.test(x, ...)

## Default S3 method:
fligner.test(x, g, ...)

## S3 method for class 'formula':
fligner.test(formula, data, subset, na.action, ...)
```

Arguments

`x`	a numeric vector of data values, or a list of numeric data vectors.
`g`	a vector or factor object giving the group for the corresponding elements of `x`. Ignored if `x` is a list.
`formula`	a formula of the form `lhs ~ rhs` where `lhs` gives the data values and `rhs` the corresponding groups.
`data`	an optional matrix or data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`subset`	an optional vector specifying a subset of observations to be used.
`na.action`	a function which indicates what should happen when the data contain `NA`s. Defaults to `getOption("na.action")`.
`...`	further arguments to be passed to or from methods.

Details

If `x` is a list, its elements are taken as the samples to be compared for homogeneity of variances, and hence have to be numeric data vectors. In this case, `g` is ignored, and one can simply use `fligner.test(x)` to perform the test. If the samples are not yet contained in a list, use `fligner.test(list(x, ...))`.

Otherwise, `x` must be a numeric data vector, and `g` must be a vector or factor object of the same length as `x` giving the group for the corresponding elements of `x`.

The Fligner-Killeen (median) test has been determined in a simulation study as one of the many tests for homogeneity of variances which is most robust against departures from normality, see Conover, Johnson & Johnson (1981). It is a k-sample simple linear rank which uses the ranks of the absolute values of the centered samples and weights $a(i) = \mathrm{qnorm}((1+i/(n+1))/2)$. The version implemented here uses median centering in each of the samples (F-K:med X^2 in the reference).

Value

A list of class `"htest"` containing the following components:

`statistic`	the Fligner-Killeen:med X^2 test statistic.
`parameter`	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
`p.value`	the p-value of the test.
`method`	the character string `"Fligner-Killeen test of homogeneity of variances"`.
`data.name`	a character string giving the names of the data.

References

William J. Conover & Mark E. Johnson & Myrle M. Johnson (1981). A comparative study of tests for homogeneity of variances, with applications to the outer continental shelf bidding data. *Technometrics* **23**, 351–361.

See Also

`ansari.test` and `mood.test` for rank-based two-sample test for a difference in scale parameters; `var.test` and `bartlett.test` for parametric tests for the homogeneity of variances.

Examples

```
require(graphics)

plot(count ~ spray, data = InsectSprays)
fligner.test(InsectSprays$count, InsectSprays$spray)
fligner.test(count ~ spray, data = InsectSprays)
## Compare this to bartlett.test()
```

`formula` *Model Formulae*

Description

The generic function `formula` and its specific methods provide a way of extracting formulae which have been included in other objects.

`as.formula` is almost identical, additionally preserving attributes when `object` already inherits from `"formula"`. The default value of the `env` argument is used only when the formula would otherwise lack an environment.

Usage

```
formula(x, ...)
as.formula(object, env = parent.frame())
```

Arguments

`x, object`	R object.
`...`	further arguments passed to or from other methods.
`env`	the environment to associate with the result.

Details

The models fit by, e.g., the `lm` and `glm` functions are specified in a compact symbolic form. The `~` operator is basic in the formation of such models. An expression of the form `y ~ model` is interpreted as a specification that the response `y` is modelled by a linear predictor specified symbolically by `model`. Such a model consists of a series of terms separated by `+` operators. The terms themselves consist of variable and factor names separated by `:` operators. Such a term is interpreted as the interaction of all the variables and factors appearing in the term.

In addition to `+` and `:`, a number of other operators are useful in model formulae. The `*` operator denotes factor crossing: `a*b` interpreted as `a+b+a:b`. The `^` operator indicates crossing to the specified degree. For example `(a+b+c)^2` is identical to `(a+b+c)*(a+b+c)` which in turn expands to a formula containing the main effects for `a`, `b` and `c` together with their second-order interactions. The `%in%` operator indicates that the terms on its left are nested within those on the right. For example `a + b %in% a` expands to the formula `a + a:b`. The `-` operator removes the specified terms, so that `(a+b+c)^2 - a:b` is identical to `a + b + c + b:c + a:c`. It can also used to remove the intercept term: `y ~ x - 1` is a line through the origin. A model with no intercept can be also specified as `y ~ x + 0` or `y ~ 0 + x`.

While formulae usually involve just variable and factor names, they can also involve arithmetic expressions. The formula `log(y) ~ a + log(x)` is quite legal. When such arithmetic expressions involve operators which are also used symbolically in model formulae, there can be confusion between arithmetic and symbolic operator use.

To avoid this confusion, the function `I()` can be used to bracket those portions of a model formula where the operators are used in their arithmetic sense. For example, in the formula `y ~ a + I(b+c)`, the term `b+c` is to be interpreted as the sum of `b` and `c`.

Variable names can be quoted by backticks `` `like this` `` in formulae, although there is no guarantee that all code using formulae will accept such non-syntactic names.

Most model-fitting functions accept formulae with right-hand-side including the function `offset` to indicate terms with a fixed coefficient of one. Some functions accept other 'specials' such as `strata` or `cluster` (see the `specials` argument of `terms.formula`).

There are two special interpretations of `.` in a formula. The usual one is in the context of a `data` argument of model fitting functions and means 'all columns not otherwise in the formula': see `terms.formula`. In the context of `update.formula`, **only**, it means 'what was previously in this part of the formula'.

When `formula` is called on a fitted model object, either a specific method is used (such as that for class `"nls"`) or the default method. The default first looks for a `"formula"` component of the object (and evaluates it), then a `"terms"` component, then a `formula` parameter of the call (and evaluates its value) and finally a `"formula"` attribute.

There is a method for data frames. If there is only one column this forms the RHS with an empty LHS. For more columns, the first column is the LHS of the formula and the remaining columns separated by + form the RHS.

Value

All the functions above produce an object of class `"formula"` which contains a symbolic model formula.

Environments

A formula object has an associated environment, and this environment (rather than the parent environment) is used by `model.frame` to evaluate variables that are not found in the supplied `data` argument.

Formulas created with the ~ operator use the environment in which they were created. Formulas created with `as.formula` will use the `env` argument for their environment. Pre-existing formulas extracted with `as.formula` will only have their environment changed if `env` is given explicitly.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`I`, `offset`.

For formula manipulation: `terms`, and `all.vars`; for typical use: `lm`, `glm`, and `coplot`.

Examples

```
class(fo <- y ~ x1*x2) # "formula"
fo
typeof(fo)# R internal : "language"
terms(fo)

environment(fo)
environment(as.formula("y ~ x"))
environment(as.formula("y ~ x", env=new.env()))

## Create a formula for a model with a large number of variables:
xnam <- paste("x", 1:25, sep="")
(fmla <- as.formula(paste("y ~ ", paste(xnam, collapse= "+"))))
```

formula.nls *Extract Model Formula from nls Object*

Description

Returns the model used to fit `object`.

Usage

```
## S3 method for class 'nls':
formula(x, ...)
```

Arguments

`x`	an object inheriting from class `"nls"`, representing a nonlinear least squares fit.
`...`	further arguments passed to or from other methods.

Value

a formula representing the model used to obtain `object`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`nls`, `formula`

Examples

```
fm1 <- nls(circumference ~ A/(1+exp((B-age)/C)), Orange,
           start = list(A=160, B=700, C = 350))
formula(fm1)
```

friedman.test *Friedman Rank Sum Test*

Description

Performs a Friedman rank sum test with unreplicated blocked data.

Usage

```
friedman.test(y, ...)

## Default S3 method:
friedman.test(y, groups, blocks, ...)

## S3 method for class 'formula':
friedman.test(formula, data, subset, na.action, ...)
```

Arguments

y	either a numeric vector of data values, or a data matrix.
groups	a vector giving the group for the corresponding elements of y if this is a vector; ignored if y is a matrix. If not a factor object, it is coerced to one.
blocks	a vector giving the block for the corresponding elements of y if this is a vector; ignored if y is a matrix. If not a factor object, it is coerced to one.
formula	a formula of the form a ~ b | c, where a, b and c give the data values and corresponding groups and blocks, respectively.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

friedman.test can be used for analyzing unreplicated complete block designs (i.e., there is exactly one observation in y for each combination of levels of groups and blocks) where the normality assumption may be violated.

The null hypothesis is that apart from an effect of blocks, the location parameter of y is the same in each of the groups.

If y is a matrix, groups and blocks are obtained from the column and row indices, respectively. NA's are not allowed in groups or blocks; if y contains NA's, corresponding blocks are removed.

Value

A list with class "htest" containing the following components:

statistic	the value of Friedman's chi-squared statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	the character string "Friedman rank sum test".
data.name	a character string giving the names of the data.

References

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric Statistical Methods.* New York: John Wiley & Sons. Pages 139–146.

See Also

quade.test.

Examples

```
## Hollander & Wolfe (1973), p. 140ff.
## Comparison of three methods ("round out", "narrow angle", and
##  "wide angle") for rounding first base.  For each of 18 players
##  and the three method, the average time of two runs from a point on
##  the first base line 35ft from home plate to a point 15ft short of
##  second base is recorded.
RoundingTimes <-
matrix(c(5.40, 5.50, 5.55,
         5.85, 5.70, 5.75,
         5.20, 5.60, 5.50,
         5.55, 5.50, 5.40,
         5.90, 5.85, 5.70,
         5.45, 5.55, 5.60,
         5.40, 5.40, 5.35,
         5.45, 5.50, 5.35,
         5.25, 5.15, 5.00,
         5.85, 5.80, 5.70,
         5.25, 5.20, 5.10,
         5.65, 5.55, 5.45,
         5.60, 5.35, 5.45,
         5.05, 5.00, 4.95,
         5.50, 5.50, 5.40,
         5.45, 5.55, 5.50,
         5.55, 5.55, 5.35,
         5.45, 5.50, 5.55,
         5.50, 5.45, 5.25,
         5.65, 5.60, 5.40,
         5.70, 5.65, 5.55,
         6.30, 6.30, 6.25),
       nrow = 22,
       byrow = TRUE,
       dimnames = list(1 : 22,
                       c("Round Out", "Narrow Angle", "Wide Angle")))
friedman.test(RoundingTimes)
## => strong evidence against the null that the methods are equivalent
##    with respect to speed

wb <- aggregate(warpbreaks$breaks,
                by = list(w = warpbreaks$wool,
                          t = warpbreaks$tension),
                FUN = mean)
wb
friedman.test(wb$x, wb$w, wb$t)
friedman.test(x ~ w | t, data = wb)
```

`ftable` *Flat Contingency Tables*

Description

Create ‘flat’ contingency tables.

Usage

```
ftable(x, ...)

## Default S3 method:
ftable(..., exclude = c(NA, NaN), row.vars = NULL,
       col.vars = NULL)
```

Arguments

`x, ...`	R objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted, or a contingency table object of class `"table"` or `"ftable"`.
`exclude`	values to use in the exclude argument of `factor` when interpreting non-factor objects.
`row.vars`	a vector of integers giving the numbers of the variables, or a character vector giving the names of the variables to be used for the rows of the flat contingency table.
`col.vars`	a vector of integers giving the numbers of the variables, or a character vector giving the names of the variables to be used for the columns of the flat contingency table.

Details

`ftable` creates 'flat' contingency tables. Similar to the usual contingency tables, these contain the counts of each combination of the levels of the variables (factors) involved. This information is then re-arranged as a matrix whose rows and columns correspond to unique combinations of the levels of the row and column variables (as specified by `row.vars` and `col.vars`, respectively). The combinations are created by looping over the variables in reverse order (so that the levels of the left-most variable vary the slowest). Displaying a contingency table in this flat matrix form (via `print.ftable`, the print method for objects of class `"ftable"`) is often preferable to showing it as a higher-dimensional array.

`ftable` is a generic function. Its default method, `ftable.default`, first creates a contingency table in array form from all arguments except `row.vars` and `col.vars`. If the first argument is of class `"table"`, it represents a contingency table and is used as is; if it is a flat table of class `"ftable"`, the information it contains is converted to the usual array representation using `as.ftable`. Otherwise, the arguments should be R objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted, which are cross-tabulated using `table`. Then, the arguments `row.vars` and `col.vars` are used to collapse the contingency table into flat form. If neither of these two is given, the last variable is used for the columns. If both are given and their union is a proper subset of all variables involved, the other variables are summed out.

When the arguments are R expressions interpreted as factors, additional arguments will be passed to `table` to control how the variable names are displayed; see the last example below.

Function `ftable.formula` provides a formula method for creating flat contingency tables.

There are methods for `as.table` and `as.data.frame`.

Value

`ftable` returns an object of class `"ftable"`, which is a matrix with counts of each combination of the levels of variables with information on the names and levels of the (row and columns) variables stored as attributes `"row.vars"` and `"col.vars"`.

See Also

ftable.formula for the formula interface (which allows a data = . argument); read.ftable for information on reading, writing and coercing flat contingency tables; table for ordinary cross-tabulation; xtabs for formula-based cross-tabulation.

Examples

```
## Start with a contingency table.
ftable(Titanic, row.vars = 1:3)
ftable(Titanic, row.vars = 1:2, col.vars = "Survived")
ftable(Titanic, row.vars = 2:1, col.vars = "Survived")

## Start with a data frame.
x <- ftable(mtcars[c("cyl", "vs", "am", "gear")])
x
ftable(x, row.vars = c(2, 4))

## Start with expressions, use table()'s "dnn" to change labels
ftable(mtcars$cyl, mtcars$vs, mtcars$am, mtcars$gear, row.vars = c(2, 4),
       dnn = c("Cylinders", "V/S", "Transmission", "Gears"))
```

---

ftable.formula *Formula Notation for Flat Contingency Tables*

Description

Produce or manipulate a flat contingency table using formula notation.

Usage

```
## S3 method for class 'formula':
ftable(formula, data = NULL, subset, na.action, ...)
```

Arguments

formula	a formula object with both left and right hand sides specifying the column and row variables of the flat table.
data	a data frame, list or environment (or similar: see model.frame) containing the variables to be cross-tabulated, or a contingency table (see below).
subset	an optional vector specifying a subset of observations to be used. Ignored if data is a contingency table.
na.action	a function which indicates what should happen when the data contain NAs. Ignored if data is a contingency table.
...	further arguments to the default ftable method may also be passed as arguments, see ftable.default.

Details

This is a method of the generic function ftable.

The left and right hand side of formula specify the column and row variables, respectively, of the flat contingency table to be created. Only the + operator is allowed for combining the variables. A . may be used once in the formula to indicate inclusion of all the remaining variables.

If data is an object of class "table" or an array with more than 2 dimensions, it is taken as a contingency table, and hence all entries should be nonnegative. Otherwise, if it is not a flat contingency table (i.e., an object of class "ftable"), it should be a data frame or matrix, list or environment containing the variables to be cross-tabulated. In this case, na.action is applied to the data to handle missing values, and, after possibly selecting a subset of the data as specified by the subset argument, a contingency table is computed from the variables.

The contingency table is then collapsed to a flat table, according to the row and column variables specified by formula.

Value

A flat contingency table which contains the counts of each combination of the levels of the variables, collapsed into a matrix for suitably displaying the counts.

See Also

ftable, ftable.default; table.

Examples

```
Titanic
x <- ftable(Survived ~ ., data = Titanic)
x
ftable(Sex ~ Class + Age, data = x)
```

GammaDist *The Gamma Distribution*

Description

Density, distribution function, quantile function and random generation for the Gamma distribution with parameters shape and scale.

Usage

```
dgamma(x, shape, rate = 1, scale = 1/rate, log = FALSE)
pgamma(q, shape, rate = 1, scale = 1/rate, lower.tail = TRUE,
       log.p = FALSE)
qgamma(p, shape, rate = 1, scale = 1/rate, lower.tail = TRUE,
       log.p = FALSE)
rgamma(n, shape, rate = 1, scale = 1/rate)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`rate`	an alternative way to specify the scale.
`shape, scale`	shape and scale parameters. Must be positive, `scale` strictly.
`log, log.p`	logical; if `TRUE`, probabilities/densities p are returned as $log(p)$.
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `scale` is omitted, it assumes the default value of `1`.

The Gamma distribution with parameters `shape` $= \alpha$ and `scale` $= \sigma$ has density

$$f(x) = \frac{1}{\sigma^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-x/\sigma}$$

for $x \geq 0$, $\alpha > 0$ and $\sigma > 0$. (Here $\Gamma(\alpha)$ is the function implemented by R's `gamma()` and defined in its help. Note that $a = 0$ corresponds to the trivial distribution with all mass at point 0.)

The mean and variance are $E(X) = \alpha\sigma$ and $Var(X) = \alpha\sigma^2$.

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-pgamma(t, ..., lower = FALSE, log = TRUE)`.

Note that for smallish values of `shape` (and moderate `scale`) a large parts of the mass of the Gamma distribution is on values of x so near zero that they will be represented as zero in computer arithmetic. So `rgamma` can well return values which will be represented as zero. (This will also happen for very large values of `scale` since the actual generation is done for `scale=1`.) Similarly, `qgamma` has a very hard job for small `scale`, and warns of potential unreliability for `scale < 1e-10`.

Value

`dgamma` gives the density, `pgamma` gives the distribution function, `qgamma` gives the quantile function, and `rgamma` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Note

The S parametrization is via `shape` and `rate`: S has no `scale` parameter.

`pgamma` is closely related to the incomplete gamma function. As defined by Abramowitz and Stegun 6.5.1 (and by 'Numerical Recipes') this is

$$P(a, x) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt$$

$P(a, x)$ is `pgamma(x, a)`. Other authors (for example Karl Pearson in his 1922 tables) omit the normalizing factor, defining the incomplete gamma function as `pgamma(x, a) * gamma(a)`. A few use the 'upper' incomplete gamma function, the integral from x to ∞ which can be computed by `pgamma(x, a, lower=FALSE) * gamma(a)`, or its normalized version. See also http://en.wikipedia.org/wiki/Incomplete_gamma_function.

Source

dgamma is computed via the Poisson density, using code contributed by Catherine Loader (see dbinom).

pgamma uses an unpublished (and not otherwise documented) algorithm 'mainly by Morten Welinder'.

qgamma is based on a C translation of

Best, D. J. and D. E. Roberts (1975). Algorithm AS91. Percentage points of the chi-squared distribution. *Applied Statistics*, **24**, 385–388.

plus a final Newton step to improve the approximation.

rgamma for shape >= 1 uses

Ahrens, J. H. and Dieter, U. (1982). Generating gamma variates by a modified rejection technique. *Communications of the ACM*, **25**, 47–54,

and for 0 < shape < 1 uses

Ahrens, J. H. and Dieter, U. (1974). Computer methods for sampling from gamma, beta, Poisson and binomial distributions. *Computing*, **12**, 223–246.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Shea, B. L. (1988) Algorithm AS 239, Chi-squared and incomplete Gamma integral, *Applied Statistics (JRSS C)* **37**, 466–473.

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions.* New York: Dover. Chapter 6: Gamma and Related Functions.

See Also

gamma for the gamma function, dbeta for the Beta distribution and dchisq for the chi-squared distribution which is a special case of the Gamma distribution.

Examples

```
-log(dgamma(1:4, shape=1))
p <- (1:9)/10
pgamma(qgamma(p,shape=2), shape=2)
1 - 1/exp(qgamma(p, shape=1))

# even for shape = 0.001 about half the mass is on numbers
# that cannot be represented accurately (and most of those as zero)
pgamma(.Machine$double.xmin, 0.001)
pgamma(5e-324, 0.001)  # on most machines this is the smallest
                       # representable non-zero number
table(rgamma(1e4, 0.001) == 0)/1e4
```

Geometric *The Geometric Distribution*

Description

Density, distribution function, quantile function and random generation for the geometric distribution with parameter `prob`.

Usage

```
dgeom(x, prob, log = FALSE)
pgeom(q, prob, lower.tail = TRUE, log.p = FALSE)
qgeom(p, prob, lower.tail = TRUE, log.p = FALSE)
rgeom(n, prob)
```

Arguments

`x, q`	vector of quantiles representing the number of failures in a sequence of Bernoulli trials before success occurs.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`prob`	probability of success in each trial. `0 < prob <= 1`.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The geometric distribution with `prob` $= p$ has density

$$p(x) = p(1-p)^x$$

for $x = 0, 1, 2, \ldots$, $0 < p \leq 1$.

If an element of `x` is not integer, the result of `pgeom` is zero, with a warning.

The quantile is defined as the smallest value x such that $F(x) \geq p$, where F is the distribution function.

Value

`dgeom` gives the density, `pgeom` gives the distribution function, `qgeom` gives the quantile function, and `rgeom` generates random deviates.

Invalid `prob` will result in return value `NaN`, with a warning.

Source

`dgeom` computes via `dbinom`, using code contributed by Catherine Loader (see `dbinom`).

`pgeom` and `qgeom` are based on the closed-form formulae.

`rgeom` uses the derivation as an exponential mixture of Poissons, see

Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer-Verlag, New York. Page 480.

See Also

dnbinom for the negative binomial which generalizes the geometric distribution.

Examples

```
qgeom((1:9)/10, prob = .2)
Ni <- rgeom(20, prob = 1/4); table(factor(Ni, 0:max(Ni)))
```

---

getInitial *Get Initial Parameter Estimates*

Description

This function evaluates initial parameter estimates for a nonlinear regression model. If `data` is a parameterized data frame or `pframe` object, its `parameters` attribute is returned. Otherwise the object is examined to see if it contains a call to a `selfStart` object whose `initial` attribute can be evaluated.

Usage

```
getInitial(object, data, ...)
```

Arguments

`object`	a formula or a `selfStart` model that defines a nonlinear regression model
`data`	a data frame in which the expressions in the formula or arguments to the `selfStart` model can be evaluated
`...`	optional additional arguments

Value

A named numeric vector or list of starting estimates for the parameters. The construction of many `selfStart` models is such that these "starting" estimates are, in fact, the converged parameter estimates.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

nls, selfStart, selfStart.default, selfStart.formula

Examples

```
PurTrt <- Puromycin[ Puromycin$state == "treated", ]
getInitial( rate ~ SSmicmen( conc, Vm, K ), PurTrt )
```

`glm` *Fitting Generalized Linear Models*

Description

`glm` is used to fit generalized linear models, specified by giving a symbolic description of the linear predictor and a description of the error distribution.

Usage

```
glm(formula, family = gaussian, data, weights, subset,
    na.action, start = NULL, etastart, mustart,
    offset, control = glm.control(...), model = TRUE,
    method = "glm.fit", x = FALSE, y = TRUE, contrasts = NULL,
    ...)

glm.fit(x, y, weights = rep(1, nobs),
        start = NULL, etastart = NULL, mustart = NULL,
        offset = rep(0, nobs), family = gaussian(),
        control = glm.control(), intercept = TRUE)

## S3 method for class 'glm':
weights(object, type = c("prior", "working"), ...)
```

Arguments

`formula`	an object of class "`formula`" (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under 'Details'.
`family`	a description of the error distribution and link function to be used in the model. This can be a character string naming a family function, a family function or the result of a call to a family function. (See `family` for details of family functions.)
`data`	an optional data frame, list or environment (or object coercible by `as.data.frame` to a data frame) containing the variables in the model. If not found in `data`, the variables are taken from `environment(formula)`, typically the environment from which `glm` is called.
`weights`	an optional vector of weights to be used in the fitting process. Should be `NULL` or a numeric vector.
`subset`	an optional vector specifying a subset of observations to be used in the fitting process.
`na.action`	a function which indicates what should happen when the data contain `NA`s. The default is set by the `na.action` setting of `options`, and is `na.fail` if that is unset. The 'factory-fresh' default is `na.omit`. Another possible value is `NULL`, no action. Value `na.exclude` can be useful.
`start`	starting values for the parameters in the linear predictor.
`etastart`	starting values for the linear predictor.
`mustart`	starting values for the vector of means.

offset	this can be used to specify an *a priori* known component to be included in the linear predictor during fitting. This should be NULL or a numeric vector of length either one or equal to the number of cases. One or more offset terms can be included in the formula instead or as well, and if both are specified their sum is used. See model.offset.
control	a list of parameters for controlling the fitting process. See the documentation for glm.control for details.
model	a logical value indicating whether *model frame* should be included as a component of the returned value.
method	the method to be used in fitting the model. The default method "glm.fit" uses iteratively reweighted least squares (IWLS). The only current alternative is "model.frame" which returns the model frame and does no fitting.
x, y	For glm: logical values indicating whether the response vector and model matrix used in the fitting process should be returned as components of the returned value. For glm.fit: x is a design matrix of dimension n * p, and y is a vector of observations of length n.
contrasts	an optional list. See the contrasts.arg of model.matrix.default.
intercept	logical. Should an intercept be included in the *null* model?
object	an object inheriting from class "glm".
type	character, partial matching allowed. Type of weights to extract from the fitted model object.
...	For glm: arguments to be passed by default to glm.control: see argument control. For weights: further arguments passed to or from other methods.

Details

A typical predictor has the form response ~ terms where response is the (numeric) response vector and terms is a series of terms which specifies a linear predictor for response. For binomial and quasibinomial families the response can also be specified as a factor (when the first level denotes failure and all others success) or as a two-column matrix with the columns giving the numbers of successes and failures. A terms specification of the form first + second indicates all the terms in first together with all the terms in second with duplicates removed. The terms in the formula will be re-ordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on: to avoid this pass a terms object as the formula.

A specification of the form first:second indicates the the set of terms obtained by taking the interactions of all terms in first with all terms in second. The specification first*second indicates the *cross* of first and second. This is the same as first + second + first:second.

glm.fit is the workhorse function.

If more than one of etastart, start and mustart is specified, the first in the list will be used. It is often advisable to supply starting values for a quasi family, and also for families with unusual links such as gaussian("log").

All of weights, subset, offset, etastart and mustart are evaluated in the same way as variables in formula, that is first in data and then in the environment of formula.

For the background to warning messages about 'fitted probabilities numerically 0 or 1 occurred' for binomial GLMs, see Venables & Ripley (2002, pp. 197–8).

Value

`glm` returns an object of class inheriting from `"glm"` which inherits from the class `"lm"`. See later in this section.

The function `summary` (i.e., `summary.glm`) can be used to obtain or print a summary of the results and the function `anova` (i.e., `anova.glm`) to produce an analysis of variance table.

The generic accessor functions `coefficients`, `effects`, `fitted.values` and `residuals` can be used to extract various useful features of the value returned by `glm`.

`weights` extracts a vector of weights, one for each case in the fit (after subsetting and `na.action`).

An object of class `"glm"` is a list containing at least the following components:

`coefficients` a named vector of coefficients

`residuals` the *working* residuals, that is the residuals in the final iteration of the IWLS fit. Since cases with zero weights are omitted, their working residuals are `NA`.

`fitted.values` the fitted mean values, obtained by transforming the linear predictors by the inverse of the link function.

`rank` the numeric rank of the fitted linear model.

`family` the `family` object used.

`linear.predictors` the linear fit on link scale.

`deviance` up to a constant, minus twice the maximized log-likelihood. Where sensible, the constant is chosen so that a saturated model has deviance zero.

`aic` Akaike's *An Information Criterion*, minus twice the maximized log-likelihood plus twice the number of coefficients (so assuming that the dispersion is known).

`null.deviance` The deviance for the null model, comparable with `deviance`. The null model will include the offset, and an intercept if there is one in the model. Note that this will be incorrect if the link function depends on the data other than through the fitted mean: specify a zero offset to force a correct calculation.

`iter` the number of iterations of IWLS used.

`weights` the *working* weights, that is the weights in the final iteration of the IWLS fit.

`prior.weights` the case weights initially supplied.

`df.residual` the residual degrees of freedom.

`df.null` the residual degrees of freedom for the null model.

`y` if requested (the default) the `y` vector used. (It is a vector even for a binomial model.)

`x` (if requested, the model matrix.

`model` (if requested (the default), the model frame.

`converged` logical. Was the IWLS algorithm judged to have converged?

`boundary` logical. Is the fitted value on the boundary of the attainable values?

`call` the matched call.

`formula` the formula supplied.

`terms` the `terms` object used.

data	the data argument.
offset	the offset vector used.
control	the value of the control argument used.
method	the name of the fitter function used, currently always "glm.fit".
contrasts	(where relevant) the contrasts used.
xlevels	(where relevant) a record of the levels of the factors used in fitting.
na.action	(where relevant) information returned by model.frame on the special handling of NAs.

In addition, non-empty fits will have components qr, R and effects relating to the final weighted linear fit.

Objects of class "glm" are normally of class c("glm", "lm"), that is inherit from class "lm", and well-designed methods for class "lm" will be applied to the weighted linear model at the final iteration of IWLS. However, care is needed, as extractor functions for class "glm" such as residuals and weights do **not** just pick out the component of the fit with the same name.

If a binomial glm model was specified by giving a two-column response, the weights returned by prior.weights are the total numbers of cases (factored by the supplied case weights) and the component y of the result is the proportion of successes.

Author(s)

The original R implementation of glm was written by Simon Davies working for Ross Ihaka at the University of Auckland, but has since been extensively re-written by members of the R Core team.

The design was inspired by the S function of the same name described in Hastie & Pregibon (1992).

References

Dobson, A. J. (1990) *An Introduction to Generalized Linear Models.* London: Chapman and Hall.

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* New York: Springer.

See Also

anova.glm, summary.glm, etc. for glm methods, and the generic functions anova, summary, effects, fitted.values, and residuals.

lm for non-generalized *linear* models (which SAS calls GLMs, for 'general' linear models).

loglin and loglm for fitting log-linear models (which binomial and Poisson GLMs are) to contingency tables.

bigglm in package **biglm** for an alternative way to fit GLMs to large datasets (especially those with many cases).

esoph, infert and predict.glm have examples of fitting binomial glms.

Examples

```
## Dobson (1990) Page 93: Randomized Controlled Trial :
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
print(d.AD <- data.frame(treatment, outcome, counts))
glm.D93 <- glm(counts ~ outcome + treatment, family=poisson())
anova(glm.D93)
summary(glm.D93)

## an example with offsets from Venables & Ripley (2002, p.189)
utils::data(anorexia, package="MASS")

anorex.1 <- glm(Postwt ~ Prewt + Treat + offset(Prewt),
                family = gaussian, data = anorexia)
summary(anorex.1)

# A Gamma example, from McCullagh & Nelder (1989, pp. 300-2)
clotting <- data.frame(
    u = c(5,10,15,20,30,40,60,80,100),
    lot1 = c(118,58,42,35,27,25,21,19,18),
    lot2 = c(69,35,26,21,18,16,13,12,12))
summary(glm(lot1 ~ log(u), data=clotting, family=Gamma))
summary(glm(lot2 ~ log(u), data=clotting, family=Gamma))

## Not run:
## for an example of the use of a terms object as a formula
demo(glm.vr)
## End(Not run)
```

---

glm.control *Auxiliary for Controlling GLM Fitting*

Description

Auxiliary function as user interface for `glm` fitting. Typically only used when calling `glm` or `glm.fit`.

Usage

```
glm.control(epsilon = 1e-8, maxit = 25, trace = FALSE)
```

Arguments

`epsilon`	positive convergence tolerance ϵ; the iterations converge when $\|dev - dev_{old}\|/(\|dev\| + 0.1) < \epsilon$.
`maxit`	integer giving the maximal number of IWLS iterations.
`trace`	logical indicating if output should be produced for each iteration.

Details

If `epsilon` is small, it is also used as the tolerance for the least squares solution.

When `trace` is true, calls to `cat` produce the output for each IWLS iteration. Hence, `options(digits = *)` can be used to increase the precision, see the example.

Value

A list with the arguments as components.

References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`glm.fit`, the fitting procedure used by `glm`.

Examples

```
### A variation on  example(glm) :

## Annette Dobson's example ...
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
oo <- options(digits = 12) # to see more when tracing :
glm.D93X <- glm(counts ~ outcome + treatment, family=poisson(),
                trace = TRUE, epsilon = 1e-14)
options(oo)
coef(glm.D93X) # the last two are closer to 0 than in ?glm's  glm.D93
```

`glm.summaries` *Accessing Generalized Linear Model Fits*

Description

These functions are all `methods` for class `glm` or `summary.glm` objects.

Usage

```
## S3 method for class 'glm':
family(object, ...)

## S3 method for class 'glm':
residuals(object, type = c("deviance", "pearson", "working",
                           "response", "partial"), ...)
```

Arguments

`object`	an object of class `glm`, typically the result of a call to `glm`.
`type`	the type of residuals which should be returned. The alternatives are: `"deviance"` (default), `"pearson"`, `"working"`, `"response"`, and `"partial"`.
`...`	further arguments passed to or from other methods.

Details

The references define the types of residuals: Davison & Snell is a good reference for the usages of each.

The partial residuals are a matrix of working residuals, with each column formed by omitting a term from the model.

How `residuals` treats cases with missing values in the original fit is determined by the `na.action` argument of that fit. If `na.action = na.omit` omitted cases will not appear in the residuals, whereas if `na.action = na.exclude` they will appear, with residual value `NA`. See also `naresid`.

For fits done with `y = FALSE` the response values are computed from other components.

References

Davison, A. C. and Snell, E. J. (1991) *Residuals and diagnostics.* In: Statistical Theory and Modelling. In Honour of Sir David Cox, FRS, eds. Hinkley, D. V., Reid, N. and Snell, E. J., Chapman & Hall.

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

See Also

`glm` for computing `glm.obj`, `anova.glm`; the corresponding *generic* functions, `summary.glm`, `coef`, `deviance`, `df.residual`, `effects`, `fitted`, `residuals`.

influence.measures for deletion diagnostics, including standardized (`rstandard`) and studentized (`rstudent`) residuals.

`hclust` *Hierarchical Clustering*

Description

Hierarchical cluster analysis on a set of dissimilarities and methods for analyzing it.

Usage

```
hclust(d, method = "complete", members=NULL)

## S3 method for class 'hclust':
plot(x, labels = NULL, hang = 0.1,
     axes = TRUE, frame.plot = FALSE, ann = TRUE,
     main = "Cluster Dendrogram",
     sub = NULL, xlab = NULL, ylab = "Height", ...)

plclust(tree, hang = 0.1, unit = FALSE, level = FALSE, hmin = 0,
        square = TRUE, labels = NULL, plot. = TRUE,
        axes = TRUE, frame.plot = FALSE, ann = TRUE,
        main = "", sub = NULL, xlab = NULL, ylab = "Height")
```

Arguments

`d`	a dissimilarity structure as produced by `dist`.
`method`	the agglomeration method to be used. This should be (an unambiguous abbreviation of) one of `"ward"`, `"single"`, `"complete"`, `"average"`, `"mcquitty"`, `"median"` or `"centroid"`.
`members`	`NULL` or a vector with length size of `d`. See the 'Details' section.
`x,tree`	an object of the type produced by `hclust`.
`hang`	The fraction of the plot height by which labels should hang below the rest of the plot. A negative value will cause the labels to hang down from 0.
`labels`	A character vector of labels for the leaves of the tree. By default the row names or row numbers of the original data are used. If `labels=FALSE` no labels at all are plotted.
`axes, frame.plot, ann`	logical flags as in `plot.default`.
`main, sub, xlab, ylab`	character strings for `title`. `sub` and `xlab` have a non-NULL default when there's a `tree$call`.
`...`	Further graphical arguments.
`unit`	logical. If true, the splits are plotted at equally-spaced heights rather than at the height in the object.
`hmin`	numeric. All heights less than `hmin` are regarded as being `hmin`: this can be used to suppress detail at the bottom of the tree.
`level, square, plot.`	as yet unimplemented arguments of `plclust` for S-PLUS compatibility.

Details

This function performs a hierarchical cluster analysis using a set of dissimilarities for the n objects being clustered. Initially, each object is assigned to its own cluster and then the algorithm proceeds iteratively, at each stage joining the two most similar clusters, continuing until there is just a single cluster. At each stage distances between clusters are recomputed by the Lance–Williams dissimilarity update formula according to the particular clustering method being used.

A number of different clustering methods are provided. *Ward's* minimum variance method aims at finding compact, spherical clusters. The *complete linkage* method finds similar clusters. The *single linkage* method (which is closely related to the minimal spanning tree) adopts a 'friends of friends' clustering strategy. The other methods can be regarded as aiming for clusters with characteristics somewhere between the single and complete link methods. Note however, that methods `"median"` and `"centroid"` are *not* leading to a *monotone distance* measure, or equivalently the resulting dendrograms can have so called *inversions* (which are hard to interpret).

If `members!=NULL`, then `d` is taken to be a dissimilarity matrix between clusters instead of dissimilarities between singletons and `members` gives the number of observations per cluster. This way the hierarchical cluster algorithm can be 'started in the middle of the dendrogram', e.g., in order to reconstruct the part of the tree above a cut (see examples). Dissimilarities between clusters can be efficiently computed (i.e., without `hclust` itself) only for a limited number of distance/linkage combinations, the simplest one being squared Euclidean distance and centroid linkage. In this case the dissimilarities between the clusters are the squared Euclidean distances between cluster means.

In hierarchical cluster displays, a decision is needed at each merge to specify which subtree should go on the left and which on the right. Since, for n observations there are $n - 1$ merges, there are

$2^{(n-1)}$ possible orderings for the leaves in a cluster tree, or dendrogram. The algorithm used in `hclust` is to order the subtree so that the tighter cluster is on the left (the last, i.e., most recent, merge of the left subtree is at a lower value than the last merge of the right subtree). Single observations are the tightest clusters possible, and merges involving two observations place them in order by their observation sequence number.

Value

An object of class **hclust** which describes the tree produced by the clustering process. The object is a list with components:

`merge`	an $n-1$ by 2 matrix. Row i of `merge` describes the merging of clusters at step i of the clustering. If an element j in the row is negative, then observation $-j$ was merged at this stage. If j is positive then the merge was with the cluster formed at the (earlier) stage j of the algorithm. Thus negative entries in `merge` indicate agglomerations of singletons, and positive entries indicate agglomerations of non-singletons.
`height`	a set of $n-1$ non-decreasing real values. The clustering *height*: that is, the value of the criterion associated with the clustering `method` for the particular agglomeration.
`order`	a vector giving the permutation of the original observations suitable for plotting, in the sense that a cluster plot using this ordering and matrix `merge` will not have crossings of the branches.
`labels`	labels for each of the objects being clustered.
`call`	the call which produced the result.
`method`	the cluster method that has been used.
`dist.method`	the distance that has been used to create `d` (only returned if the distance object has a `"method"` attribute).

There are `print`, `plot` and `identify` (see `identify.hclust`) methods and the `rect.hclust()` function for `hclust` objects. The `plclust()` function is basically the same as the plot method, `plot.hclust`, primarily for back compatibility with S-plus. Its extra arguments are not yet implemented.

Author(s)

The `hclust` function is based on Fortran code contributed to STATLIB by F. Murtagh.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (S version.)

Everitt, B. (1974). *Cluster Analysis*. London: Heinemann Educ. Books.

Hartigan, J. A. (1975). *Clustering Algorithms*. New York: Wiley.

Sneath, P. H. A. and R. R. Sokal (1973). *Numerical Taxonomy*. San Francisco: Freeman.

Anderberg, M. R. (1973). *Cluster Analysis for Applications*. Academic Press: New York.

Gordon, A. D. (1999). *Classification*. Second Edition. London: Chapman and Hall / CRC

Murtagh, F. (1985). "Multidimensional Clustering Algorithms", in *COMPSTAT Lectures 4*. Wuerzburg: Physica-Verlag (for algorithmic details of algorithms used).

McQuitty, L.L. (1966). Similarity Analysis by Reciprocal Pairs for Discrete and Continuous Data. *Educational and Psychological Measurement*, **26**, 825–831.

See Also

identify.hclust, rect.hclust, cutree, dendrogram, kmeans.

For the Lance–Williams formula and methods that apply it generally, see agnes from package **cluster**.

Examples

```
require(graphics)

hc <- hclust(dist(USArrests), "ave")
plot(hc)
plot(hc, hang = -1)

## Do the same with centroid clustering and squared Euclidean distance,
## cut the tree into ten clusters and reconstruct the upper part of the
## tree from the cluster centers.
hc <- hclust(dist(USArrests)^2, "cen")
memb <- cutree(hc, k = 10)
cent <- NULL
for(k in 1:10){
  cent <- rbind(cent, colMeans(USArrests[memb == k, , drop = FALSE]))
}
hc1 <- hclust(dist(cent)^2, method = "cen", members = table(memb))
opar <- par(mfrow = c(1, 2))
plot(hc,  labels = FALSE, hang = -1, main = "Original Tree")
plot(hc1, labels = FALSE, hang = -1, main = "Re-start from 10 clusters")
par(opar)
```

---

heatmap *Draw a Heat Map*

Description

A heat map is a false color image (basically image(t(x))) with a dendrogram added to the left side and to the top. Typically, reordering of the rows and columns according to some set of values (row or column means) within the restrictions imposed by the dendrogram is carried out.

Usage

```
heatmap(x, Rowv=NULL, Colv=if(symm)"Rowv" else NULL,
        distfun = dist, hclustfun = hclust,
        reorderfun = function(d,w) reorder(d,w),
        add.expr, symm = FALSE, revC = identical(Colv, "Rowv"),
        scale=c("row", "column", "none"), na.rm = TRUE,
        margins = c(5, 5), ColSideColors, RowSideColors,
        cexRow = 0.2 + 1/log10(nr), cexCol = 0.2 + 1/log10(nc),
        labRow = NULL, labCol = NULL, main = NULL,
        xlab = NULL, ylab = NULL,
        keep.dendro = FALSE, verbose = getOption("verbose"), ...)
```

Arguments

x	numeric matrix of the values to be plotted.
Rowv	determines if and how the *row* dendrogram should be computed and reordered. Either a dendrogram or a vector of values used to reorder the row dendrogram or NA to suppress any row dendrogram (and reordering) or by default, NULL, see 'Details' below.
Colv	determines if and how the *column* dendrogram should be reordered. Has the same options as the Rowv argument above and *additionally* when x is a square matrix, Colv = "Rowv" means that columns should be treated identically to the rows.
distfun	function used to compute the distance (dissimilarity) between both rows and columns. Defaults to dist.
hclustfun	function used to compute the hierarchical clustering when Rowv or Colv are not dendrograms. Defaults to hclust. Should take as argument a result of distfun and return an object to which as.dendrogram can be applied.
reorderfun	function(d,w) of dendrogram and weights for reordering the row and column dendrograms. The default uses reorder.dendrogram.
add.expr	expression that will be evaluated after the call to image. Can be used to add components to the plot.
symm	logical indicating if x should be treated **symm**etrically; can only be true when x is a square matrix.
revC	logical indicating if the column order should be reversed for plotting, such that e.g., for the symmetric case, the symmetry axis is as usual.
scale	character indicating if the values should be centered and scaled in either the row direction or the column direction, or none. The default is "row" if symm false, and "none" otherwise.
na.rm	logical indicating whether NA's should be removed.
margins	numeric vector of length 2 containing the margins (see par(mar= *)) for column and row names, respectively.
ColSideColors	(optional) character vector of length ncol(x) containing the color names for a horizontal side bar that may be used to annotate the columns of x.
RowSideColors	(optional) character vector of length nrow(x) containing the color names for a vertical side bar that may be used to annotate the rows of x.
cexRow, cexCol	positive numbers, used as cex.axis in for the row or column axis labeling. The defaults currently only use number of rows or columns, respectively.
labRow, labCol	character vectors with row and column labels to use; these default to rownames(x) or colnames(x), respectively.
main, xlab, ylab	main, x- and y-axis titles; defaults to none.
keep.dendro	logical indicating if the dendrogram(s) should be kept as part of the result (when Rowv and/or Colv are not NA).
verbose	logical indicating if information should be printed.
...	additional arguments passed on to image, e.g., col specifying the colors.

Details

If either `Rowv` or `Colv` are dendrograms they are honored (and not reordered). Otherwise, dendrograms are computed as `dd <- as.dendrogram(hclustfun(distfun(X)))` where `X` is either `x` or `t(x)`.

If either is a vector (of 'weights') then the appropriate dendrogram is reordered according to the supplied values subject to the constraints imposed by the dendrogram, by `reorder(dd, Rowv)`, in the row case. If either is missing, as by default, then the ordering of the corresponding dendrogram is by the mean value of the rows/columns, i.e., in the case of rows, `Rowv <- rowMeans(x, na.rm=na.rm)`. If either is `NULL`, *no reordering* will be done for the corresponding side.

By default (`scale = "row"`) the rows are scaled to have mean zero and standard deviation one. There is some empirical evidence from genomic plotting that this is useful.

The default colors are not pretty. Consider using enhancements such as the **RColorBrewer** package, `http://cran.r-project.org/package=RColorBrewer`.

Value

Invisibly, a list with components

`rowInd`	**r**ow index permutation vector as returned by `order.dendrogram`.
`colInd`	**c**olumn index permutation vector.
`Rowv`	the row dendrogram; only if input `Rowv` was not NA and `keep.dendro` is true.
`Colv`	the column dendrogram; only if input `Colv` was not NA and `keep.dendro` is true.

Note

Unless `Rowv = NA` (or `Colw = NA`), the original rows and columns are reordered *in any case* to match the dendrogram, e.g., the rows by `order.dendrogram(Rowv)` where `Rowv` is the (possibly `reorder()`ed) row dendrogram.

`heatmap()` uses `layout` and draws the `image` in the lower right corner of a 2x2 layout. Consequentially, it can **not** be used in a multi column/row layout, i.e., when `par(mfrow= *)` or `(mfcol= *)` has been called.

Author(s)

Andy Liaw, original; R. Gentleman, M. Maechler, W. Huber, revisions.

See Also

`image`, `hclust`

Examples

```
require(graphics); require(grDevices)
x  <- as.matrix(mtcars)
rc <- rainbow(nrow(x), start=0, end=.3)
cc <- rainbow(ncol(x), start=0, end=.3)
hv <- heatmap(x, col = cm.colors(256), scale="column",
              RowSideColors = rc, ColSideColors = cc, margins=c(5,10),
              xlab = "specification variables", ylab= "Car Models",
              main = "heatmap(<Mtcars data>, ..., scale = \"column\")")
```

```
utils::str(hv) # the two re-ordering index vectors

## no column dendrogram (nor reordering) at all:
heatmap(x, Colv = NA, col = cm.colors(256), scale="column",
        RowSideColors = rc, margins=c(5,10),
        xlab = "specification variables", ylab= "Car Models",
        main = "heatmap(<Mtcars data>, ..., scale = \"column\")")

## "no nothing"
heatmap(x, Rowv = NA, Colv = NA, scale="column",
        main = "heatmap(*, NA, NA) ~= image(t(x))")

round(Ca <- cor(attitude), 2)
symnum(Ca) # simple graphic
heatmap(Ca,             symm = TRUE, margins=c(6,6))# with reorder()
heatmap(Ca, Rowv=FALSE, symm = TRUE, margins=c(6,6))# _NO_ reorder()

## For variable clustering, rather use distance based on cor():
symnum( cU <- cor(USJudgeRatings) )

hU <- heatmap(cU, Rowv = FALSE, symm = TRUE, col = topo.colors(16),
              distfun = function(c) as.dist(1 - c), keep.dendro = TRUE)
## The Correlation matrix with same reordering:
round(100 * cU[hU[[1]], hU[[2]]])
## The column dendrogram:
utils::str(hU$Colv)
```

`HoltWinters` *Holt-Winters Filtering*

Description

Computes Holt-Winters Filtering of a given time series. Unknown parameters are determined by minimizing the squared prediction error.

Usage

```
HoltWinters(x, alpha = NULL, beta = NULL, gamma = NULL,
            seasonal = c("additive", "multiplicative"),
            start.periods = 2, l.start = NULL, b.start = NULL,
            s.start = NULL,
            optim.start = c(alpha = 0.3, beta = 0.1, gamma = 0.1),
            optim.control = list())
```

Arguments

`x`	An object of class `ts`
`alpha`	$alpha$ parameter of Holt-Winters Filter.
`beta`	$beta$ parameter of Holt-Winters Filter. If set to 0, the function will do exponential smoothing.
`gamma`	$gamma$ parameter used for the seasonal component. If set to 0, an non-seasonal model is fitted.

`seasonal` Character string to select an `"additive"` (the default) or `"multiplicative"` seasonal model. The first few characters are sufficient. (Only takes effect if `gamma` is non-zero).

`start.periods` Start periods used in the autodetection of start values. Must be at least 2.

`l.start` Start value for level (a[0]).

`b.start` Start value for trend (b[0]).

`s.start` Vector of start values for the seasonal component ($s_1[0]...s_p[0]$)

`optim.start` Vector with named components `alpha`, `beta`, and `gamma` containing the starting values for the optimizer. Only the values needed must be specified. Ignored in the one-parameter case.

`optim.control` Optional list with additional control parameters passed to `optim` if this is used. Ignored in the one-parameter case.

Details

The additive Holt-Winters prediction function (for time series with period length p) is

$$\hat{Y}[t+h] = a[t] + hb[t] + s[t+1+(h-1) \bmod p],$$

where $a[t]$, $b[t]$ and $s[t]$ are given by

$$a[t] = \alpha(Y[t] - s[t-p]) + (1-\alpha)(a[t-1] + b[t-1])$$

$$b[t] = \beta(a[t] - a[t-1]) + (1-\beta)b[t-1]$$

$$s[t] = \gamma(Y[t] - a[t]) + (1-\gamma)s[t-p]$$

The multiplicative Holt-Winters prediction function (for time series with period length p) is

$$\hat{Y}[t+h] = (a[t] + hb[t]) \times s[t+1+(h-1) \bmod p].$$

where $a[t]$, $b[t]$ and $s[t]$ are given by

$$a[t] = \alpha(Y[t]/s[t-p]) + (1-\alpha)(a[t-1] + b[t-1])$$

$$b[t] = \beta(a[t] - a[t-1]) + (1-\beta)b[t-1]$$

$$s[t] = \gamma(Y[t]/a[t]) + (1-\gamma)s[t-p]$$

The function tries to find the optimal values of α and/or β and/or γ by minimizing the squared one-step prediction error if they are omitted. `optimize` will be used for the univariate case, and `optim` else.

For seasonal models, start values for `a`, `b` and `s` are detected by performing a simple decomposition in trend and seasonal component using moving averages (see function `decompose`) on the `start.periods` first periods (a simple linear regression on the trend component is used for starting level and trend.). For level/trend-models (no seasonal component), start values for a and b are x[2] and x[2] - x[1], respectively. For level-only models (ordinary exponential smoothing), the start value for a is x[1].

Value

An object of class `"HoltWinters"`, a list with components:

`fitted`	A multiple time series with one column for the filtered series as well as for the level, trend and seasonal components, estimated contemporaneously (that is at time t and not at the end of the series).
`x`	The original series
`alpha`	alpha used for filtering
`beta`	beta used for filtering
`gamma`	gamma used for filtering
`coefficients`	A vector with named components `a, b, s1, ..., sp` containing the estimated values for the level, trend and seasonal components
`seasonal`	The specified `seasonal`-parameter
`SSE`	The final sum of squared errors achieved in optimizing
`call`	The call used

Author(s)

David Meyer ⟨David.Meyer@wu-wien.ac.at⟩

References

C. C. Holt (1957) Forecasting seasonals and trends by exponentially weighted moving averages, *ONR Research Memorandum, Carnigie Institute* **52**.

P. R. Winters (1960) Forecasting sales by exponentially weighted moving averages, *Management Science* **6**, 324–342.

See Also

`predict.HoltWinters`,`optim`

Examples

```
require(graphics)

## Seasonal Holt-Winters
(m <- HoltWinters(co2))
plot(m)
plot(fitted(m))

(m <- HoltWinters(AirPassengers, seasonal = "mult"))
plot(m)

## Non-Seasonal Holt-Winters
x <- uspop + rnorm(uspop, sd = 5)
m <- HoltWinters(x, gamma = 0)
plot(m)

## Exponential Smoothing
m2 <- HoltWinters(x, gamma = 0, beta = 0)
lines(fitted(m2)[,1], col = 3)
```

`Hypergeometric` *The Hypergeometric Distribution*

Description

Density, distribution function, quantile function and random generation for the hypergeometric distribution.

Usage

```
dhyper(x, m, n, k, log = FALSE)
phyper(q, m, n, k, lower.tail = TRUE, log.p = FALSE)
qhyper(p, m, n, k, lower.tail = TRUE, log.p = FALSE)
rhyper(nn, m, n, k)
```

Arguments

`x, q`	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls.
`m`	the number of white balls in the urn.
`n`	the number of black balls in the urn.
`k`	the number of balls drawn from the urn.
`p`	probability, it must be between 0 and 1.
`nn`	number of observations. If `length(nn) > 1`, the length is taken to be the number required.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The hypergeometric distribution is used for sampling *without* replacement. The density of this distribution with parameters `m`, `n` and `k` (named Np, $N - Np$, and n, respectively in the reference below) is given by

$$p(x) = \binom{m}{x}\binom{n}{k-x} \Big/ \binom{m+n}{k}$$

for $x = 0, \ldots, k$.

Value

`dhyper` gives the density, `phyper` gives the distribution function, `qhyper` gives the quantile function, and `rhyper` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Source

dhyper computes via binomial probabilities, using code contributed by Catherine Loader (see dbinom).

phyper is based on calculating dhyper and phyper(...)/dhyper(...) (as a summation), based on ideas of Ian Smith and Morten Welinder.

qhyper is based on inversion.

rhyper is based on a corrected version of

Kachitvichyanukul, V. and Schmeiser, B. (1985). Computer generation of hypergeometric random variates. *Journal of Statistical Computation and Simulation*, **22**, 127–145.

References

Johnson, N. L., Kotz, S., and Kemp, A. W. (1992) *Univariate Discrete Distributions*, Second Edition. New York: Wiley.

Examples

```
m <- 10; n <- 7; k <- 8
x <- 0:(k+1)
rbind(phyper(x, m, n, k), dhyper(x, m, n, k))
all(phyper(x, m, n, k) == cumsum(dhyper(x, m, n, k)))# FALSE
## but error is very small:
signif(phyper(x, m, n, k) - cumsum(dhyper(x, m, n, k)), digits=3)
```

---

identify.hclust *Identify Clusters in a Dendrogram*

Description

identify.hclust reads the position of the graphics pointer when the (first) mouse button is pressed. It then cuts the tree at the vertical position of the pointer and highlights the cluster containing the horizontal position of the pointer. Optionally a function is applied to the index of data points contained in the cluster.

Usage

```
## S3 method for class 'hclust':
identify(x, FUN = NULL, N = 20, MAXCLUSTER = 20, DEV.FUN = NULL,
         ...)
```

Arguments

x	an object of the type produced by hclust.
FUN	(optional) function to be applied to the index numbers of the data points in a cluster (see 'Details' below).
N	the maximum number of clusters to be identified.
MAXCLUSTER	the maximum number of clusters that can be produced by a cut (limits the effective vertical range of the pointer).
DEV.FUN	(optional) integer scalar. If specified, the corresponding graphics device is made active before FUN is applied.
...	further arguments to FUN.

Details

By default clusters can be identified using the mouse and an `invisible` list of indices of the respective data points is returned.

If `FUN` is not `NULL`, then the index vector of data points is passed to this function as first argument, see the examples below. The active graphics device for `FUN` can be specified using `DEV.FUN`.

The identification process is terminated by pressing any mouse button other than the first, see also `identify`.

Value

Either a list of data point index vectors or a list of return values of `FUN`.

See Also

`hclust`, `rect.hclust`

Examples

```
## Not run:
require(graphics)

hca <- hclust(dist(USArrests))
plot(hca)
(x <- identify(hca)) ##  Terminate with 2nd mouse button !!

hci <- hclust(dist(iris[,1:4]))
plot(hci)
identify(hci, function(k) print(table(iris[k,5])))

# open a new device (one for dendrogram, one for bars):
get(getOption("device"))() # << make that narrow (& small)
                           # and *beside* 1st one
nD <- dev.cur()            # to be for the barplot
dev.set(dev.prev())# old one for dendrogram
plot(hci)
## select subtrees in dendrogram and "see" the species distribution:
identify(hci, function(k) barplot(table(iris[k,5]),col=2:4), DEV.FUN = nD)
## End(Not run)
```

`influence.measures` *Regression Deletion Diagnostics*

Description

This suite of functions can be used to compute some of the regression (leave-one-out deletion) diagnostics for linear and generalized linear models discussed in Belsley, Kuh and Welsch (1980), Cook and Weisberg (1982), etc.

Usage

```
influence.measures(model)

rstandard(model, ...)
## S3 method for class 'lm':
rstandard(model, infl = lm.influence(model, do.coef = FALSE),
          sd = sqrt(deviance(model)/df.residual(model)), ...)
## S3 method for class 'glm':
rstandard(model, infl = lm.influence(model, do.coef = FALSE),
          ...)

rstudent(model, ...)
## S3 method for class 'lm':
rstudent(model, infl = lm.influence(model, do.coef = FALSE),
         res = infl$wt.res, ...)
## S3 method for class 'glm':
rstudent(model, infl = influence(model, do.coef = FALSE), ...)

dffits(model, infl = , res = )

dfbeta(model, ...)
## S3 method for class 'lm':
dfbeta(model, infl = lm.influence(model, do.coef = TRUE), ...)

dfbetas(model, ...)
## S3 method for class 'lm':
dfbetas(model, infl = lm.influence(model, do.coef = TRUE), ...)

covratio(model, infl = lm.influence(model, do.coef = FALSE),
         res = weighted.residuals(model))

cooks.distance(model, ...)
## S3 method for class 'lm':
cooks.distance(model, infl = lm.influence(model, do.coef = FALSE),
               res = weighted.residuals(model),
               sd = sqrt(deviance(model)/df.residual(model)),
               hat = infl$hat, ...)
## S3 method for class 'glm':
cooks.distance(model, infl = influence(model, do.coef = FALSE),
               res = infl$pear.res,
               dispersion = summary(model)$dispersion,
               hat = infl$hat, ...)

hatvalues(model, ...)
## S3 method for class 'lm':
hatvalues(model, infl = lm.influence(model, do.coef = FALSE), ...)

hat(x, intercept = TRUE)
```

Arguments

`model`	an R object, typically returned by `lm` or `glm`.

infl	influence structure as returned by lm.influence or influence (the latter only for the glm method of rstudent and cooks.distance).
res	(possibly weighted) residuals, with proper default.
sd	standard deviation to use, see default.
dispersion	dispersion (for glm objects) to use, see default.
hat	hat values H_{ii}, see default.
x	the X or design matrix.
intercept	should an intercept column be prepended to x?
...	further arguments passed to or from other methods.

Details

The primary high-level function is influence.measures which produces a class "infl" object tabular display showing the DFBETAS for each model variable, DFFITS, covariance ratios, Cook's distances and the diagonal elements of the hat matrix. Cases which are influential with respect to any of these measures are marked with an asterisk.

The functions dfbetas, dffits, covratio and cooks.distance provide direct access to the corresponding diagnostic quantities. Functions rstandard and rstudent give the standardized and Studentized residuals respectively. (These re-normalize the residuals to have unit variance, using an overall and leave-one-out measure of the error variance respectively.)

Values for generalized linear models are approximations, as described in Williams (1987) (except that Cook's distances are scaled as F rather than as chi-square values). The approximations can be poor when some cases have large influence.

The optional infl, res and sd arguments are there to encourage the use of these direct access functions, in situations where, e.g., the underlying basic influence measures (from lm.influence or the generic influence) are already available.

Note that cases with weights == 0 are *dropped* from all these functions, but that if a linear model has been fitted with na.action = na.exclude, suitable values are filled in for the cases excluded during fitting.

The function hat() exists mainly for S (version 2) compatibility; we recommend using hatvalues() instead.

Note

For hatvalues, dfbeta, and dfbetas, the method for linear models also works for generalized linear models.

Author(s)

Several R core team members and John Fox, originally in his 'car' package.

References

Belsley, D. A., Kuh, E. and Welsch, R. E. (1980) *Regression Diagnostics*. New York: Wiley.

Cook, R. D. and Weisberg, S. (1982) *Residuals and Influence in Regression*. London: Chapman and Hall.

Williams, D. A. (1987) Generalized linear model diagnostics using the deviance and single case deletions. *Applied Statistics* **36**, 181–191.

Fox, J. (1997) *Applied Regression, Linear Models, and Related Methods*. Sage.

Fox, J. (2002) *An R and S-Plus Companion to Applied Regression*. Sage Publ.; http://www.socsci.mcmaster.ca/jfox/Books/Companion/.

See Also

`influence` (containing `lm.influence`).

'plotmath' for the use of `hat` in plot annotation.

Examples

```
require(graphics)

## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)

inflm.SR <- influence.measures(lm.SR)
which(apply(inflm.SR$is.inf, 1, any))
# which observations 'are' influential
summary(inflm.SR) # only these
inflm.SR          # all
plot(rstudent(lm.SR) ~ hatvalues(lm.SR)) # recommended by some

## The 'infl' argument is not needed, but avoids recomputation:
rs <- rstandard(lm.SR)
iflSR <- influence(lm.SR)
identical(rs, rstandard(lm.SR, infl = iflSR))
## to "see" the larger values:
1000 * round(dfbetas(lm.SR, infl = iflSR), 3)

## Huber's data [Atkinson 1985]
xh <- c(-4:0, 10)
yh <- c(2.48, .73, -.04, -1.44, -1.32, 0)
summary(lmH <- lm(yh ~ xh))
(im <- influence.measures(lmH))
plot(xh,yh, main = "Huber's data: L.S. line and influential obs.")
abline(lmH); points(xh[im$is.inf], yh[im$is.inf], pch=20, col=2)

## Irwin's data [Williams 1987]
xi <- 1:5
yi <- c(0,2,14,19,30) # number of mice responding to does xi
mi <- rep(40, 5)      # number of mice exposed
summary(lmI <- glm(cbind(yi, mi -yi) ~ xi, family = binomial))
signif(cooks.distance(lmI), 3)# ~= Ci in Table 3, p.184
(imI <- influence.measures(lmI))
stopifnot(all.equal(imI$infmat[,"cook.d"],
          cooks.distance(lmI)))
```

`integrate` *Integration of One-Dimensional Functions*

Description

Adaptive quadrature of functions of one variable over a finite or infinite interval.

Usage

```
integrate(f, lower, upper, ..., subdivisions=100,
          rel.tol = .Machine$double.eps^0.25, abs.tol = rel.tol,
          stop.on.error = TRUE, keep.xy = FALSE, aux = NULL)
```

Arguments

`f`	an R function taking a numeric first argument and returning a numeric vector of the same length. Returning a non-finite element will generate an error.
`lower, upper`	the limits of integration. Can be infinite.
`...`	additional arguments to be passed to `f`.
`subdivisions`	the maximum number of subintervals.
`rel.tol`	relative accuracy requested.
`abs.tol`	absolute accuracy requested.
`stop.on.error`	logical. If true (the default) an error stops the function. If false some errors will give a result with a warning in the `message` component.
`keep.xy`	unused. For compatibility with S.
`aux`	unused. For compatibility with S.

Details

Note that arguments after `...` must be matched exactly.

If one or both limits are infinite, the infinite range is mapped onto a finite interval.

For a finite interval, globally adaptive interval subdivision is used in connection with extrapolation by the Epsilon algorithm.

`rel.tol` cannot be less than `max(50*.Machine$double.eps, 0.5e-28)` if `abs.tol <= 0`.

Value

A list of class `"integrate"` with components

`value`	the final estimate of the integral.
`abs.error`	estimate of the modulus of the absolute error.
`subdivisions`	the number of subintervals produced in the subdivision process.
`message`	`"OK"` or a character string giving the error message.
`call`	the matched call.

Note

Like all numerical integration routines, these evaluate the function on a finite set of points. If the function is approximately constant (in particular, zero) over nearly all its range it is possible that the result and error estimate may be seriously wrong.

When integrating over infinite intervals do so explicitly, rather than just using a large number as the endpoint. This increases the chance of a correct answer – any function whose integral over an infinite interval is finite must be near zero for most of that interval.

`f` must accept a vector of inputs and produce a vector of function evaluations at those points. The `Vectorize` function may be helpful to convert `f` to this form.

References

Based on QUADPACK routines `dqags` and `dqagi` by R. Piessens and E. deDoncker-Kapenga, available from Netlib.

See
R. Piessens, E. deDoncker-Kapenga, C. Uberhuber, D. Kahaner (1983) *Quadpack: a Subroutine Package for Automatic Integration*; Springer Verlag.

See Also

The function `adapt` in the **adapt** package on CRAN, for multivariate integration.

Examples

```
integrate(dnorm, -1.96, 1.96)
integrate(dnorm, -Inf, Inf)

## a slowly-convergent integral
integrand <- function(x) {1/((x+1)*sqrt(x))}
integrate(integrand, lower = 0, upper = Inf)

## don't do this if you really want the integral from 0 to Inf
integrate(integrand, lower = 0, upper = 10)
integrate(integrand, lower = 0, upper = 100000)
integrate(integrand, lower = 0, upper = 1000000, stop.on.error = FALSE)

## some functions do not handle vector input properly
f <- function(x) 2.0
try(integrate(f, 0, 1))
integrate(Vectorize(f), 0, 1)  ## correct
integrate(function(x) rep(2.0, length(x)), 0, 1)  ## correct

## integrate can fail if misused
integrate(dnorm,0,2)
integrate(dnorm,0,20)
integrate(dnorm,0,200)
integrate(dnorm,0,2000)
integrate(dnorm,0,20000) ## fails on many systems
integrate(dnorm,0,Inf)   ## works
```

---

interaction.plot *Two-way Interaction Plot*

Description

Plots the mean (or other summary) of the response for two-way combinations of factors, thereby illustrating possible interactions.

Usage

```
interaction.plot(x.factor, trace.factor, response, fun = mean,
                 type = c("l", "p", "b"), legend = TRUE,
                 trace.label = deparse(substitute(trace.factor)),
                 fixed = FALSE,
```

```
                xlab = deparse(substitute(x.factor)),
                ylab = ylabel,
                ylim = range(cells, na.rm=TRUE),
                lty = nc:1, col = 1, pch = c(1:9, 0, letters),
                xpd = NULL, leg.bg = par("bg"), leg.bty = "n",
                xtick = FALSE, xaxt = par("xaxt"), axes = TRUE,
                ...)
```

Arguments

`x.factor`	a factor whose levels will form the x axis.
`trace.factor`	another factor whose levels will form the traces.
`response`	a numeric variable giving the response
`fun`	the function to compute the summary. Should return a single real value.
`type`	the type of plot: lines or points.
`legend`	logical. Should a legend be included?
`trace.label`	overall label for the legend.
`fixed`	logical. Should the legend be in the order of the levels of `trace.factor` or in the order of the traces at their right-hand ends?
`xlab,ylab`	the x and y label of the plot each with a sensible default.
`ylim`	numeric of length 2 giving the y limits for the plot.
`lty`	line type for the lines drawn, with sensible default.
`col`	the color to be used for plotting.
`pch`	a vector of plotting symbols or characters, with sensible default.
`xpd`	determines clipping behaviour for the `legend` used, see `par(xpd)`. Per default, the legend is *not* clipped at the figure border.
`leg.bg, leg.bty`	arguments passed to `legend()`.
`xtick`	logical. Should tick marks be used on the x axis?
`xaxt, axes, ...`	graphics parameters to be passed to the plotting routines.

Details

By default the levels of `x.factor` are plotted on the x axis in their given order, with extra space left at the right for the legend (if specified). If `x.factor` is an ordered factor and the levels are numeric, these numeric values are used for the x axis.

The response and hence its summary can contain missing values. If so, the missing values and the line segments joining them are omitted from the plot (and this can be somewhat disconcerting).

The graphics parameters `xlab`, `ylab`, `ylim`, `lty`, `col` and `pch` are given suitable defaults (and `xlim` and `xaxs` are set and cannot be overridden). The defaults are to cycle through the line types, use the foreground colour, and to use the symbols 1:9, 0, and the capital letters to plot the traces.

Note

Some of the argument names and the precise behaviour are chosen for S-compatibility.

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Examples

```
require(graphics)

with(ToothGrowth, {
interaction.plot(dose, supp, len, fixed=TRUE)
dose <- ordered(dose)
interaction.plot(dose, supp, len, fixed=TRUE, col = 2:3, leg.bty = "o")
interaction.plot(dose, supp, len, fixed=TRUE, col = 2:3, type = "p")
})

with(OrchardSprays, {
  interaction.plot(treatment, rowpos, decrease)
  interaction.plot(rowpos, treatment, decrease, cex.axis=0.8)
  ## order the rows by their mean effect
  rowpos <- factor(rowpos,
                   levels = sort.list(tapply(decrease, rowpos, mean)))
  interaction.plot(rowpos, treatment, decrease, col = 2:9, lty = 1)
})

with(esoph, {
  interaction.plot(agegp, alcgp, ncases/ncontrols, main = "'esoph' Data")
  interaction.plot(agegp, tobgp, ncases/ncontrols, trace.label="tobacco",
                   fixed=TRUE, xaxt = "n")
})
## deal with NAs:
esoph[66,] # second to last age group: 65-74
esophNA <- esoph; esophNA$ncases[66] <- NA
with(esophNA, {
  interaction.plot(agegp, alcgp, ncases/ncontrols, col= 2:5)
                                # doesn't show *last* group either
  interaction.plot(agegp, alcgp, ncases/ncontrols, col= 2:5, type = "b")
  ## alternative take non-NA's  {"cheating"}
  interaction.plot(agegp, alcgp, ncases/ncontrols, col= 2:5,
                   fun = function(x) mean(x, na.rm=TRUE),
                   sub = "function(x) mean(x, na.rm=TRUE)")
})
rm(esophNA) # to clear up
```

`IQR` *The Interquartile Range*

Description

computes interquartile range of the `x` values.

Usage

```
IQR(x, na.rm = FALSE)
```

Arguments

`x`	a numeric vector.
`na.rm`	logical. Should missing values be removed?

Details

Note that this function computes the quartiles using the `quantile` function rather than following Tukey's recommendations, i.e., `IQR(x) = quantile(x,3/4) - quantile(x,1/4)`.

For normally $N(m, 1)$ distributed X, the expected value of `IQR(X)` is `2*qnorm(3/4) = 1.3490`, i.e., for a normal-consistent estimate of the standard deviation, use `IQR(x) / 1.349`.

References

Tukey, J. W. (1977). *Exploratory Data Analysis.* Reading: Addison-Wesley.

See Also

`fivenum`, `mad` which is more robust, `range`, `quantile`.

Examples

```
IQR(rivers)
```

`is.empty.model` *Check if a Model is Empty*

Description

R model notation allows models with no intercept and no predictors. These require special handling internally. `is.empty.model()` checks whether an object describes an empty model.

Usage

```
is.empty.model(x)
```

Arguments

`x`	A `terms` object or an object with a `terms` method.

Value

`TRUE` if the model is empty

See Also

`lm`,`glm`

Examples

```
y <- rnorm(20)
is.empty.model(y ~ 0)
is.empty.model(y ~ -1)
is.empty.model(lm(y ~ 0))
```

`isoreg` *Isotonic / Monotone Regression*

Description

Compute the isotonic (monotonely increasing nonparametric) least squares regression which is piecewise constant.

Usage

```
isoreg(x, y = NULL)
```

Arguments

`x, y`	coordinate vectors of the regression points. Alternatively a single plotting structure can be specified: see `xy.coords`.

Details

The algorithm determines the convex minorant $m(x)$ of the *cumulative* data (i.e., `cumsum(y)`) which is piecewise linear and the result is $m'(x)$, a step function with level changes at locations where the convex $m(x)$ touches the cumulative data polygon and changes slope.
`as.stepfun()` returns a `stepfun` object which can be more parsimonious.

Value

`isoreg()` returns an object of class `isoreg` which is basically a list with components

`x`	original (constructed) abscissa values `x`.
`y`	corresponding y values.
`yf`	fitted values corresponding to *ordered* x values.
`yc`	cumulative y values corresponding to *ordered* x values.
`iKnots`	integer vector giving indices where the fitted curve jumps, i.e., where the convex minorant has kinks.
`isOrd`	logical indicating if original x values were ordered increasingly already.
`ord`	`if(!isOrd)`: integer permutation `order(x)` of *original* `x`.
`call`	the `call` to `isoreg()` used.

Note

The code should be improved to accept *weights* additionally and solve the corresponding weighted least squares problem.
'Patches are welcome!'

References

Barlow, R. E., Bartholomew, D. J., Bremner, J. M., and Brunk, H. D. (1972) *Statistical inference under order restrictions*; Wiley, London.

Robertson, T., Wright,F. T. and Dykstra, R. L. (1988) *Order Restricted Statistical Inference*; Wiley, New York.

See Also

the plotting method `plot.isoreg` with more examples; `isoMDS()` from the **MASS** package internally uses isotonic regression.

Examples

```
require(graphics)

(ir <- isoreg(c(1,0,4,3,3,5,4,2,0)))
plot(ir, plot.type = "row")

(ir3 <- isoreg(y3 <- c(1,0,4,3,3,5,4,2, 3)))# last "3", not "0"
(fi3 <- as.stepfun(ir3))
(ir4 <- isoreg(1:10, y4 <- c(5, 9, 1:2, 5:8, 3, 8)))
cat(sprintf("R^2 = %.2f\n",
            1 - sum(residuals(ir4)^2) / ((10-1)*var(y4))))

## If you are interested in the knots alone :
with(ir4, cbind(iKnots, yf[iKnots]))

## Example of unordered x[] with ties:
x <- sample((0:30)/8)
y <- exp(x)
x. <- round(x) # ties!
plot(m <- isoreg(x., y))
stopifnot(all.equal(with(m, yf[iKnots]),
                    as.vector(tapply(y, x., mean))))
```

KalmanLike *Kalman Filtering*

Description

Use Kalman Filtering to find the (Gaussian) log-likelihood, or for forecasting or smoothing.

Usage

```
KalmanLike(y, mod, nit = 0, fast=TRUE)
KalmanRun(y, mod, nit = 0, fast=TRUE)
KalmanSmooth(y, mod, nit = 0)
KalmanForecast(n.ahead = 10, mod, fast=TRUE)
makeARIMA(phi, theta, Delta, kappa = 1e6)
```

Arguments

`y`	a univariate time series.
`mod`	A list describing the state-space model: see 'Details'.
`nit`	The time at which the initialization is computed. `nit = 0` implies that the initialization is for a one-step prediction, so `Pn` should not be computed at the first step.
`n.ahead`	The number of steps ahead for which prediction is required.

`phi, theta`	numeric vectors of length ≥ 0 giving AR and MA parameters.
`Delta`	vector of differencing coefficients, so an ARMA model is fitted to `y[t] - Delta[1]*y[t-1] - ...`
`kappa`	the prior variance (as a multiple of the innovations variance) for the past observations in a differenced model.
`fast`	If `TRUE` the `mod` object may be modified.

Details

These functions work with a general univariate state-space model with state vector `a`, transitions `a <- T a + R e`, $e \sim \mathcal{N}(0, \kappa Q)$ and observation equation `y = Z'a + eta`, $(eta \equiv \eta), \eta \sim \mathcal{N}(0, \kappa h)$. The likelihood is a profile likelihood after estimation of κ.

The model is specified as a list with at least components

`T` the transition matrix

`Z` the observation coefficients

`h` the observation variance

`V` `RQR'`

`a` the current state estimate

`P` the current estimate of the state uncertainty matrix

`Pn` the estimate at time $t - 1$ of the state uncertainty matrix

`KalmanSmooth` is the workhorse function for `tsSmooth`.

`makeARIMA` constructs the state-space model for an ARIMA model.

Value

For `KalmanLike`, a list with components `Lik` (the log-likelihood less some constants) and `s2`, the estimate of of κ.

For `KalmanRun`, a list with components `values`, a vector of length 2 giving the output of `KalmanLike`, `resid` (the residuals) and `states`, the contemporaneous state estimates, a matrix with one row for each time.

For `KalmanSmooth`, a list with two components. Component `smooth` is a `n` by `p` matrix of state estimates based on all the observations, with one row for each time. Component `var` is a `n` by `p` by `p` array of variance matrices.

For `KalmanForecast`, a list with components `pred`, the predictions, and `var`, the unscaled variances of the prediction errors (to be multiplied by `s2`).

For `makeARIMA`, a model list including components for its arguments.

Warning

These functions are designed to be called from other functions which check the validity of the arguments passed, so very little checking is done.

In particular, `KalmanLike` alters the objects passed as the elements `a`, `P` and `Pn` of `mod`, so these should not be shared. Use `fast=FALSE` to prevent this.

References

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press.

See Also

arima, StructTS. tsSmooth.

`kernapply` *Apply Smoothing Kernel*

Description

`kernapply` computes the convolution between an input sequence and a specific kernel.

Usage

```
kernapply(x, ...)

## Default S3 method:
kernapply(x, k, circular = FALSE, ...)
## S3 method for class 'ts':
kernapply(x, k, circular = FALSE, ...)
## S3 method for class 'vector':
kernapply(x, k, circular = FALSE, ...)

## S3 method for class 'tskernel':
kernapply(x, k, ...)
```

Arguments

`x`	an input vector, matrix, time series or kernel to be smoothed.
`k`	smoothing `"tskernel"` object.
`circular`	a logical indicating whether the input sequence to be smoothed is treated as circular, i.e., periodic.
`...`	arguments passed to or from other methods.

Value

A smoothed version of the input sequence.

Author(s)

A. Trapletti

See Also

`kernel`, `convolve`, `filter`, `spectrum`

Examples

```
## see 'kernel' for examples
```

`kernel` *Smoothing Kernel Objects*

Description

The `"tskernel"` class is designed to represent discrete symmetric normalized smoothing kernels. These kernels can be used to smooth vectors, matrices, or time series objects.

There are `print`, `plot` and `[` methods for these kernel objects.

Usage

```
kernel(coef, m, r, name)

df.kernel(k)
bandwidth.kernel(k)
is.tskernel(k)
## S3 method for class 'tskernel':
plot(x, type = "h", xlab = "k", ylab = "W[k]",
     main = attr(x,"name"), ...)
```

Arguments

`coef`	the upper half of the smoothing kernel coefficients (including coefficient zero) *or* the name of a kernel (currently `"daniell"`, `"dirichlet"`, `"fejer"` or `"modified.daniell"`.
`m`	the kernel dimension(s). When `m` has length larger than one, it means the convolution of kernels of dimension `m[j]`, for `j in 1:length(m)`. Currently this is supported only for the named "*daniell" kernels.
`name`	the name the kernel will be called.
`r`	the kernel order for a Fejer kernel.
`k,x`	a `"tskernel"` object.
`type, xlab, ylab, main, ...`	arguments passed to `plot.default`.

Details

`kernel` is used to construct a general kernel or named specific kernels. The modified Daniell kernel halves the end coefficients (as used by S-PLUS).

The `[` method allows natural indexing of kernel objects with indices in `(-m) : m`. The normalization is such that for `k <- kernel(*)`, `sum(k[ -k$m : k$m ])` is one.

`df.kernel` returns the 'equivalent degrees of freedom' of a smoothing kernel as defined in Brockwell and Davis (1991), page 362, and `bandwidth.kernel` returns the equivalent bandwidth as defined in Bloomfield (1976), p. 201, with a continuity correction.

Value

`kernel()` returns an object of class `"tskernel"` which is basically a list with the two components `coef` and the kernel dimension `m`. An additional attribute is `"name"`.

Author(s)

A. Trapletti; modifications by B.D. Ripley

References

Bloomfield, P. (1976) *Fourier Analysis of Time Series: An Introduction.* Wiley.

Brockwell, P.J. and Davis, R.A. (1991) *Time Series: Theory and Methods.* Second edition. Springer, pp. 350–365.

See Also

kernapply

Examples

```
require(graphics)

## Demonstrate a simple trading strategy for the
## financial time series German stock index DAX.
x <- EuStockMarkets[,1]
k1 <- kernel("daniell", 50)  # a long moving average
k2 <- kernel("daniell", 10)  # and a short one
plot(k1)
plot(k2)
x1 <- kernapply(x, k1)
x2 <- kernapply(x, k2)
plot(x)
lines(x1, col = "red")    # go long if the short crosses the long upwards
lines(x2, col = "green")  # and go short otherwise

## More interesting kernels
kd <- kernel("daniell", c(3,3))
kd # note the unusual indexing
kd[-2:2]
plot(kernel("fejer", 100, r=6))
plot(kernel("modified.daniell", c(7,5,3)))

# Reproduce example 10.4.3 from Brockwell and Davis (1991)
spectrum(sunspot.year, kernel=kernel("daniell", c(11,7,3)), log="no")
```

---

kmeans *K-Means Clustering*

Description

Perform k-means clustering on a data matrix.

Usage

```
kmeans(x, centers, iter.max = 10, nstart = 1,
       algorithm = c("Hartigan-Wong", "Lloyd", "Forgy",
                     "MacQueen"))
```

Arguments

x	A numeric matrix of data, or an object that can be coerced to such a matrix (such as a numeric vector or a data frame with all numeric columns).
centers	Either the number of clusters or a set of initial (distinct) cluster centres. If a number, a random set of (distinct) rows in x is chosen as the initial centres.
iter.max	The maximum number of iterations allowed.
nstart	If centers is a number, how many random sets should be chosen?
algorithm	character: may be abbreviated.

Details

The data given by x is clustered by the k-means method, which aims to partition the points into k groups such that the sum of squares from points to the assigned cluster centres is minimized. At the minimum, all cluster centres are at the mean of their Voronoi sets (the set of data points which are nearest to the cluster centre).

The algorithm of Hartigan and Wong (1979) is used by default. Note that some authors use k-means to refer to a specific algorithm rather than the general method: most commonly the algorithm given by MacQueen (1967) but sometimes that given by Lloyd (1957) and Forgy (1965). The Hartigan–Wong algorithm generally does a better job than either of those, but trying several random starts is often recommended.

Except for the Lloyd–Forgy method, k clusters will always be returned if a number is specified. If an initial matrix of centres is supplied, it is possible that no point will be closest to one or more centres, which is currently an error for the Hartigan–Wong method.

Value

An object of class "kmeans" which is a list with components:

cluster	A vector of integers indicating the cluster to which each point is allocated.
centers	A matrix of cluster centres.
withinss	The within-cluster sum of squares for each cluster.
size	The number of points in each cluster.

There is a print method for this class.

References

Forgy, E. W. (1965) Cluster analysis of multivariate data: efficiency vs interpretability of classifications. *Biometrics* **21**, 768–769.

Hartigan, J. A. and Wong, M. A. (1979). A K-means clustering algorithm. *Applied Statistics* **28**, 100–108.

Lloyd, S. P. (1957, 1982) Least squares quantization in PCM. Technical Note, Bell Laboratories. Published in 1982 in *IEEE Transactions on Information Theory* **28**, 128–137.

MacQueen, J. (1967) Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, eds L. M. Le Cam & J. Neyman, **1**, pp. 281–297. Berkeley, CA: University of California Press.

Examples

```
require(graphics)

# a 2-dimensional example
x <- rbind(matrix(rnorm(100, sd = 0.3), ncol = 2),
           matrix(rnorm(100, mean = 1, sd = 0.3), ncol = 2))
colnames(x) <- c("x", "y")
(cl <- kmeans(x, 2))
plot(x, col = cl$cluster)
points(cl$centers, col = 1:2, pch = 8, cex=2)

## random starts do help here with too many clusters
(cl <- kmeans(x, 5, nstart = 25))
plot(x, col = cl$cluster)
points(cl$centers, col = 1:5, pch = 8)
```

---

kruskal.test *Kruskal-Wallis Rank Sum Test*

Description

Performs a Kruskal-Wallis rank sum test.

Usage

```
kruskal.test(x, ...)

## Default S3 method:
kruskal.test(x, g, ...)

## S3 method for class 'formula':
kruskal.test(formula, data, subset, na.action, ...)
```

Arguments

`x`	a numeric vector of data values, or a list of numeric data vectors.
`g`	a vector or factor object giving the group for the corresponding elements of `x`. Ignored if `x` is a list.
`formula`	a formula of the form `lhs ~ rhs` where `lhs` gives the data values and `rhs` the corresponding groups.
`data`	an optional matrix or data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`subset`	an optional vector specifying a subset of observations to be used.
`na.action`	a function which indicates what should happen when the data contain `NA`s. Defaults to `getOption("na.action")`.
`...`	further arguments to be passed to or from methods.

Details

kruskal.test performs a Kruskal-Wallis rank sum test of the null that the location parameters of the distribution of x are the same in each group (sample). The alternative is that they differ in at least one.

If x is a list, its elements are taken as the samples to be compared, and hence have to be numeric data vectors. In this case, g is ignored, and one can simply use kruskal.test(x) to perform the test. If the samples are not yet contained in a list, use kruskal.test(list(x, ...)).

Otherwise, x must be a numeric data vector, and g must be a vector or factor object of the same length as x giving the group for the corresponding elements of x.

Value

A list with class "htest" containing the following components:

statistic	the Kruskal-Wallis rank sum statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	the character string "Kruskal-Wallis rank sum test".
data.name	a character string giving the names of the data.

References

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric Statistical Methods.* New York: John Wiley & Sons. Pages 115–120.

See Also

The Wilcoxon rank sum test (wilcox.test) as the special case for two samples; lm together with anova for performing one-way location analysis under normality assumptions; with Student's t test (t.test) as the special case for two samples.

Examples

```
## Hollander & Wolfe (1973), 116.
## Mucociliary efficiency from the rate of removal of dust in normal
##  subjects, subjects with obstructive airway disease, and subjects
##  with asbestosis.
x <- c(2.9, 3.0, 2.5, 2.6, 3.2) # normal subjects
y <- c(3.8, 2.7, 4.0, 2.4)      # with obstructive airway disease
z <- c(2.8, 3.4, 3.7, 2.2, 2.0) # with asbestosis
kruskal.test(list(x, y, z))
## Equivalently,
x <- c(x, y, z)
g <- factor(rep(1:3, c(5, 4, 5)),
            labels = c("Normal subjects",
                       "Subjects with obstructive airway disease",
                       "Subjects with asbestosis"))
kruskal.test(x, g)

## Formula interface.
require(graphics)
boxplot(Ozone ~ Month, data = airquality)
kruskal.test(Ozone ~ Month, data = airquality)
```

`ks.test` *Kolmogorov-Smirnov Tests*

Description

Performs one or two sample Kolmogorov-Smirnov tests.

Usage

```
ks.test(x, y, ...,
        alternative = c("two.sided", "less", "greater"),
        exact = NULL)
```

Arguments

`x`	a numeric vector of data values.
`y`	either a numeric vector of data values, or a character string naming a cumulative distribution function or an actual cumulative distribution function such as `pnorm`.
`...`	parameters of the distribution specified (as a character string) by `y`.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"` (default), `"less"`, or `"greater"`. You can specify just the initial letter of the value, but the argument name must be give in full. See 'Details' for the meanings of the possible values.
`exact`	`NULL` or a logical indicating whether an exact p-value should be computed. See 'Details' for the meaning of `NULL`. Not used for the one-sided two-sample case.

Details

If `y` is numeric, a two-sample test of the null hypothesis that `x` and `y` were drawn from the same *continuous* distribution is performed.

Alternatively, `y` can be a character string naming a continuous (cumulative) distribution function, or such a function. In this case, a one-sample test is carried out of the null that the distribution function which generated `x` is distribution `y` with parameters specified by `...`.

The presence of ties generates a warning, since continuous distributions do not generate them.

The possible values `"two.sided"`, `"less"` and `"greater"` of `alternative` specify the null hypothesis that the true distribution function of `x` is equal to, not less than or not greater than the hypothesized distribution function (one-sample case) or the distribution function of `y` (two-sample case), respectively. This is a comparison of cumulative distribution functions, and the test statistic is the maximum difference in value, with the statistic in the `"greater"` alternative being $D^+ = \max_u[F_x(u) - F_y(u)]$. Thus in the two-sample case `alternative="greater"` includes distributions for which `x` is stochastically *smaller* than `y` (the CDF of `x` lies above and hence to the left of that for `y`), in contrast to `t.test` or `wilcox.test`.

Exact p-values are not available for the one-sided two-sample case, or in the case of ties. If `exact = NULL` (the default), an exact p-value is computed if the sample size is less than 100 in the one-sample case, and if the product of the sample sizes is less than 10000 in the two-sample case. Otherwise, asymptotic distributions are used whose approximations may be inaccurate in small samples. In the one-sample two-sided case, exact p-values are obtained as described in Marsaglia,

Tsang & Wang (2003). The formula of Birnbaum & Tingey (1951) is used for the one-sample one-sided case.

If a single-sample test is used, the parameters specified in `...` must be pre-specified and not estimated from the data. There is some more refined distribution theory for the KS test with estimated parameters (see Durbin, 1973), but that is not implemented in `ks.test`.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic.
`p.value`	the p-value of the test.
`alternative`	a character string describing the alternative hypothesis.
`method`	a character string indicating what type of test was performed.
`data.name`	a character string giving the name(s) of the data.

References

Z. W. Birnbaum & Fred H. Tingey (1951), One-sided confidence contours for probability distribution functions. *The Annals of Mathematical Statistics*, **22**/4, 592–596.

William J. Conover (1971), *Practical Nonparametric Statistics*. New York: John Wiley & Sons. Pages 295–301 (one-sample Kolmogorov test), 309–314 (two-sample Smirnov test).

Durbin, J. (1973) *Distribution theory for tests based on the sample distribution function*. SIAM.

George Marsaglia, Wai Wan Tsang & Jingbo Wang (2003), Evaluating Kolmogorov's distribution. *Journal of Statistical Software*, **8**/18. http://www.jstatsoft.org/v08/i18/.

See Also

`shapiro.test` which performs the Shapiro-Wilk test for normality.

Examples

```
require(graphics)

x <- rnorm(50)
y <- runif(30)
# Do x and y come from the same distribution?
ks.test(x, y)
# Does x come from a shifted gamma distribution with shape 3 and rate 2?
ks.test(x+2, "pgamma", 3, 2) # two-sided, exact
ks.test(x+2, "pgamma", 3, 2, exact = FALSE)
ks.test(x+2, "pgamma", 3, 2, alternative = "gr")

# test if x is stochastically larger than x2
x2 <- rnorm(50, -1)
plot(ecdf(x), xlim=range(c(x, x2)))
plot(ecdf(x2), add=TRUE, lty="dashed")
t.test(x, x2, alternative="g")
wilcox.test(x, x2, alternative="g")
ks.test(x, x2, alternative="l")
```

`ksmooth` *Kernel Regression Smoother*

Description

The Nadaraya–Watson kernel regression estimate.

Usage

```
ksmooth(x, y, kernel = c("box", "normal"), bandwidth = 0.5,
        range.x = range(x),
        n.points = max(100, length(x)), x.points)
```

Arguments

`x`	input x values
`y`	input y values
`kernel`	the kernel to be used.
`bandwidth`	the bandwidth. The kernels are scaled so that their quartiles (viewed as probability densities) are at $\pm$ `0.25*bandwidth`.
`range.x`	the range of points to be covered in the output.
`n.points`	the number of points at which to evaluate the fit.
`x.points`	points at which to evaluate the smoothed fit. If missing, `n.points` are chosen uniformly to cover `range.x`.

Value

A list with components

`x`	values at which the smoothed fit is evaluated. Guaranteed to be in increasing order.
`y`	fitted values corresponding to `x`.

Note

This function is implemented purely for compatibility with S, although it is nowhere near as slow as the S function. Better kernel smoothers are available in other packages.

Examples

```
require(graphics)

with(cars, {
    plot(speed, dist)
    lines(ksmooth(speed, dist, "normal", bandwidth=2), col=2)
    lines(ksmooth(speed, dist, "normal", bandwidth=5), col=3)
})
```

`lag` *Lag a Time Series*

Description

Compute a lagged version of a time series, shifting the time base back by a given number of observations.

Usage

```
lag(x, ...)

## Default S3 method:
lag(x, k = 1, ...)
```

Arguments

`x`	A vector or matrix or univariate or multivariate time series
`k`	The number of lags (in units of observations).
`...`	further arguments to be passed to or from methods.

Details

Vector or matrix arguments `x` are coerced to time series.

`lag` is a generic function; this page documents its default method.

Value

A time series object.

Note

Note the sign of `k`: a series lagged by a positive `k` starts *earlier*.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`diff`, `deltat`

Examples

```
lag(ldeaths, 12) # starts one year earlier
```

`lag.plot` *Time Series Lag Plots*

Description

Plot time series against lagged versions of themselves. Helps visualizing 'auto-dependence' even when auto-correlations vanish.

Usage

```
lag.plot(x, lags = 1, layout = NULL, set.lags = 1:lags,
         main = NULL, asp = 1,
         diag = TRUE, diag.col = "gray", type = "p", oma = NULL,
         ask = NULL, do.lines = (n <= 150), labels = do.lines,
         ...)
```

Arguments

`x`	time-series (univariate or multivariate)
`lags`	number of lag plots desired, see arg `set.lags`.
`layout`	the layout of multiple plots, basically the `mfrow` `par()` argument. The default uses about a square layout (see `n2mfrow` such that all plots are on one page.
`set.lags`	vector of positive integers allowing specification of the set of lags used; defaults to `1:lags`.
`main`	character with a main header title to be done on the top of each page.
`asp`	Aspect ratio to be fixed, see `plot.default`.
`diag`	logical indicating if the x=y diagonal should be drawn.
`diag.col`	color to be used for the diagonal `if(diag)`.
`type`	plot type to be used, but see `plot.ts` about its restricted meaning.
`oma`	outer margins, see `par`.
`ask`	logical or `NULL`; if true, the user is asked to confirm before a new page is started.
`do.lines`	logical indicating if lines should be drawn.
`labels`	logical indicating if labels should be used.
`...`	Further arguments to `plot.ts`. Several graphical parameters are set in this function and so cannot be changed: these include `xlab`, `ylab`, `mgp`, `col.lab` and `font.lab`: this also applies to the arguments `xy.labels` and `xy.lines`.

Details

If just one plot is produced, this is a conventional plot. If more than one plot is to be produced, `par(mfrow)` and several other graphics parameters will be set, so it is not (easily) possible to mix such lag plots with other plots on the same page.

If `ask = NULL`, `par(ask = TRUE)` will be called if more than one page of plots is to be produced and the device is interactive.

Note

It is more flexible and has different default behaviour than the S version. We use `main =` instead of `head =` for internal consistency.

Author(s)

Martin Maechler

See Also

`plot.ts` which is the basic work horse.

Examples

```
require(graphics)

lag.plot(nhtemp, 8, diag.col = "forest green")
lag.plot(nhtemp, 5, main="Average Temperatures in New Haven")
## ask defaults to TRUE when we have more than one page:
lag.plot(nhtemp, 6, layout = c(2,1), asp = NA,
         main = "New Haven Temperatures", col.main = "blue")

## Multivariate (but non-stationary! ...)
lag.plot(freeny.x, lags = 3)
## Not run:
no lines for long series :
lag.plot(sqrt(sunspots), set = c(1:4, 9:12), pch = ".", col = "gold")
## End(Not run)
```

`line` *Robust Line Fitting*

Description

Fit a line robustly as recommended in *Exploratory Data Analysis*.

Usage

```
line(x, y)
```

Arguments

`x,y`	the arguments can be any way of specifying x-y pairs.

Value

An object of class `"tukeyline"`.

Methods are available for the generic functions `coef`, `residuals`, `fitted`, and `print`.

References

Tukey, J. W. (1977). *Exploratory Data Analysis*, Reading Massachusetts: Addison-Wesley.

See Also

lm.

Examples

```
require(graphics)

plot(cars)
(z <- line(cars))
abline(coef(z))
## Tukey-Anscombe Plot :
plot(residuals(z) ~ fitted(z), main = deparse(z$call))
```

---

lm *Fitting Linear Models*

Description

lm is used to fit linear models. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although aov may provide a more convenient interface for these).

Usage

```
lm(formula, data, subset, weights, na.action,
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
   singular.ok = TRUE, contrasts = NULL, offset, ...)
```

Arguments

formula	an object of class "formula" (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under 'Details'.
data	an optional data frame, list or environment (or object coercible by as.data.frame to a data frame) containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which lm is called.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
weights	an optional vector of weights to be used in the fitting process. Should be NULL or a numeric vector. If non-NULL, weighted least squares is used with weights weights (that is, minimizing sum(w*e^2)); otherwise ordinary least squares is used.
na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The 'factory-fresh' default is na.omit. Another possible value is NULL, no action. Value na.exclude can be useful.
method	the method to be used; for fitting, currently only method = "qr" is supported; method = "model.frame" returns the model frame (the same as with model = TRUE, see below).

`model, x, y, qr`
: logicals. If `TRUE` the corresponding components of the fit (the model frame, the model matrix, the response, the QR decomposition) are returned.

`singular.ok`
: logical. If `FALSE` (the default in S but not in R) a singular fit is an error.

`contrasts`
: an optional list. See the `contrasts.arg` of `model.matrix.default`.

`offset`
: this can be used to specify an *a priori* known component to be included in the linear predictor during fitting. This should be `NULL` or a numeric vector of length either one or equal to the number of cases. One or more `offset` terms can be included in the formula instead or as well, and if both are specified their sum is used. See `model.offset`.

`...`
: additional arguments to be passed to the low level regression fitting functions (see below).

Details

Models for `lm` are specified symbolically. A typical model has the form `response ~ terms` where `response` is the (numeric) response vector and `terms` is a series of terms which specifies a linear predictor for `response`. A terms specification of the form `first + second` indicates all the terms in `first` together with all the terms in `second` with duplicates removed. A specification of the form `first:second` indicates the set of terms obtained by taking the interactions of all terms in `first` with all terms in `second`. The specification `first*second` indicates the *cross* of `first` and `second`. This is the same as `first + second + first:second`.

If the formula includes an `offset`, this is evaluated and subtracted from the response.

If `response` is a matrix a linear model is fitted separately by least-squares to each column of the matrix.

See `model.matrix` for some further details. The terms in the formula will be re-ordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on: to avoid this pass a `terms` object as the formula (see `aov` and `demo(glm.vr)` for an example).

A formula has an implied intercept term. To remove this use either `y ~ x - 1` or `y ~ 0 + x`. See `formula` for more details of allowed formulae.

`lm` calls the lower level functions `lm.fit`, etc, see below, for the actual numerical computations. For programming only, you may consider doing likewise.

All of `weights`, `subset` and `offset` are evaluated in the same way as variables in `formula`, that is first in `data` and then in the environment of `formula`.

Value

`lm` returns an object of `class` `"lm"` or for multiple responses of class `c("mlm", "lm")`.

The functions `summary` and `anova` are used to obtain and print a summary and analysis of variance table of the results. The generic accessor functions `coefficients`, `effects`, `fitted.values` and `residuals` extract various useful features of the value returned by `lm`.

An object of class `"lm"` is a list containing at least the following components:

`coefficients` a named vector of coefficients

`residuals` the residuals, that is response minus fitted values.

`fitted.values`
: the fitted mean values.

`rank` the numeric rank of the fitted linear model.

`weights` (only for weighted fits) the specified weights.

df.residual	the residual degrees of freedom.
call	the matched call.
terms	the terms object used.
contrasts	(only where relevant) the contrasts used.
xlevels	(only where relevant) a record of the levels of the factors used in fitting.
offset	the offset used (missing if none were used).
y	if requested, the response used.
x	if requested, the model matrix used.
model	if requested (the default), the model frame used.
na.action	(where relevant) information returned by model.frame on the special handling of NAs.

In addition, non-null fits will have components assign, effects and (unless not requested) qr relating to the linear fit, for use by extractor functions such as summary and effects.

Using time series

Considerable care is needed when using lm with time series.

Unless na.action = NULL, the time series attributes are stripped from the variables before the regression is done. (This is necessary as omitting NAs would invalidate the time series attributes, and if NAs are omitted in the middle of the series the result would no longer be a regular time series.)

Even if the time series attributes are retained, they are not used to line up series, so that the time shift of a lagged or differenced regressor would be ignored. It is good practice to prepare a data argument by ts.intersect(..., dframe = TRUE), then apply a suitable na.action to that data frame and call lm with na.action = NULL so that residuals and fitted values are time series.

Note

Offsets specified by offset will not be included in predictions by predict.lm, whereas those specified by an offset term in the formula will be.

Author(s)

The design was inspired by the S function of the same name described in Chambers (1992). The implementation of model formula by Ross Ihaka was based on Wilkinson & Rogers (1973).

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Wilkinson, G. N. and Rogers, C. E. (1973) Symbolic descriptions of factorial models for analysis of variance. *Applied Statistics*, **22**, 392–9.

See Also

summary.lm for summaries and anova.lm for the ANOVA table; aov for a different interface.

The generic functions coef, effects, residuals, fitted, vcov.

predict.lm (via predict) for prediction, including confidence and prediction intervals; confint for confidence intervals of *parameters*.

lm.influence for regression diagnostics, and glm for **generalized** linear models.

The underlying low level functions, lm.fit for plain, and lm.wfit for weighted regression fitting.

More lm() examples are available e.g., in anscombe, attitude, freeny, LifeCycleSavings, longley, stackloss, swiss.

biglm in package **biglm** for an alternative way to fit linear models to large datasets (especially those with many cases).

Examples

```
require(graphics)

## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2,10,20, labels=c("Ctl","Trt"))
weight <- c(ctl, trt)
anova(lm.D9 <- lm(weight ~ group))
summary(lm.D90 <- lm(weight ~ group - 1))# omitting intercept
summary(resid(lm.D9) - resid(lm.D90)) #- residuals almost identical

opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(lm.D9, las = 1)      # Residuals, Fitted, ...
par(opar)

## model frame :
stopifnot(identical(lm(weight ~ group, method = "model.frame"),
                    model.frame(lm.D9)))

### less simple examples in "See Also" above
```

---

lm.fit *Fitter Functions for Linear Models*

Description

These are the basic computing engines called by lm used to fit linear models. These should usually *not* be used directly unless by experienced users.

Usage

```
lm.fit (x, y,     offset = NULL, method = "qr", tol = 1e-7,
        singular.ok = TRUE, ...)

lm.wfit(x, y, w, offset = NULL, method = "qr", tol = 1e-7,
         singular.ok = TRUE, ...)
```

Arguments

`x`	design matrix of dimension `n * p`.
`y`	vector of observations of length `n`, or a matrix with `n` rows.
`w`	vector of weights (length `n`) to be used in the fitting process for the `wfit` functions. Weighted least squares is used with weights `w`, i.e., `sum(w * e^2)` is minimized.
`offset`	numeric of length `n`). This can be used to specify an *a priori* known component to be included in the linear predictor during fitting.
`method`	currently, only `method="qr"` is supported.
`tol`	tolerance for the `qr` decomposition. Default is 1e-7.
`singular.ok`	logical. If `FALSE`, a singular model is an error.
`...`	currently disregarded.

Value

a list with components

`coefficients`	`p` vector
`residuals`	`n` vector or matrix
`fitted.values`	`n` vector or matrix
`effects`	(not null fits)`n` vector of orthogonal single-df effects. The first `rank` of them correspond to non-aliased coeffcients, and are named accordingly.
`weights`	`n` vector — *only* for the `*wfit*` functions.
`rank`	integer, giving the rank
`df.residual`	degrees of freedom of residuals
`qr`	(not null fits) the QR decomposition, see `qr`.

See Also

`lm` which you should use for linear least squares regression, unless you know better.

Examples

```
require(utils)
set.seed(129)
n <- 7 ; p <- 2
X <- matrix(rnorm(n * p), n,p) # no intercept!
y <- rnorm(n)
w <- rnorm(n)^2

str(lmw <- lm.wfit(x=X, y=y, w=w))

str(lm. <- lm.fit (x=X, y=y))
```

`lm.influence` *Regression Diagnostics*

Description

This function provides the basic quantities which are used in forming a wide variety of diagnostics for checking the quality of regression fits.

Usage

```
influence(model, ...)
## S3 method for class 'lm':
influence(model, do.coef = TRUE, ...)
## S3 method for class 'glm':
influence(model, do.coef = TRUE, ...)

lm.influence(model, do.coef = TRUE)
```

Arguments

`model`	an object as returned by `lm` or `glm`.
`do.coef`	logical indicating if the changed `coefficients` (see below) are desired. These need $O(n^2p)$ computing time.
`...`	further arguments passed to or from other methods.

Details

The `influence.measures()` and other functions listed in **See Also** provide a more user oriented way of computing a variety of regression diagnostics. These all build on `lm.influence`. Note that for GLMs (other than the Gaussian family with identity link) these are based on one-step approximations which may be inadequate if a case has high influence.

An attempt is made to ensure that computed hat values that are probably one are treated as one, and the corresponding rows in `sigma` and `coefficients` are `NaN`. (Dropping such a case would normally result in a variable being dropped, so it is not possible to give simple drop-one diagnostics.)

`naresid` is applied to the results and so will fill in with `NA`s it the fit had `na.action = na.exclude`.

Value

A list containing the following components of the same length or number of rows n, which is the number of non-zero weights. Cases omitted in the fit are omitted unless a `na.action` method was used (such as `na.exclude`) which restores them.

`hat`	a vector containing the diagonal of the 'hat' matrix.
`coefficients`	(unless `do.coef` is false) a matrix whose i-th row contains the change in the estimated coefficients which results when the i-th case is dropped from the regression. Note that aliased coefficients are not included in the matrix.

`sigma`	a vector whose i-th element contains the estimate of the residual standard deviation obtained when the i-th case is dropped from the regression. (The approximations needed for GLMs can result in this being `NaN`.)
`wt.res`	a vector of *weighted* (or for class `glm` rather *deviance*) residuals.

Note

The `coefficients` returned by the R version of `lm.influence` differ from those computed by S. Rather than returning the coefficients which result from dropping each case, we return the changes in the coefficients. This is more directly useful in many diagnostic measures.
Since these need $O(n^2p)$ computing time, they can be omitted by `do.coef = FALSE`.

Note that cases with `weights == 0` are *dropped* (contrary to the situation in S).

If a model has been fitted with `na.action=na.exclude` (see `na.exclude`), cases excluded in the fit *are* considered here.

References

See the list in the documentation for `influence.measures`.

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`summary.lm` for `summary` and related methods;
`influence.measures`,
`hat` for the hat matrix diagonals,
`dfbetas`, `dffits`, `covratio`, `cooks.distance`, `lm`.

Examples

```
## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
summary(lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi,
                    data = LifeCycleSavings),
        corr = TRUE)
utils::str(lmI <- lm.influence(lm.SR))

## For more "user level" examples, use example(influence.measures)
```

---

lm.summaries *Accessing Linear Model Fits*

Description

All these functions are `methods` for class `"lm"` objects.

Usage

```
## S3 method for class 'lm':
family(object, ...)

## S3 method for class 'lm':
formula(x, ...)

## S3 method for class 'lm':
residuals(object,
          type = c("working", "response", "deviance", "pearson",
                   "partial"),
          ...)

## S3 method for class 'lm':
labels(object, ...)

weights(object, ...)
```

Arguments

`object, x`	an object inheriting from class `lm`, usually the result of a call to `lm` or `aov`.
`...`	further arguments passed to or from other methods.
`type`	the type of residuals which should be returned.

Details

The generic accessor functions `coef`, `effects`, `fitted` and `residuals` can be used to extract various useful features of the value returned by `lm`.

The working and response residuals are 'observed - fitted'. The deviance and pearson residuals are weighted residuals, scaled by the square root of the weights used in fitting. The partial residuals are a matrix with each column formed by omitting a term from the model. In all these, zero weight cases are never omitted (as opposed to the standardized `rstudent` residuals, and the `weighted.residuals`).

How `residuals` treats cases with missing values in the original fit is determined by the `na.action` argument of that fit. If `na.action = na.omit` omitted cases will not appear in the residuals, whereas if `na.action = na.exclude` they will appear, with residual value `NA`. See also `naresid`.

The `"lm"` method for generic `labels` returns the term labels for estimable terms, that is the names of the terms with an least one estimable coefficient.

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

The model fitting function `lm`, `anova.lm`.

`coef`, `deviance`, `df.residual`, `effects`, `fitted`, `glm` for **generalized** linear models, `influence` (etc on that page) for regression diagnostics, `weighted.residuals`, `residuals`, `residuals.glm`, `summary.lm`.

influence.measures for deletion diagnostics, including standardized (`rstandard`) and studentized (`rstudent`) residuals.

Examples

```
##-- Continuing the  lm(.) example:
coef(lm.D90)# the bare coefficients

## The 2 basic regression diagnostic plots [plot.lm(.) is preferred]
plot(resid(lm.D90), fitted(lm.D90))# Tukey-Anscombe's
abline(h=0, lty=2, col = 'gray')

qqnorm(residuals(lm.D90))
```

`loadings` *Print Loadings in Factor Analysis*

Description

Extract or print loadings in factor analysis (or principal components analysis).

Usage

```
loadings(x)

## S3 method for class 'loadings':
print(x, digits = 3, cutoff = 0.1, sort = FALSE, ...)

## S3 method for class 'factanal':
print(x, digits = 3, ...)
```

Arguments

`x`	an object of class `"factanal"` or `"princomp"` or the `loadings` component of such an object.
`digits`	number of decimal places to use in printing uniquenesses and loadings.
`cutoff`	loadings smaller than this (in absolute value) are suppressed.
`sort`	logical. If true, the variables are sorted by their importance on each factor. Each variable with any loading larger than 0.5 (in modulus) is assigned to the factor with the largest loading, and the variables are printed in the order of the factor they are assigned to, then those unassigned.
`...`	further arguments for other methods, such as `cutoff` and `sort` for `print.factanal`.

See Also

`factanal`, `princomp`

`loess` *Local Polynomial Regression Fitting*

Description

Fit a polynomial surface determined by one or more numerical predictors, using local fitting.

Usage

```
loess(formula, data, weights, subset, na.action, model = FALSE,
      span = 0.75, enp.target, degree = 2,
      parametric = FALSE, drop.square = FALSE, normalize = TRUE,
      family = c("gaussian", "symmetric"),
      method = c("loess", "model.frame"),
      control = loess.control(...), ...)
```

Arguments

`formula`	a formula specifying the numeric response and one to four numeric predictors (best specified via an interaction, but can also be specified additively). Will be coerced to a formula if necessary.
`data`	an optional data frame, list or environment (or object coercible by `as.data.frame` to a data frame) containing the variables in the model. If not found in `data`, the variables are taken from `environment(formula)`, typically the environment from which `loess` is called.
`weights`	optional weights for each case.
`subset`	an optional specification of a subset of the data to be used.
`na.action`	the action to be taken with missing values in the response or predictors. The default is given by `getOption("na.action")`.
`model`	should the model frame be returned?
`span`	the parameter α which controls the degree of smoothing.
`enp.target`	an alternative way to specify `span`, as the approximate equivalent number of parameters to be used.
`degree`	the degree of the polynomials to be used, up to 2.
`parametric`	should any terms be fitted globally rather than locally? Terms can be specified by name, number or as a logical vector of the same length as the number of predictors.
`drop.square`	for fits with more than one predictor and `degree=2`, should the quadratic term (and cross-terms) be dropped for particular predictors? Terms are specified in the same way as for `parametric`.
`normalize`	should the predictors be normalized to a common scale if there is more than one? The normalization used is to set the 10% trimmed standard deviation to one. Set to false for spatial coordinate predictors and others know to be a common scale.
`family`	if `"gaussian"` fitting is by least-squares, and if `"symmetric"` a re-descending M estimator is used with Tukey's biweight function.
`method`	fit the model or just extract the model frame.
`control`	control parameters: see `loess.control`.
`...`	control parameters can also be supplied directly.

Details

Fitting is done locally. That is, for the fit at point x, the fit is made using points in a neighbourhood of x, weighted by their distance from x (with differences in 'parametric' variables being ignored when computing the distance). The size of the neighbourhood is controlled by α (set by `span` or `enp.target`). For $\alpha < 1$, the neighbourhood includes proportion α of the points, and these have tricubic weighting (proportional to $(1 - (\text{dist}/\text{maxdist})^3)^3$. For $\alpha > 1$, all points are used, with the 'maximum distance' assumed to be $\alpha^{1/p}$ times the actual maximum distance for p explanatory variables.

For the default family, fitting is by (weighted) least squares. For `family="symmetric"` a few iterations of an M-estimation procedure with Tukey's biweight are used. Be aware that as the initial value is the least-squares fit, this need not be a very resistant fit.

It can be important to tune the control list to achieve acceptable speed. See `loess.control` for details.

Value

An object of class `"loess"`.

Note

As this is based on the `cloess` package available at `netlib`, it is similar to but not identical to the `loess` function of S. In particular, conditioning is not implemented.

The memory usage of this implementation of `loess` is roughly quadratic in the number of points, with 1000 points taking about 10Mb.

Author(s)

B. D. Ripley, based on the `cloess` package of Cleveland, Grosse and Shyu available at `http://www.netlib.org/a/`.

References

W. S. Cleveland, E. Grosse and W. M. Shyu (1992) Local regression models. Chapter 8 of *Statistical Models in S* eds J.M. Chambers and T.J. Hastie, Wadsworth & Brooks/Cole.

See Also

`loess.control`, `predict.loess`.

`lowess`, the ancestor of `loess` (with different defaults!).

Examples

```
cars.lo <- loess(dist ~ speed, cars)
predict(cars.lo, data.frame(speed = seq(5, 30, 1)), se = TRUE)
# to allow extrapolation
cars.lo2 <- loess(dist ~ speed, cars,
  control = loess.control(surface = "direct"))
predict(cars.lo2, data.frame(speed = seq(5, 30, 1)), se = TRUE)
```

`loess.control` *Set Parameters for Loess*

Description

Set control parameters for `loess` fits.

Usage

```
loess.control(surface = c("interpolate", "direct"),
              statistics = c("approximate", "exact"),
              trace.hat = c("exact", "approximate"),
              cell = 0.2, iterations = 4, ...)
```

Arguments

`surface`	should be fitted surface be computed exactly or via interpolation from a kd tree?
`statistics`	should the statistics be computed exactly or approximately? Exact computation can be very slow.
`trace.hat`	should the trace of the smoother matrix be computed exactly or approximately? It is recommended to use the approximation for more than about 1000 data points.
`cell`	if interpolation is used this controls the accuracy of the approximation via the maximum number of points in a cell in the kd tree. Cells with more than `floor(n*span*cell)` points are subdivided.
`iterations`	the number of iterations used in robust fitting.
`...`	further arguments which are ignored.

Value

A list with components

`surface`

`statistics`

`trace.hat`

`cell`

`iterations`

with meanings as explained under 'Arguments'.

See Also

`loess`

`Logistic` *The Logistic Distribution*

Description

Density, distribution function, quantile function and random generation for the logistic distribution with parameters `location` and `scale`.

Usage

```
dlogis(x, location = 0, scale = 1, log = FALSE)
plogis(q, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
qlogis(p, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
rlogis(n, location = 0, scale = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`location, scale`	location and scale parameters.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `location` or `scale` are omitted, they assume the default values of `0` and `1` respectively.

The Logistic distribution with `location` $= \mu$ and `scale` $= \sigma$ has distribution function

$$F(x) = \frac{1}{1 + e^{-(x-\mu)/\sigma}}$$

and density

$$f(x) = \frac{1}{\sigma} \frac{e^{(x-\mu)/\sigma}}{(1 + e^{(x-\mu)/\sigma})^2}$$

It is a long-tailed distribution with mean μ and variance $\pi^2/3\sigma^2$.

Value

`dlogis` gives the density, `plogis` gives the distribution function, `qlogis` gives the quantile function, and `rlogis` generates random deviates.

Note

`qlogis(p)` is the same as the well known '*logit*' function, $logit(p) = \log(p/(1-p))$, and `plogis(x)` has consequently been called the 'inverse logit'.

The distribution function is a rescaled hyperbolic tangent, `plogis(x) == (1+tanh(x/2))/2`, and it is called a *sigmoid function* in contexts such as neural networks.

Source

`[dpr]logis` are calculated directly from the definitions.

`rlogis` uses inversion.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 2, chapter 23. Wiley, New York.

Examples

```
var(rlogis(4000, 0, scale = 5))# approximately (+/- 3)
pi^2/3 * 5^2
```

`logLik` *Extract Log-Likelihood*

Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include: `glm`, `lm`, `nls`, `Arima` and `gls`, `lme` and others in package **nlme**.

Usage

```
logLik(object, ...)

## S3 method for class 'lm':
logLik(object, REML = FALSE, ...)
```

Arguments

`object`	any object from which a log-likelihood value, or a contribution to a log-likelihood value, can be extracted.
`...`	some methods for this generic function require additional arguments.
`REML`	an optional logical value. If `TRUE` the restricted log-likelihood is returned, else, if `FALSE`, the log-likelihood is returned. Defaults to `FALSE`.

Details

For a `"glm"` fit the `family` does not have to specify how to calculate the log-likelihood, so this is based on the family's `aic()` function to compute the AIC. For the `gaussian`, `Gamma` and `inverse.gaussian` families it assumed that the dispersion of the GLM is estimated has been counted as a parameter in the AIC value, and for all other families it is assumed that the dispersion is known.

Note that this procedure is not completely accurate for the gamma and inverse gaussian families, as the estimate of dispersion used is not the MLE.

For `"lm"` fits it is assumed that the scale has been estimated (by maximum likelihood or REML), and all the constants in the log-likelihood are included.

Value

Returns an object, say `r`, of class `logLik` which is a number with attributes, `attr(r, "df")` (**d**egrees of **f**reedom) giving the number of (estimated) parameters in the model. There is a simple `print` method for `logLik` objects.

The details depend on the method function used; see the appropriate documentation.

Author(s)

Jose Pinheiro and Douglas Bates

References

For `logLik.lm`:

Harville, D.A. (1974). Bayesian inference for variance components using only error contrasts. *Biometrika*, **61**, 383–385.

See Also

`logLik.gls`, `logLik.lme`, in package **nlme**, etc.

Examples

```
x <- 1:5
lmx <- lm(x ~ 1)
logLik(lmx) # using print.logLik() method
utils::str(logLik(lmx))

## lm method
(fm1 <- lm(rating ~ ., data = attitude))
logLik(fm1)
logLik(fm1, REML = TRUE)

res <- try(utils::data(Orthodont, package="nlme"))
if(!inherits(res, "try-error")) {
  fm1 <- lm(distance ~ Sex * age, Orthodont)
  print(logLik(fm1))
  print(logLik(fm1, REML = TRUE))
}
```

`loglin` *Fitting Log-Linear Models*

Description

`loglin` is used to fit log-linear models to multidimensional contingency tables by Iterative Proportional Fitting.

Usage

```
loglin(table, margin, start = rep(1, length(table)), fit = FALSE,
       eps = 0.1, iter = 20, param = FALSE, print = TRUE)
```

Arguments

`table`	a contingency table to be fit, typically the output from `table`.
`margin`	a list of vectors with the marginal totals to be fit. (Hierarchical) log-linear models can be specified in terms of these marginal totals which give the 'maximal' factor subsets contained in the model. For example, in a three-factor model, `list(c(1, 2), c(1, 3))` specifies a model which contains parameters for the grand mean, each factor, and the 1-2 and 1-3 interactions, respectively (but no 2-3 or 1-2-3 interaction), i.e., a model where factors 2 and 3 are independent conditional on factor 1 (sometimes represented as '[12][13]'). The names of factors (i.e., `names(dimnames(table))`) may be used rather than numeric indices.
`start`	a starting estimate for the fitted table. This optional argument is important for incomplete tables with structural zeros in `table` which should be preserved in the fit. In this case, the corresponding entries in `start` should be zero and the others can be taken as one.
`fit`	a logical indicating whether the fitted values should be returned.
`eps`	maximum deviation allowed between observed and fitted margins.
`iter`	maximum number of iterations.
`param`	a logical indicating whether the parameter values should be returned.
`print`	a logical. If `TRUE`, the number of iterations and the final deviation are printed.

Details

The Iterative Proportional Fitting algorithm as presented in Haberman (1972) is used for fitting the model. At most `iter` iterations are performed, convergence is taken to occur when the maximum deviation between observed and fitted margins is less than `eps`. All internal computations are done in double precision; there is no limit on the number of factors (the dimension of the table) in the model.

Assuming that there are no structural zeros, both the Likelihood Ratio Test and Pearson test statistics have an asymptotic chi-squared distribution with `df` degrees of freedom.

Note that the IPF steps are applied to the factors in the order given in `margin`. Hence if the model is decomposable and the order given in `margin` is a running intersection property ordering then IPF will converge in one iteration.

Package **MASS** contains `loglm`, a front-end to `loglin` which allows the log-linear model to be specified and fitted in a formula-based manner similar to that of other fitting functions such as `lm` or `glm`.

Value

A list with the following components.

`lrt`	the Likelihood Ratio Test statistic.
`pearson`	the Pearson test statistic (X-squared).
`df`	the degrees of freedom for the fitted model. There is no adjustment for structural zeros.
`margin`	list of the margins that were fit. Basically the same as the input `margin`, but with numbers replaced by names where possible.

`fit`	An array like `table` containing the fitted values. Only returned if `fit` is `TRUE`.
`param`	A list containing the estimated parameters of the model. The 'standard' constraints of zero marginal sums (e.g., zero row and column sums for a two factor parameter) are employed. Only returned if `param` is `TRUE`.

Author(s)

Kurt Hornik

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Haberman, S. J. (1972) Log-linear fit for contingency tables—Algorithm AS51. *Applied Statistics*, **21**, 218–225.

Agresti, A. (1990) *Categorical data analysis*. New York: Wiley.

See Also

`table`.

`loglm` in package **MASS** for a user-friendly wrapper.

`glm` for another way to fit log-linear models.

Examples

```
## Model of joint independence of sex from hair and eye color.
fm <- loglin(HairEyeColor, list(c(1, 2), c(1, 3), c(2, 3)))
fm
1 - pchisq(fm$lrt, fm$df)
## Model with no three-factor interactions fits well.
```

Lognormal *The Log Normal Distribution*

Description

Density, distribution function, quantile function and random generation for the log normal distribution whose logarithm has mean equal to `meanlog` and standard deviation equal to `sdlog`.

Usage

```
dlnorm(x, meanlog = 0, sdlog = 1, log = FALSE)
plnorm(q, meanlog = 0, sdlog = 1, lower.tail = TRUE, log.p = FALSE)
qlnorm(p, meanlog = 0, sdlog = 1, lower.tail = TRUE, log.p = FALSE)
rlnorm(n, meanlog = 0, sdlog = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`meanlog, sdlog`	mean and standard deviation of the distribution on the log scale with default values of `0` and `1` respectively.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The log normal distribution has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} e^{-(\log(x)-\mu)^2/2\sigma^2}$$

where μ and σ are the mean and standard deviation of the logarithm. The mean is $E(X) = exp(\mu + 1/2\sigma^2)$, the median is $med(X) = exp(\mu)$, and the variance $Var(X) = exp(2\mu+\sigma^2)(exp(\sigma^2)-1)$ and hence the coefficient of variation is $\sqrt{exp(\sigma^2) - 1}$ which is approximately σ when that is small (e.g., $\sigma < 1/2$).

Value

`dlnorm` gives the density, `plnorm` gives the distribution function, `qlnorm` gives the quantile function, and `rlnorm` generates random deviates.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-plnorm(t, r, lower = FALSE, log = TRUE)`.

Source

`dlnorm` is calculated from the definition (in 'Details'). `[pqr]lnorm` are based on the relationship to the normal.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 14. Wiley, New York.

See Also

`dnorm` for the normal distribution.

Examples

```
dlnorm(1) == dnorm(0)
```

`lowess` *Scatter Plot Smoothing*

Description

This function performs the computations for the *LOWESS* smoother which uses locally-weighted polynomial regression (see the references).

Usage

```
lowess(x, y = NULL, f = 2/3, iter = 3,
       delta = 0.01 * diff(range(xy$x[o])))
```

Arguments

`x, y`	vectors giving the coordinates of the points in the scatter plot. Alternatively a single plotting structure can be specified – see `xy.coords`.
`f`	the smoother span. This gives the proportion of points in the plot which influence the smooth at each value. Larger values give more smoothness.
`iter`	the number of 'robustifying' iterations which should be performed. Using smaller values of `iter` will make `lowess` run faster.
`delta`	See 'Details'. Defaults to 1/100th of the range of `x`.

Details

`lowess` is defined by a complex algorithm, the Ratfor original of which (by W. S. Cleveland) can be found in the R sources as file 'src/appl/lowess.doc'. Normally a local linear polynomial fit is used, but under some circumstances (see the file) a local constant fit can be used. 'Local' is defined by the distance to the `floor(f*n)` th nearest neighbour, and tricubic weighting is used for `x` which fall within the neighbourhood.

The initial fit is done using weighted least squares. If `iter > 0`, further weighted fits are done using the product of the weights from the proximity of the `x` values and case weights derived from the residuals at the previous iteration. Specifically, the case weight is Tukey's biweight, with cutoff 6 times the MAD of the residuals. (The current R implementation differs from the original in stopping iteration if the MAD is effectively zero since the algorithm is highly unstable in that case.)

`delta` is used to speed up computation: instead of computing the local polynomial fit at each data point it is not computed for points within `delta` of the last computed point, and linear interpolation is used to fill in the fitted values for the skipped points.

Value

`lowess` returns a list containing components `x` and `y` which give the coordinates of the smooth. The smooth can be added to a plot of the original points with the function `lines`: see the examples.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1979) Robust locally weighted regression and smoothing scatterplots. *J. Amer. Statist. Assoc.* **74**, 829–836.

Cleveland, W. S. (1981) LOWESS: A program for smoothing scatterplots by robust locally weighted regression. *The American Statistician*, **35**, 54.

See Also

`loess`, a newer formula based version of `lowess` (with different defaults!).

Examples

```
require(graphics)

plot(cars, main = "lowess(cars)")
lines(lowess(cars), col = 2)
lines(lowess(cars, f=.2), col = 3)
legend(5, 120, c(paste("f = ", c("2/3", ".2"))), lty = 1, col = 2:3)
```

`ls.diag` *Compute Diagnostics for 'lsfit' Regression Results*

Description

Computes basic statistics, including standard errors, t- and p-values for the regression coefficients.

Usage

```
ls.diag(ls.out)
```

Arguments

`ls.out`	Typically the result of `lsfit()`

Value

A `list` with the following numeric components.

`std.dev`	The standard deviation of the errors, an estimate of σ.
`hat`	diagonal entries h_{ii} of the hat matrix H
`std.res`	standardized residuals
`stud.res`	studentized residuals
`cooks`	Cook's distances
`dfits`	DFITS statistics
`correlation`	correlation matrix
`std.err`	standard errors of the regression coefficients
`cov.scaled`	Scaled covariance matrix of the coefficients
`cov.unscaled`	Unscaled covariance matrix of the coefficients

References

Belsley, D. A., Kuh, E. and Welsch, R. E. (1980) *Regression Diagnostics.* New York: Wiley.

See Also

hat for the hat matrix diagonals, ls.print, lm.influence, summary.lm, anova.

Examples

```
##-- Using the same data as the lm(.) example:
lsD9 <- lsfit(x = as.numeric(gl(2, 10, 20)), y = weight)
dlsD9 <- ls.diag(lsD9)
utils::str(dlsD9, give.attr=FALSE)
abs(1 - sum(dlsD9$hat) / 2) < 10*.Machine$double.eps # sum(h.ii) = p
plot(dlsD9$hat, dlsD9$stud.res, xlim=c(0,0.11))
abline(h = 0, lty = 2, col = "lightgray")
```

---

ls.print *Print 'lsfit' Regression Results*

Description

Computes basic statistics, including standard errors, t- and p-values for the regression coefficients and prints them if print.it is TRUE.

Usage

```
ls.print(ls.out, digits = 4, print.it = TRUE)
```

Arguments

ls.out	Typically the result of lsfit()
digits	The number of significant digits used for printing
print.it	a logical indicating whether the result should also be printed

Value

A list with the components

summary	The ANOVA table of the regression
coef.table	matrix with regression coefficients, standard errors, t- and p-values

Note

Usually, you'd rather use summary(lm(...)) and anova(lm(...)) for obtaining similar output.

See Also

ls.diag, lsfit, also for examples; lm, lm.influence which usually are preferable.

`lsfit` *Find the Least Squares Fit*

Description

The least squares estimate of β in the model

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

is found.

Usage

```
lsfit(x, y, wt = NULL, intercept = TRUE, tolerance = 1e-07,
      yname = NULL)
```

Arguments

`x`	a matrix whose rows correspond to cases and whose columns correspond to variables.
`y`	the responses, possibly a matrix if you want to fit multiple left hand sides.
`wt`	an optional vector of weights for performing weighted least squares.
`intercept`	whether or not an intercept term should be used.
`tolerance`	the tolerance to be used in the matrix decomposition.
`yname`	names to be used for the response variables.

Details

If weights are specified then a weighted least squares is performed with the weight given to the *j*th case specified by the *j*th entry in `wt`.

If any observation has a missing value in any field, that observation is removed before the analysis is carried out. This can be quite inefficient if there is a lot of missing data.

The implementation is via a modification of the LINPACK subroutines which allow for multiple left-hand sides.

Value

A list with the following named components:

`coef`	the least squares estimates of the coefficients in the model (β as stated above).
`residuals`	residuals from the fit.
`intercept`	indicates whether an intercept was fitted.
`qr`	the QR decomposition of the design matrix.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`lm` which usually is preferable; `ls.print`, `ls.diag`.

Examples

```
##-- Using the same data as the lm(.) example:
lsD9 <- lsfit(x = unclass(gl(2,10)), y = weight)
ls.print(lsD9)
```

---

mad *Median Absolute Deviation*

Description

Compute the median absolute deviation, i.e., the (lo-/hi-) median of the absolute deviations from the median, and (by default) adjust by a factor for asymptotically normal consistency.

Usage

```
mad(x, center = median(x), constant = 1.4826, na.rm = FALSE,
    low = FALSE, high = FALSE)
```

Arguments

`x`	a numeric vector.
`center`	Optionally, the centre: defaults to the median.
`constant`	scale factor.
`na.rm`	if `TRUE` then `NA` values are stripped from `x` before computation takes place.
`low`	if `TRUE`, compute the 'lo-median', i.e., for even sample size, do not average the two middle values, but take the smaller one.
`high`	if `TRUE`, compute the 'hi-median', i.e., take the larger of the two middle values for even sample size.

Details

The actual value calculated is `constant * cMedian(abs(x - center))` with the default value of `center` being `median(x)`, and `cMedian` being the usual, the 'low' or 'high' median, see the arguments description for `low` and `high` above.

The default `constant = 1.4826` (approximately $1/\Phi^{-1}(\frac{3}{4}) =$ `1/qnorm(3/4)`) ensures consistency, i.e.,

$$E[mad(X_1, \ldots, X_n)] = \sigma$$

for X_i distributed as $N(\mu, \sigma^2)$ and large n.

If `na.rm` is `TRUE` then `NA` values are stripped from `x` before computation takes place. If this is not done then an `NA` value in `x` will cause `mad` to return `NA`.

See Also

`IQR` which is simpler but less robust, `median`, `var`.

Examples

```
mad(c(1:9))
print(mad(c(1:9),     constant=1)) ==
      mad(c(1:8,100), constant=1)       # = 2 ; TRUE
x <- c(1,2,3, 5,7,8)
sort(abs(x - median(x)))
c(mad(x, constant=1),
  mad(x, constant=1, low = TRUE),
  mad(x, constant=1, high = TRUE))
```

---

mahalanobis *Mahalanobis Distance*

Description

Returns the squared Mahalanobis distance of all rows in `x` and the vector $\mu =$`center` with respect to $\Sigma =$`cov`. This is (for vector `x`) defined as

$$D^2 = (x - \mu)'\Sigma^{-1}(x - \mu)$$

Usage

```
mahalanobis(x, center, cov, inverted=FALSE, ...)
```

Arguments

`x`	vector or matrix of data with, say, p columns.
`center`	mean vector of the distribution or second data vector of length p.
`cov`	covariance matrix ($p \times p$) of the distribution.
`inverted`	logical. If `TRUE`, `cov` is supposed to contain the *inverse* of the covariance matrix.
`...`	passed to `solve` for computing the inverse of the covariance matrix (if `inverted` is false).

See Also

`cov`, `var`

Examples

```
require(graphics)

ma <- cbind(1:6, 1:3)
(S <-  var(ma))
mahalanobis(c(0,0), 1:2, S)

x <- matrix(rnorm(100*3), ncol = 3)
stopifnot(mahalanobis(x, 0, diag(ncol(x))) == rowSums(x*x))
        ##- Here, D^2 = usual squared Euclidean distances

Sx <- cov(x)
D2 <- mahalanobis(x, colMeans(x), Sx)
```

```
plot(density(D2, bw=.5),
     main="Squared Mahalanobis distances, n=100, p=3") ; rug(D2)
qqplot(qchisq(ppoints(100), df=3), D2,
       main = expression("Q-Q plot of Mahalanobis" * ~D^2 *
                         " vs. quantiles of" * ~ chi[3]^2))
abline(0, 1, col = 'gray')
```

`make.link` *Create a Link for GLM Families*

Description

This function is used with the `family` functions in `glm()`. Given the name of a link, it returns a link function, an inverse link function, the derivative $d\mu/d\eta$ and a function for domain checking.

Usage

```
make.link(link)
```

Arguments

`link`	character or numeric; one of `"logit"`, `"probit"`, `"cauchit"`, `"cloglog"`, `"identity"`, `"log"`, `"sqrt"`, `"1/mu^2"`, `"inverse"`, or (deprecated) a non-negative number, say λ resulting in `power` link $= \mu^\lambda$. Also (deprecated) a string like `"power(0.5)"` to indicate a call to `power`.

Value

A object of class `"link-glm"`, a list with components

`linkfun`	Link function `function(mu)`
`linkinv`	Inverse link function `function(eta)`
`mu.eta`	Derivative `function(eta)` $d\mu/d\eta$
`valideta`	`function(eta)` { `TRUE` if `eta` is in the domain of `linkinv` }.
`name`	a name to be used for the link

.

See Also

`power`, `glm`, `family`.

Examples

```
utils::str(make.link("logit"))
```

`makepredictcall` *Utility Function for Safe Prediction*

Description

A utility to help `model.frame.default` create the right matrices when predicting from models with terms like `poly` or `ns`.

Usage

```
makepredictcall(var, call)
```

Arguments

`var`	A variable.
`call`	The term in the formula, as a call.

Details

This is a generic function with methods for `poly`, `bs` and `ns`: the default method handles `scale`. If `model.frame.default` encounters such a term when creating a model frame, it modifies the `predvars` attribute of the terms supplied to replace the term with one that will work for predicting new data. For example `makepredictcall.ns` adds arguments for the knots and intercept.

To make use of this, have your model-fitting function return the `terms` attribute of the model frame, or copy the `predvars` attribute of the `terms` attribute of the model frame to your `terms` object.

To extend this, make sure the term creates variables with a class, and write a suitable method for that class.

Value

A replacement for `call` for the `predvars` attribute of the terms.

See Also

`model.frame`, `poly`, `scale`; `bs` and `ns` in package **splines**, `cars`

Examples

```
require(graphics)

## using poly: this did not work in R < 1.5.0
fm <- lm(weight ~ poly(height, 2), data = women)
plot(women, xlab = "Height (in)", ylab = "Weight (lb)")
ht <- seq(57, 73, len = 200)
lines(ht, predict(fm, data.frame(height=ht)))

## see also example(cars)

## see bs and ns for spline examples.
```

manova *Multivariate Analysis of Variance*

Description

A class for the multivariate analysis of variance.

Usage

```
manova(...)
```

Arguments

`...`	Arguments to be passed to `aov`.

Details

Class `"manova"` differs from class `"aov"` in selecting a different `summary` method. Function `manova` calls `aov` and then add class `"manova"` to the result object for each stratum.

Value

See `aov` and the comments in 'Details' here.

Note

`manova` does not support multistratum analysis of variance, so the formula should not include an `Error` term.

References

Krzanowski, W. J. (1988) *Principles of Multivariate Analysis. A User's Perspective.* Oxford.

Hand, D. J. and Taylor, C. C. (1987) *Multivariate Analysis of Variance and Repeated Measures.* Chapman and Hall.

See Also

`aov`, `summary.manova`, the latter containing examples.

`mantelhaen.test` *Cochran-Mantel-Haenszel Chi-Squared Test for Count Data*

Description

Performs a Cochran-Mantel-Haenszel chi-squared test of the null that two nominal variables are conditionally independent in each stratum, assuming that there is no three-way interaction.

Usage

```
mantelhaen.test(x, y = NULL, z = NULL,
                alternative = c("two.sided", "less", "greater"),
                correct = TRUE, exact = FALSE, conf.level = 0.95)
```

Arguments

`x`	either a 3-dimensional contingency table in array form where each dimension is at least 2 and the last dimension corresponds to the strata, or a factor object with at least 2 levels.
`y`	a factor object with at least 2 levels; ignored if `x` is an array.
`z`	a factor object with at least 2 levels identifying to which stratum the corresponding elements in `x` and `y` belong; ignored if `x` is an array.
`alternative`	indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. You can specify just the initial letter. Only used in the 2 by 2 by K case.
`correct`	a logical indicating whether to apply continuity correction when computing the test statistic. Only used in the 2 by 2 by K case.
`exact`	a logical indicating whether the Mantel-Haenszel test or the exact conditional test (given the strata margins) should be computed. Only used in the 2 by 2 by K case.
`conf.level`	confidence level for the returned confidence interval. Only used in the 2 by 2 by K case.

Details

If `x` is an array, each dimension must be at least 2, and the entries should be nonnegative integers. `NA`'s are not allowed. Otherwise, `x`, `y` and `z` must have the same length. Triples containing `NA`'s are removed. All variables must take at least two different values.

Value

A list with class `"htest"` containing the following components:

`statistic`	Only present if no exact test is performed. In the classical case of a 2 by 2 by K table (i.e., of dichotomous underlying variables), the Mantel-Haenszel chi-squared statistic; otherwise, the generalized Cochran-Mantel-Haenszel statistic.
`parameter`	the degrees of freedom of the approximate chi-squared distribution of the test statistic (1 in the classical case). Only present if no exact test is performed.
`p.value`	the p-value of the test.

`conf.int`	a confidence interval for the common odds ratio. Only present in the 2 by 2 by K case.
`estimate`	an estimate of the common odds ratio. If an exact test is performed, the conditional Maximum Likelihood Estimate is given; otherwise, the Mantel-Haenszel estimate. Only present in the 2 by 2 by K case.
`null.value`	the common odds ratio under the null of independence, `1`. Only present in the 2 by 2 by K case.
`alternative`	a character string describing the alternative hypothesis. Only present in the 2 by 2 by K case.
`method`	a character string indicating the method employed, and whether or not continuity correction was used.
`data.name`	a character string giving the names of the data.

Note

The asymptotic distribution is only valid if there is no three-way interaction. In the classical 2 by 2 by K case, this is equivalent to the conditional odds ratios in each stratum being identical. Currently, no inference on homogeneity of the odds ratios is performed.

See also the example below.

References

Alan Agresti (1990). *Categorical data analysis*. New York: Wiley. Pages 230–235.

Alan Agresti (2002). *Categorical data analysis* (second edition). New York: Wiley.

Examples

```
## Agresti (1990), pages 231--237, Penicillin and Rabbits
## Investigation of the effectiveness of immediately injected or 1.5
##  hours delayed penicillin in protecting rabbits against a lethal
##  injection with beta-hemolytic streptococci.
Rabbits <-
array(c(0, 0, 6, 5,
        3, 0, 3, 6,
        6, 2, 0, 4,
        5, 6, 1, 0,
        2, 5, 0, 0),
      dim = c(2, 2, 5),
      dimnames = list(
          Delay = c("None", "1.5h"),
          Response = c("Cured", "Died"),
          Penicillin.Level = c("1/8", "1/4", "1/2", "1", "4")))
Rabbits
## Classical Mantel-Haenszel test
mantelhaen.test(Rabbits)
## => p = 0.047, some evidence for higher cure rate of immediate
##               injection
## Exact conditional test
mantelhaen.test(Rabbits, exact = TRUE)
## => p - 0.040
## Exact conditional test for one-sided alternative of a higher
## cure rate for immediate injection
mantelhaen.test(Rabbits, exact = TRUE, alternative = "greater")
```

```
## => p = 0.020

## UC Berkeley Student Admissions
mantelhaen.test(UCBAdmissions)
## No evidence for association between admission and gender
## when adjusted for department.  However,
apply(UCBAdmissions, 3, function(x) (x[1,1]*x[2,2])/(x[1,2]*x[2,1]))
## This suggests that the assumption of homogeneous (conditional)
## odds ratios may be violated.  The traditional approach would be
## using the Woolf test for interaction:
woolf <- function(x) {
  x <- x + 1 / 2
  k <- dim(x)[3]
  or <- apply(x, 3, function(x) (x[1,1]*x[2,2])/(x[1,2]*x[2,1]))
  w <-  apply(x, 3, function(x) 1 / sum(1 / x))
  1 - pchisq(sum(w * (log(or) - weighted.mean(log(or), w)) ^ 2), k - 1)
}
woolf(UCBAdmissions)
## => p = 0.003, indicating that there is significant heterogeneity.
## (And hence the Mantel-Haenszel test cannot be used.)

## Agresti (2002), p. 287f and p. 297.
## Job Satisfaction example.
Satisfaction <-
    as.table(array(c(1, 2, 0, 0, 3, 3, 1, 2,
                     11, 17, 8, 4, 2, 3, 5, 2,
                     1, 0, 0, 0, 1, 3, 0, 1,
                     2, 5, 7, 9, 1, 1, 3, 6),
                   dim = c(4, 4, 2),
                   dimnames =
                   list(Income =
                        c("<5000", "5000-15000",
                          "15000-25000", ">25000"),
                        "Job Satisfaction" =
                        c("V_D", "L_S", "M_S", "V_S"),
                        Gender = c("Female", "Male"))))
## (Satisfaction categories abbreviated for convenience.)
ftable(. ~ Gender + Income, Satisfaction)
## Table 7.8 in Agresti (2002), p. 288.
mantelhaen.test(Satisfaction)
## See Table 7.12 in Agresti (2002), p. 297.
```

`mauchly.test` *Mauchly's Test of Sphericity*

Description

Tests whether a Wishart-distributed covariance matrix (or transformation thereof) is proportional to a given matrix.

Usage

```
mauchly.test(object, ...)
## S3 method for class 'mlm':
```

```
mauchly.test(object,...)
## S3 method for class 'SSD':
mauchly.test(object, Sigma = diag(nrow = p),
   T = Thin.row(proj(M) - proj(X)), M = diag(nrow = p), X = ~0,
   idata = data.frame(index = seq_len(p)), ...)
```

Arguments

`object`	object of class `SSD` or `mlm`.
`Sigma`	matrix to be proportional to.
`T`	transformation matrix. By default computed from `M` and `X`.
`M`	formula or matrix describing the outer projection (see below).
`X`	formula or matrix describing the inner projection (see below).
`idata`	data frame describing intra-block design.
`...`	arguments to be passed to or from other methods.

Details

Mauchly's test test for whether a covariance matrix can be assumed to be proportional to a given matrix.

This is a generic function with methods for classes "`mlm`" and "`SSD`".

The basic method is for objects of class `SSD` the method for `mlm` objects just extracts the SSD matrix and invokes the corresponding method with the same options and arguments.

The `T` argument is used to transform the observations prior to testing. This typically involves transformation to intra-block differences, but more complicated within-block designs can be encountered, making more elaborate transformations necessary. A matrix `T` can be given directly or specified as the difference between two projections onto the spaces spanned by `M` and `X`, which in turn can be given as matrices or as model formulas with respect to `idata` (the tests will be invariant to parametrization of the quotient space `M/X`).

The common use of this test is in repeated measurements designs, with `X=~1`. This is almost, but not quite the same as testing for compound symmetry in the untransformed covariance matrix.

Notice that the defaults involve `p`, which is calculated internally as the dimension of the SSD matrix, and a couple of hidden functions in the **stats** name space, namely `proj` which calculates projection matrices from design matrices or model formulas and `Thin.row` which removes linearly dependent rows from a matrix until it has full row rank.

Value

An object of class "`htest`"

Note

The p-value differs slightly from that of SAS because a second order term is included in the asymptotic approximation in R.

References

T. W. Anderson (1958). *An Introduction to Multivariate Statistical Analysis.* Wiley.

See Also

SSD, anova.mlm

Examples

```
utils::example(SSD) # Brings in the mlmfit and reacttime objects

### traditional test of intrasubj. contrasts
mauchly.test(mlmfit, X=~1)

### tests using intra-subject 3x2 design
idata <- data.frame(deg=gl(3,1,6, labels=c(0,4,8)),
                    noise=gl(2,3,6, labels=c("A","P")))
mauchly.test(mlmfit, X = ~ deg + noise, idata = idata)
mauchly.test(mlmfit, M = ~ deg + noise, X = ~ noise, idata=idata)
```

---

mcnemar.test *McNemar's Chi-squared Test for Count Data*

Description

Performs McNemar's chi-squared test for symmetry of rows and columns in a two-dimensional contingency table.

Usage

```
mcnemar.test(x, y = NULL, correct = TRUE)
```

Arguments

x	either a two-dimensional contingency table in matrix form, or a factor object.
y	a factor object; ignored if x is a matrix.
correct	a logical indicating whether to apply continuity correction when computing the test statistic.

Details

The null is that the probabilities of being classified into cells [i,j] and [j,i] are the same.

If x is a matrix, it is taken as a two-dimensional contingency table, and hence its entries should be nonnegative integers. Otherwise, both x and y must be vectors of the same length. Incomplete cases are removed, the vectors are coerced into factor objects, and the contingency table is computed from these.

Continuity correction is only used in the 2-by-2 case if correct is TRUE.

Value

A list with class "htest" containing the following components:

statistic	the value of McNemar's statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	a character string indicating the type of test performed, and whether continuity correction was used.
data.name	a character string giving the name(s) of the data.

References

Alan Agresti (1990). *Categorical data analysis*. New York: Wiley. Pages 350–354.

Examples

```
## Agresti (1990), p. 350.
## Presidential Approval Ratings.
##  Approval of the President's performance in office in two surveys,
##  one month apart, for a random sample of 1600 voting-age Americans.
Performance <-
matrix(c(794, 86, 150, 570),
       nrow = 2,
       dimnames = list("1st Survey" = c("Approve", "Disapprove"),
                       "2nd Survey" = c("Approve", "Disapprove")))
Performance
mcnemar.test(Performance)
## => significant change (in fact, drop) in approval ratings
```

---

median *Median Value*

Description

Compute the sample median.

Usage

```
median(x, na.rm = FALSE)
```

Arguments

x	an object for which a method has been defined, or a numeric vector containing the values whose median is to be computed.
na.rm	a logical value indicating whether NA values should be stripped before the computation proceeds.

Details

This is a generic function for which methods can be written. However, the default method makes use of `sort` and `mean`, both of which are generic, and so the default method will work for most classes (e.g. `"Date"`) for which a median is a reasonable concept.

Value

The default method returns a length-one object of the same type as `x`, except when `x` is integer of even length, when the result will be double.

If there are no values or if `na.rm = FALSE` and there are `NA` values the result is `NA` of the same type as `x` (or more generally the result of `x[FALSE][NA]`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`quantile` for general quantiles.

Examples

```
median(1:4)# = 2.5 [even number]
median(c(1:3,100,1000))# = 3 [odd, robust]
```

`medpolish` *Median Polish of a Matrix*

Description

Fits an additive model using Tukey's *median polish* procedure.

Usage

```
medpolish(x, eps = 0.01, maxiter = 10, trace.iter = TRUE,
          na.rm = FALSE)
```

Arguments

`x`	a numeric matrix.
`eps`	real number greater than 0. A tolerance for convergence: see 'Details'.
`maxiter`	the maximum number of iterations
`trace.iter`	logical. Should progress in convergence be reported?
`na.rm`	logical. Should missing values be removed?

Details

The model fitted is additive (constant + rows + columns). The algorithm works by alternately removing the row and column medians, and continues until the proportional reduction in the sum of absolute residuals is less than `eps` or until there have been `maxiter` iterations. The sum of absolute residuals is printed at each iteration of the fitting process, if `trace.iter` is `TRUE`. If `na.rm` is `FALSE` the presence of any `NA` value in `x` will cause an error, otherwise `NA` values are ignored.

`medpolish` returns an object of class `medpolish` (see below). There are printing and plotting methods for this class, which are invoked via by the generics `print` and `plot`.

Value

An object of class `medpolish` with the following named components:

`overall`	the fitted constant term.
`row`	the fitted row effects.
`col`	the fitted column effects.
`residuals`	the residuals.
`name`	the name of the dataset.

References

Tukey, J. W. (1977). *Exploratory Data Analysis*, Reading Massachusetts: Addison-Wesley.

See Also

`median`; `aov` for a *mean* instead of *median* decomposition.

Examples

```
require(graphics)

## Deaths from sport parachuting;  from ABC of EDA, p.224:
deaths <-
    rbind(c(14,15,14),
          c( 7, 4, 7),
          c( 8, 2,10),
          c(15, 9,10),
          c( 0, 2, 0))
dimnames(deaths) <- list(c("1-24", "25-74", "75-199", "200++", "NA"),
                         paste(1973:1975))
deaths
(med.d <- medpolish(deaths))
plot(med.d)
## Check decomposition:
all(deaths ==
    med.d$overall + outer(med.d$row,med.d$col, "+") + med.d$residuals)
```

`model.extract` *Extract Components from a Model Frame*

Description

Returns the response, offset, subset, weights or other special components of a model frame passed as optional arguments to `model.frame`.

Usage

```
model.extract(frame, component)
model.offset(x)
model.response(data, type = "any")
model.weights(x)
```

Arguments

`frame, x, data`	A model frame.
`component`	literal character string or name. The name of a component to extract, such as `"weights"`, `"subset"`.
`type`	One of `"any"`, `"numeric"`, `"double"`. Using either of latter two coerces the result to have storage mode `"double"`.

Details

`model.extract` is provided for compatibility with S, which does not have the more specific functions. It is also useful to extract e.g. the `etastart` and `mustart` components of a `glm` fit.

`model.offset` and `model.response` are equivalent to `model.extract(, "offset")` and `model.extract(, "response")` respectively. `model.offset` sums any terms specified by `offset` terms in the formula or by `offset` arguments in the call producing the model frame: it does check that the offset is numeric.

`model.weights` is slightly different from `model.frame(, "weights")` in not naming the vector it returns.

Value

The specified component of the model frame, usually a vector.

See Also

`model.frame`, `offset`

Examples

```
a <- model.frame(cbind(ncases,ncontrols) ~ agegp+tobgp+alcgp, data=esoph)
model.extract(a, "response")
stopifnot(model.extract(a, "response") == model.response(a))

a <- model.frame(ncases/(ncases+ncontrols) ~ agegp+tobgp+alcgp,
                 data = esoph, weights = ncases+ncontrols)
```

```
model.response(a)
model.extract(a, "weights")

a <- model.frame(cbind(ncases,ncontrols) ~ agegp,
                 something = tobgp, data = esoph)
names(a)
stopifnot(model.extract(a, "something") == esoph$tobgp)
```

`model.frame` *Extracting the "Environment" of a Model Formula*

Description

`model.frame` (a generic function) and its methods return a `data.frame` with the variables needed to use `formula` and any `...` arguments.

Usage

```
model.frame(formula, ...)

## Default S3 method:
model.frame(formula, data = NULL,
            subset = NULL, na.action = na.fail,
            drop.unused.levels = FALSE, xlev = NULL, ...)

## S3 method for class 'aovlist':
model.frame(formula, data = NULL, ...)

## S3 method for class 'glm':
model.frame(formula, ...)

## S3 method for class 'lm':
model.frame(formula, ...)

get_all_vars(formula, data, ...)
```

Arguments

`formula`	a model `formula` or `terms` object or an R object.
`data`	a data.frame, list or environment (or object coercible by `as.data.frame` to a data.frame), containing the variables in `formula`. Neither a matrix nor an array will be accepted.
`subset`	a specification of the rows to be used: defaults to all rows. This can be any valid indexing vector (see `[.data.frame`) for the rows of `data` or if that is not supplied, a data frame made up of the variables used in `formula`.
`na.action`	how `NA`s are treated. The default is first, any `na.action` attribute of `data`, second a `na.action` setting of `options`, and third `na.fail` if that is unset. The 'factory-fresh' default is `na.omit`. Another possible value is `NULL`.
`drop.unused.levels`	should factors have unused levels dropped? Defaults to `FALSE`.

xlev	a named list of character vectors giving the full set of levels to be assumed for each factor.
...	further arguments such as data, na.action, subset. Any additional arguments such as offset and weights which reach the default method are used to create further columns in the model frame, with parenthesised names such as "(offset)".

Details

Exactly what happens depends on the class and attributes of the object formula. If this is an object of fitted-model class such as "lm", the method will either return the saved model frame used when fitting the model (if any, often selected by argument model = TRUE) or pass the call used when fitting on to the default method. The default method itself can cope with rather standard model objects such as those of class "lqs" from package **MASS** if no other arguments are supplied.

The rest of this section applies only to the default method.

If either formula or data is already a model frame (a data frame with a "terms" attribute) and the other is missing, the model frame is returned. Unless formula is a terms object, as.formula and then terms is called on it. (If you wish to use the keep.order argument of terms.formula, pass a terms object rather than a formula.)

Row names for the model frame are taken from the data argument if present, then from the names of the response in the formula (or rownames if it is a matrix), if there is one.

All the variables in formula, subset and in ... are looked for first in data and then in the environment of formula (see the help for formula() for further details) and collected into a data frame. Then the subset expression is evaluated, and it is used as a row index to the data frame. Then the na.action function is applied to the data frame (and may well add attributes). The levels of any factors in the data frame are adjusted according to the drop.unused.levels and xlev arguments.

Unless na.action = NULL, time-series attributes will be removed from the variables found (since they will be wrong if NAs are removed).

Note that *all* the variables in the formula are included in the data frame, even those preceded by −.

Only variables whose type is raw, logical, integer, real, complex or character can be included in a model frame: this includes classed variables such as factors (whose underlying type is integer), but excludes lists.

get_all_vars returns a data.frame containing the variables used in formula plus those specified .... Unlike model.frame.default, it returns the input variables and not those resulting from function calls in formula.

Value

A data.frame containing the variables used in formula plus those specified in .... It will have additional attributes, including "terms" for an object of class "terms" derived from formula, and possibly "na.action" giving information on the handling of NAs (which will not be present if no special handling was done, e.g. by na.pass).

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

model.matrix for the 'design matrix', formula for formulas and expand.model.frame for model.frame manipulation.

Examples

```
data.class(model.frame(dist ~ speed, data = cars))
```

---

model.matrix *Construct Design Matrices*

Description

model.matrix creates a design matrix.

Usage

```
model.matrix(object, ...)

## Default S3 method:
model.matrix(object, data = environment(object),
             contrasts.arg = NULL, xlev = NULL, ...)
```

Arguments

object	an object of an appropriate class. For the default method, a model formula or terms object.
data	a data frame created with model.frame. If another sort of object, model.frame is called first.
contrasts.arg	A list, whose entries are contrasts suitable for input to the contrasts replacement function and whose names are the names of columns of data containing factors.
xlev	to be used as argument of model.frame if data has no "terms" attribute.
...	further arguments passed to or from other methods.

Details

model.matrix creates a design matrix from the description given in terms(object), using the data in data which must contain variables with the same names as would be created by a call to model.frame(object) or, more precisely, by evaluating attr(terms(object), "variables"). If it is a data frame, there may be other columns and the order of columns is not important. Any character variables are coerced to factors, with a warning. After coercion, all the variables used in RHD of the formula must be logical, integer, numeric or factor.

If contrasts.arg is specified for a factor it overrides the default factor coding for that variable and any "contrasts" attribute set by C or contrasts.

In an interaction term, the variable whose levels vary fastest is the first one to appear in the formula (and not in the term), so in ~ a + b + b:a the interaction will have a varying fastest.

By convention, if the response variable also appears on the right-hand side of the formula it is dropped (with a warning), although interactions involving the term are retained.

Value

The design matrix for a regression model with the specified formula and data.

There is an attribute `"assign"`, an integer vector with an entry for each column in the matrix giving the term in the formula which gave rise to the column. Value `0` corresponds to the intercept (if any), and positive values to terms in the order given by the `terms.labels` attribute of the `terms` structure corresponding to `object`.

If there are any factors in terms in the model, there is an attribute `"contrasts"`, a named list with an entry for each factor. This specifies the contrasts that would be used in terms in which the factor is coded by contrasts (in some terms dummy coding may be used), either as a character vector naming a function or as a numeric matrix.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`model.frame`, `model.extract`, `terms`

Examples

```
ff <- log(Volume) ~ log(Height) + log(Girth)
utils::str(m <- model.frame(ff, trees))
mat <- model.matrix(ff, m)

dd <- data.frame(a = gl(3,4), b = gl(4,1,12))# balanced 2-way
options("contrasts")
model.matrix(~ a + b, dd)
model.matrix(~ a + b, dd, contrasts = list(a="contr.sum"))
model.matrix(~ a + b, dd, contrasts = list(a="contr.sum", b="contr.poly"))
m.orth <- model.matrix(~a+b, dd, contrasts = list(a="contr.helmert"))
crossprod(m.orth)# m.orth is  ALMOST  orthogonal
```

`model.tables` *Compute Tables of Results from an Aov Model Fit*

Description

Computes summary tables for model fits, especially complex `aov` fits.

Usage

```
model.tables(x, ...)

## S3 method for class 'aov':
model.tables(x, type = "effects", se = FALSE, cterms, ...)

## S3 method for class 'aovlist':
model.tables(x, type = "effects", se = FALSE, ...)
```

Arguments

x	a model object, usually produced by aov
type	type of table: currently only "effects" and "means" are implemented.
se	should standard errors be computed?
cterms	A character vector giving the names of the terms for which tables should be computed. The default is all tables.
...	further arguments passed to or from other methods.

Details

For type = "effects" give tables of the coefficients for each term, optionally with standard errors.

For type = "means" give tables of the mean response for each combinations of levels of the factors in a term.

The "aov" method cannot be applied to components of a "aovlist" fit.

Value

An object of class "tables.aov", as list which may contain components

tables	A list of tables for each requested term.
n	The replication information for each term.
se	Standard error information.

Warning

The implementation is incomplete, and only the simpler cases have been tested thoroughly.

Weighted aov fits are not supported.

See Also

aov, proj, replications, TukeyHSD, se.contrast

Examples

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)
options(contrasts=c("contr.helmert", "contr.treatment"))
npk.aov <- aov(yield ~ block + N*P*K, npk)
model.tables(npk.aov, "means", se = TRUE)

## as a test, not particularly sensible statistically
npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
model.tables(npk.aovE, se=TRUE)
model.tables(npk.aovE, "means")
```

`monthplot` *Plot a Seasonal or other Subseries from a Time Series*

Description

These functions plot seasonal (or other) subseries of a time series. For each season (or other category), a time series is plotted.

Usage

```
monthplot(x, ...)

## S3 method for class 'stl':
monthplot(x, labels = NULL, ylab = choice, choice = "seasonal",
          ...)

## S3 method for class 'StructTS':
monthplot(x, labels = NULL, ylab = choice, choice = "sea", ...)

## S3 method for class 'ts':
monthplot(x, labels = NULL, times = time(x), phase = cycle(x),
             ylab = deparse(substitute(x)), ...)

## Default S3 method:
monthplot(x, labels = 1:12,
          ylab = deparse(substitute(x)),
          times = 1:length(x),
          phase = (times - 1)%%length(labels) + 1, base = mean,
          axes = TRUE, type = c("l", "h"), box = TRUE,
          add = FALSE, ...)
```

Arguments

`x`	Time series or related object.
`labels`	Labels to use for each 'season'.
`ylab`	y label.
`times`	Time of each observation.
`phase`	Indicator for each 'season'.
`base`	Function to use for reference line for subseries.
`choice`	Which series of an `stl` or `StructTS` object?
`...`	Arguments to be passed to the default method or graphical parameters.
`axes`	Should axes be drawn (ignored if `add=TRUE`)?
`type`	Type of plot. The default is to join the points with lines, and `"h"` is for histogram-like vertical lines.
`box`	Should a box be drawn (ignored if `add=TRUE`?
`add`	Should thus just add on an existing plot.

Details

These functions extract subseries from a time series and plot them all in one frame. The `ts`, `stl`, and `StructTS` methods use the internally recorded frequency and start and finish times to set the scale and the seasons. The default method assumes observations come in groups of 12 (though this can be changed).

If the `labels` are not given but the `phase` is given, then the `labels` default to the unique values of the `phase`. If both are given, then the `phase` values are assumed to be indices into the `labels` array, i.e., they should be in the range from 1 to `length(labels)`.

Value

These functions are executed for their side effect of drawing a seasonal subseries plot on the current graphical window.

Author(s)

Duncan Murdoch

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`ts`, `stl`, `StructTS`

Examples

```
require(graphics)

## The CO2 data
fit <- stl(log(co2), s.window = 20, t.window = 20)
plot(fit)
op <- par(mfrow = c(2,2))
monthplot(co2, ylab = "data", cex.axis = 0.8)
monthplot(fit, choice = "seasonal", cex.axis = 0.8)
monthplot(fit, choice = "trend", cex.axis = 0.8)
monthplot(fit, choice = "remainder", type = "h", cex.axis = 0.8)
par(op)

## The CO2 data, grouped quarterly
quarter <- (cycle(co2) - 1) %/% 3
monthplot(co2, phase = quarter)

## see also JohnsonJohnson
```

mood.test *Mood Two-Sample Test of Scale*

Description

Performs Mood's two-sample test for a difference in scale parameters.

Usage

```
mood.test(x, ...)

## Default S3 method:
mood.test(x, y,
          alternative = c("two.sided", "less", "greater"), ...)

## S3 method for class 'formula':
mood.test(formula, data, subset, na.action, ...)
```

Arguments

x, y	numeric vectors of data values.
alternative	indicates the alternative hypothesis and must be one of "two.sided" (default), "greater" or "less" all of which can be abbreviated.
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs a factor with two levels giving the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

The underlying model is that the two samples are drawn from $f(x - l)$ and $f((x - l)/s)/s$, respectively, where l is a common location parameter and s is a scale parameter.

The null hypothesis is $s = 1$.

There are more useful tests for this problem.

In the case of ties, the formulation of Mielke (1967) is employed.

Value

A list with class "htest" containing the following components:

statistic	the value of the test statistic.
p.value	the p-value of the test.
alternative	a character string describing the alternative hypothesis.
method	the character string "Mood two-sample test of scale".
data.name	a character string giving the names of the data.

References

William J. Conover (1971), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 234f.

Paul W. Mielke, Jr. (1967), Note on some squared rank tests with existing ties. *Technometrics*, **9**/2, 312–314.

See Also

`fligner.test` for a rank-based (nonparametric) k-sample test for homogeneity of variances; `ansari.test` for another rank-based two-sample test for a difference in scale parameters; `var.test` and `bartlett.test` for parametric tests for the homogeneity in variance.

Examples

```
## Same data as for the Ansari-Bradley test:
## Serum iron determination using Hyland control sera
ramsay <- c(111, 107, 100, 99, 102, 106, 109, 108, 104, 99,
            101, 96, 97, 102, 107, 113, 116, 113, 110, 98)
jung.parekh <- c(107, 108, 106, 98, 105, 103, 110, 105, 104,
            100, 96, 108, 103, 104, 114, 114, 113, 108, 106, 99)
mood.test(ramsay, jung.parekh)
## Compare this to ansari.test(ramsay, jung.parekh)
```

`Multinomial` *The Multinomial Distribution*

Description

Generate multinomially distributed random number vectors and compute multinomial probabilities.

Usage

```
rmultinom(n, size, prob)
dmultinom(x, size = NULL, prob, log = FALSE)
```

Arguments

`x`	vector of length K of integers in `0:size`.
`n`	number of random vectors to draw.
`size`	integer, say N, specifying the total number of objects that are put into K boxes in the typical multinomial experiment. For `dmultinom`, it defaults to `sum(x)`.
`prob`	numeric non-negative vector of length K, specifying the probability for the K classes; is internally normalized to sum 1.
`log`	logical; if TRUE, log probabilities are computed.

Details

If x is a K-component vector, `dmultinom(x, prob)` is the probability

$$P(X_1 = x_1, \ldots, X_K = x_k) = C \times \prod_{j=1}^{K} \pi_j^{x_j}$$

where C is the 'multinomial coefficient' $C = N!/(x_1! \cdots x_K!)$ and $N = \sum_{j=1}^{K} x_j$.
By definition, each component X_j is binomially distributed as `Bin(size, prob[j])` for $j = 1, \ldots, K$.

The `rmultinom()` algorithm draws binomials from $Bin(n_j, P_j)$ sequentially, where $n_1 = N$ (N := `size`), $P_1 = \pi_1$ (π is `prob` scaled to sum 1), and for $j \geq 2$, recursively $n_j = N - \sum_{k=1}^{j-1} n_k$ and $P_j = \pi_j/(1 - \sum_{k=1}^{j-1} \pi_k)$.

Value

For `rmultinom()`, an integer `K x n` matrix where each column is a random vector generated according to the desired multinomial law, and hence summing to `size`. Whereas the *transposed* result would seem more natural at first, the returned matrix is more efficient because of columnwise storage.

Note

`dmultinom` is currently *not vectorized* at all and has no C interface (API); this may be amended in the future.

See Also

`rbinom` which is a special case conceptually.

Examples

```
rmultinom(10, size = 12, prob=c(0.1,0.2,0.8))

pr <- c(1,3,6,10) # normalization not necessary for generation
rmultinom(10, 20, prob = pr)

## all possible outcomes of Multinom(N = 3, K = 3)
X <- t(as.matrix(expand.grid(0:3, 0:3))); X <- X[, colSums(X) <= 3]
X <- rbind(X, 3:3 - colSums(X)); dimnames(X) <- list(letters[1:3], NULL)
X
round(apply(X, 2, function(x) dmultinom(x, prob = c(1,2,5))), 3)
```

---

na.action *NA Action*

Description

Extract information on the NA action used to create an object.

Usage

```
na.action(object, ...)
```

Arguments

`object`	any object whose `NA` action is given.
`...`	further arguments special methods could require.

Details

`na.action` is a generic function, and `na.action.default` its default method. The latter extracts the `"na.action"` component of a list if present, otherwise the `"na.action"` attribute.

When `model.frame` is called, it records any information on `NA` handling in a `"na.action"` attribute. Most model-fitting functions return this as a component of their result.

Value

Information from the action which was applied to `object` if `NA`s were handled specially, or `NULL`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`options("na.action")`, `na.omit`, `na.fail`, also for `na.exclude`, `na.pass`.

Examples

```
na.action(na.omit(c(1, NA)))
```

`na.contiguous` *Find Longest Contiguous Stretch of non-NAs*

Description

Find the longest consecutive stretch of non-missing values in a time series object. (In the event of a tie, the first such stretch.)

Usage

```
na.contiguous(object, ...)
```

Arguments

`object`	a univariate or multivariate time series.
`...`	further arguments passed to or from other methods.

Value

A time series without missing values. The class of `object` will be preserved.

See Also

`na.omit` and `na.omit.ts`; `na.fail`

Examples

```
na.contiguous(presidents)
```

`na.fail` *Handle Missing Values in Objects*

Description

These generic functions are useful for dealing with NAs in e.g., data frames. `na.fail` returns the object if it does not contain any missing values, and signals an error otherwise. `na.omit` returns the object with incomplete cases removed. `na.pass` returns the object unchanged.

Usage

```
na.fail(object, ...)
na.omit(object, ...)
na.exclude(object, ...)
na.pass(object, ...)
```

Arguments

`object`	an R object, typically a data frame
`...`	further arguments special methods could require.

Details

At present these will handle vectors, matrices and data frames comprising vectors and matrices (only).

If `na.omit` removes cases, the row numbers of the cases form the `"na.action"` attribute of the result, of class `"omit"`.

`na.exclude` differs from `na.omit` only in the class of the `"na.action"` attribute of the result, which is `"exclude"`. This gives different behaviour in functions making use of `naresid` and `napredict`: when `na.exclude` is used the residuals and predictions are padded to the correct length by inserting NAs for cases omitted by `na.exclude`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`na.action`; `options` with argument `na.action` for setting NA actions; and `lm` and `glm` for functions using these. `na.contiguous` as alternative for time series.

Examples

```
DF <- data.frame(x = c(1, 2, 3), y = c(0, 10, NA))
na.omit(DF)
m <- as.matrix(DF)
na.omit(m)
stopifnot(all(na.omit(1:3) == 1:3))  # does not affect objects with no NA's
try(na.fail(DF))#> Error: missing values in ...

options("na.action")
```

---

naprint *Adjust for Missing Values*

Description

Use missing value information to report the effects of an `na.action`.

Usage

```
naprint(x, ...)
```

Arguments

`x`	An object produced by an `na.action` function.
`...`	further arguments passed to or from other methods.

Details

This is a generic function, and the exact information differs by method. `naprint.omit` reports the number of rows omitted: `naprint.default` reports an empty string.

Value

A character string providing information on missing values, for example the number.

---

naresid *Adjust for Missing Values*

Description

Use missing value information to adjust residuals and predictions.

Usage

```
naresid(omit, x, ...)
napredict(omit, x, ...)
```

Arguments

omit	an object produced by an na.action function, typically the "na.action" attribute of the result of na.omit or na.exclude.
x	a vector, data frame, or matrix to be adjusted based upon the missing value information.
...	further arguments passed to or from other methods.

Details

These are utility functions used to allow predict, fitted and residuals methods for modelling functions to compensate for the removal of NAs in the fitting process. They are used by the default, "lm", "glm" and "nls" methods, and by further methods in packages **MASS**, **rpart** and **survival**. Also used for the scores returned by factanal, prcomp and princomp.

The default methods do nothing. The default method for the na.exclude action is to pad the object with NAs in the correct positions to have the same number of rows as the original data frame.

Currently naresid and napredict are identical, but future methods need not be. naresid is used for residuals, and napredict for fitted values and predictions.

Value

These return a similar object to x.

Note

Packages rpart and survival5 used to contain versions of these functions that had an na.omit action equivalent to that now used for na.exclude.

NegBinomial *The Negative Binomial Distribution*

Description

Density, distribution function, quantile function and random generation for the negative binomial distribution with parameters size and prob.

Usage

```
dnbinom(x, size, prob, mu, log = FALSE)
pnbinom(q, size, prob, mu, lower.tail = TRUE, log.p = FALSE)
qnbinom(p, size, prob, mu, lower.tail = TRUE, log.p = FALSE)
rnbinom(n, size, prob, mu)
```

Arguments

x	vector of (non-negative integer) quantiles.
q	vector of quantiles.
p	vector of probabilities.
n	number of observations. If length(n) > 1, the length is taken to be the number required.

`size`	target for number of successful trials, or dispersion parameter (the shape parameter of the gamma mixing distribution). Must be strictly positive.
`prob`	probability of success in each trial. `0 < prob <= 1`.
`mu`	alternative parametrization via mean: see 'Details'.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

The negative binomial distribution with `size` $= n$ and `prob` $= p$ has density

$$p(x) = \frac{\Gamma(x+n)}{\Gamma(n)x!}p^n(1-p)^x$$

for $x = 0, 1, 2, \ldots$, $n > 0$ and $0 < p \le 1$.

This represents the number of failures which occur in a sequence of Bernoulli trials before a target number of successes is reached.

A negative binomial distribution can arise as a mixture of Poisson distributions with mean distributed as a Γ (`pgamma`) distribution with scale parameter `(1 - prob)/prob` and shape parameter `size`. (This definition allows non-integer values of `size`.) In this model `prob` = `scale/(1+scale)`, and the mean is `size * (1 - prob)/prob`.

The alternative parametrization (often used in ecology) is by the *mean* `mu`, and `size`, the *dispersion parameter*, where `prob` = `size/(size+mu)`. The variance is `mu + mu^2/size` in this parametrization or $n(1-p)/p^2$ in the first one.

If an element of `x` is not integer, the result of `dnbinom` is zero, with a warning.

The quantile is defined as the smallest value x such that $F(x) \ge p$, where F is the distribution function.

Value

`dnbinom` gives the density, `pnbinom` gives the distribution function, `qnbinom` gives the quantile function, and `rnbinom` generates random deviates.

Invalid `size` or `prob` will result in return value `NaN`, with a warning.

Source

`dnbinom` computes via binomial probabilities, using code contributed by Catherine Loader (see `dbinom`).

`pnbinom` uses `pbeta`.

`qnbinom` uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

`rnbinom` uses the derivation as a gamma mixture of Poissons, see

Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer-Verlag, New York. Page 480.

See Also

`dbinom` for the binomial, `dpois` for the Poisson and `dgeom` for the geometric distribution, which is a special case of the negative binomial.

Examples

```
require(graphics)
x <- 0:11
dnbinom(x, size = 1, prob = 1/2) * 2^(1 + x) # == 1
126 /  dnbinom(0:8, size  = 2, prob  = 1/2) #- theoretically integer

## Cumulative ('p') = Sum of discrete prob.s ('d');  Relative error :
summary(1 - cumsum(dnbinom(x, size = 2, prob = 1/2)) /
                  pnbinom(x, size  = 2, prob = 1/2))

x <- 0:15
size <- (1:20)/4
persp(x,size, dnb <- outer(x, size, function(x,s) dnbinom(x,s, prob= 0.4)),
      xlab = "x", ylab = "s", zlab="density", theta = 150)
title(tit <- "negative binomial density(x,s, pr = 0.4)  vs.  x & s")

image  (x,size, log10(dnb), main= paste("log [",tit,"]"))
contour(x,size, log10(dnb),add=TRUE)

## Alternative parametrization
x1 <- rnbinom(500, mu = 4, size = 1)
x2 <- rnbinom(500, mu = 4, size = 10)
x3 <- rnbinom(500, mu = 4, size = 100)
h1 <- hist(x1, breaks = 20, plot = FALSE)
h2 <- hist(x2, breaks = h1$breaks, plot = FALSE)
h3 <- hist(x3, breaks = h1$breaks, plot = FALSE)
barplot(rbind(h1$counts, h2$counts, h3$counts),
        beside = TRUE, col = c("red","blue","cyan"),
        names.arg = round(h1$breaks[-length(h1$breaks)]))
```

---

nextn *Highly Composite Numbers*

Description

`nextn` returns the smallest integer, greater than or equal to `n`, which can be obtained as a product of powers of the values contained in `factors`. `nextn` is intended to be used to find a suitable length to zero-pad the argument of `fft` to so that the transform is computed quickly. The default value for `factors` ensures this.

Usage

```
nextn(n, factors = c(2,3,5))
```

Arguments

`n`	an integer.
`factors`	a vector of positive integer factors.

See Also

`convolve`, `fft`.

Examples

```
nextn(1001) # 1024
table(sapply(599:630, nextn))
```

---

nlm *Non-Linear Minimization*

Description

This function carries out a minimization of the function `f` using a Newton-type algorithm. See the references for details.

Usage

```
nlm(f, p, ..., hessian = FALSE, typsize = rep(1, length(p)),
    fscale = 1, print.level = 0, ndigit = 12, gradtol = 1e-6,
    stepmax = max(1000 * sqrt(sum((p/typsize)^2)), 1000),
    steptol = 1e-6, iterlim = 100, check.analyticals = TRUE)
```

Arguments

`f`	the function to be minimized. If the function value has an attribute called `gradient` or both `gradient` and `hessian` attributes, these will be used in the calculation of updated parameter values. Otherwise, numerical derivatives are used. `deriv` returns a function with suitable `gradient` attribute. This should be a function of a vector of the length of `p` followed by any other arguments specified by the `...` argument.
`p`	starting parameter values for the minimization.
`...`	additional arguments to `f`.
`hessian`	if `TRUE`, the hessian of `f` at the minimum is returned.
`typsize`	an estimate of the size of each parameter at the minimum.
`fscale`	an estimate of the size of `f` at the minimum.
`print.level`	this argument determines the level of printing which is done during the minimization process. The default value of `0` means that no printing occurs, a value of `1` means that initial and final details are printed and a value of 2 means that full tracing information is printed.
`ndigit`	the number of significant digits in the function `f`.
`gradtol`	a positive scalar giving the tolerance at which the scaled gradient is considered close enough to zero to terminate the algorithm. The scaled gradient is a measure of the relative change in `f` in each direction `p[i]` divided by the relative change in `p[i]`.
`stepmax`	a positive scalar which gives the maximum allowable scaled step length. `stepmax` is used to prevent steps which would cause the optimization function to overflow, to prevent the algorithm from leaving the area of interest in parameter space, or to detect divergence in the algorithm. `stepmax` would be chosen small enough to prevent the first two of these occurrences, but should be larger than any anticipated reasonable step.
`steptol`	A positive scalar providing the minimum allowable relative step length.

`iterlim` a positive integer specifying the maximum number of iterations to be performed before the program is terminated.

`check.analyticals` a logical scalar specifying whether the analytic gradients and Hessians, if they are supplied, should be checked against numerical derivatives at the initial parameter values. This can help detect incorrectly formulated gradients or Hessians.

Details

Note that arguments after `...` must be matched exactly.

If a gradient or hessian is supplied but evaluates to the wrong mode or length, it will be ignored if `check.analyticals = TRUE` (the default) with a warning. The hessian is not even checked unless the gradient is present and passes the sanity checks.

From the three methods available in the original source, we always use method "1" which is line search.

The functions supplied must always return finite (including not `NA` and not `NaN`) values.

Value

A list containing the following components:

`minimum` the value of the estimated minimum of `f`.

`estimate` the point at which the minimum value of `f` is obtained.

`gradient` the gradient at the estimated minimum of `f`.

`hessian` the hessian at the estimated minimum of `f` (if requested).

`code` an integer indicating why the optimization process terminated.

- **1:** relative gradient is close to zero, current iterate is probably solution.
- **2:** successive iterates within tolerance, current iterate is probably solution.
- **3:** last global step failed to locate a point lower than `estimate`. Either `estimate` is an approximate local minimum of the function or `steptol` is too small.
- **4:** iteration limit exceeded.
- **5:** maximum step size `stepmax` exceeded five consecutive times. Either the function is unbounded below, becomes asymptotic to a finite value from above in some direction or `stepmax` is too small.

`iterations` the number of iterations performed.

References

Dennis, J. E. and Schnabel, R. B. (1983) *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, Englewood Cliffs, NJ.

Schnabel, R. B., Koontz, J. E. and Weiss, B. E. (1985) A modular system of algorithms for unconstrained minimization. *ACM Trans. Math. Software*, **11**, 419–440.

See Also

`optim` and `nlminb`.

`constrOptim` for constrained optimization, `optimize` for one-dimensional minimization and `uniroot` for root finding. `deriv` to calculate analytical derivatives.

For nonlinear regression, `nls` may be better.

Examples

```
f <- function(x) sum((x-1:length(x))^2)
nlm(f, c(10,10))
nlm(f, c(10,10), print.level = 2)
utils::str(nlm(f, c(5), hessian = TRUE))

f <- function(x, a) sum((x-a)^2)
nlm(f, c(10,10), a=c(3,5))
f <- function(x, a)
{
    res <- sum((x-a)^2)
    attr(res, "gradient") <- 2*(x-a)
    res
}
nlm(f, c(10,10), a=c(3,5))

## more examples, including the use of derivatives.
## Not run: demo(nlm)
```

---

nlminb *Optimization using PORT routines*

Description

Unconstrained and constrained optimization using PORT routines.

Usage

```
nlminb(start, objective, gradient = NULL, hessian = NULL, ...,
       scale = 1, control = list(), lower = -Inf, upper = Inf)
```

Arguments

`start`	numeric vector, initial values for the parameters to be optimized.
`objective`	Function to be minimized. Must return a scalar value (possibly NA/Inf). The first argument to `objective` is the vector of parameters to be optimized, whose initial values are supplied through `start`. Further arguments (fixed during the course of the optimization) to `objective` may be specified as well (see `...`).
`gradient`	Optional function that takes the same arguments as `objective` and evaluates the gradient of `objective` at its first argument. Must return a vector as long as `start`.
`hessian`	Optional function that takes the same arguments as `objective` and evaluates the hessian of `objective` at its first argument. Must return a square matrix of order `length(start)`. Only the lower triangle is used.
`...`	Further arguments to be supplied to `objective`.
`scale`	See PORT documentation (or leave alone).
`control`	A list of control parameters. See below for details.
`lower, upper`	vectors of lower and upper bounds, replicated to be as long as `start`. If unspecified, all parameters are assumed to be unconstrained.

Details

Any names of `start` are (as from R 2.8.1) passed on to `objective` and where applicable, `gradient` and `hessian`. The parameter vector will be coerced to double.

The PORT documentation is at http://netlib.bell-labs.com/cm/cs/cstr/153.pdf.

Value

A list with components:

`par`	The best set of parameters found.
`objective`	The value of `objective` corresponding to `par`.
`convergence`	An integer code. `0` indicates successful convergence.
`message`	A character string giving any additional information returned by the optimizer, or `NULL`. For details, see PORT documentation.
`iterations`	Number of iterations performed.
`evaluations`	Number of objective function and gradient function evaluations

Control parameters

Possible names in the `control` list and their default values are:

`eval.max` Maximum number of evaluations of the objective function allowed. Defaults to 200.

`iter.max` Maximum number of iterations allowed. Defaults to 150.

`trace` The value of the objective function and the parameters is printed every trace'th iteration. Defaults to 0 which indicates no trace information is to be printed.

`abs.tol` Absolute tolerance. Defaults to `1e-20`.

`rel.tol` Relative tolerance. Defaults to `1e-10`.

`x.tol` X tolerance. Defaults to `1.5e-8`.

`step.min` Minimum step size. Defaults to `2.2e-14`.

Author(s)

(of R port) Douglas Bates and Deepayan Sarkar.

References

http://netlib.bell-labs.com/netlib/port/

See Also

`optim` and `nlm`.

`optimize` for one-dimensional minimization and `constrOptim` for constrained optimization.

Examples

```
x <- rnbinom(100, mu = 10, size = 10)
hdev <- function(par) {
    -sum(dnbinom(x, mu = par[1], size = par[2], log = TRUE))
}
nlminb(c(9, 12), hdev)
nlminb(c(20, 20), hdev, lower = 0, upper = Inf)
nlminb(c(20, 20), hdev, lower = 0.001, upper = Inf)

## slightly modified from the S-PLUS help page for nlminb
# this example minimizes a sum of squares with known solution y
sumsq <- function( x, y) {sum((x-y)^2)}
y <- rep(1,5)
x0 <- rnorm(length(y))
nlminb(start = x0, sumsq, y = y)
# now use bounds with a y that has some components outside the bounds
y <- c( 0, 2, 0, -2, 0)
nlminb(start = x0, sumsq, lower = -1, upper = 1, y = y)
# try using the gradient
sumsq.g <- function(x,y) 2*(x-y)
nlminb(start = x0, sumsq, sumsq.g,
       lower = -1, upper = 1, y = y)
# now use the hessian, too
sumsq.h <- function(x,y) diag(2, nrow = length(x))
nlminb(start = x0, sumsq, sumsq.g, sumsq.h,
       lower = -1, upper = 1, y = y)

## Rest lifted from optim help page

fr <- function(x) {   ## Rosenbrock Banana function
    x1 <- x[1]
    x2 <- x[2]
    100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
grr <- function(x) { ## Gradient of 'fr'
    x1 <- x[1]
    x2 <- x[2]
    c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
       200 *      (x2 - x1 * x1))
}
nlminb(c(-1.2,1), fr)
nlminb(c(-1.2,1), fr, grr)

flb <- function(x)
    { p <- length(x); sum(c(1, rep(4, p-1)) * (x - c(1, x[-p])^2)^2) }
## 25-dimensional box constrained
## par[24] is *not* at boundary
nlminb(rep(3, 25), flb,
          lower=rep(2, 25),
          upper=rep(4, 25))
```

`nls` *Nonlinear Least Squares*

Description

Determine the nonlinear (weighted) least-squares estimates of the parameters of a nonlinear model.

Usage

```
nls(formula, data, start, control, algorithm,
    trace, subset, weights, na.action, model,
    lower, upper, ...)
```

Arguments

formula	a nonlinear model formula including variables and parameters. Will be coerced to a formula if necessary.
data	an optional data frame in which to evaluate the variables in formula and weights. Can also be a list or an environment, but not a matrix.
start	a named list or named numeric vector of starting estimates. When start is missing, a very cheap guess for start is tried (if algorithm != "plinear").
control	an optional list of control settings. See nls.control for the names of the settable control values and their effect.
algorithm	character string specifying the algorithm to use. The default algorithm is a Gauss-Newton algorithm. Other possible values are "plinear" for the Golub-Pereyra algorithm for partially linear least-squares models and "port" for the 'nl2sol' algorithm from the Port library – see the references.
trace	logical value indicating if a trace of the iteration progress should be printed. Default is FALSE. If TRUE the residual (weighted) sum-of-squares and the parameter values are printed at the conclusion of each iteration. When the "plinear" algorithm is used, the conditional estimates of the linear parameters are printed after the nonlinear parameters. When the "port" algorithm is used the objective function value printed is half the residual (weighted) sum-of-squares.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
weights	an optional numeric vector of (fixed) weights. When present, the objective function is weighted least squares.
na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The 'factory-fresh' default is na.omit. Value na.exclude can be useful.
model	logical. If true, the model frame is returned as part of the object. Default is FALSE.
lower, upper	vectors of lower and upper bounds, replicated to be as long as start. If unspecified, all parameters are assumed to be unconstrained. Bounds can only be used with the "port" algorithm. They are ignored, with a warning, if given for other algorithms.
...	Additional optional arguments. None are used at present.

Details

An nls object is a type of fitted model object. It has methods for the generic functions anova, coef, confint, deviance, df.residual, fitted, formula, logLik, predict, print, profile, residuals, summary, vcov and weights.

Variables in formula (and weights if not missing) are looked for first in data, then the environment of formula and finally along the search path. Functions in formula are searched for first in the environment of formula and then along the search path.

Arguments subset and na.action are supported only when all the variables in the formula taken from data are of the same length: other cases give a warning.

Note that the anova method does not check that the models are nested: this cannot easily be done automatically, so use with care.

Value

A list of

m	an nlsModel object incorporating the model.
data	the expression that was passed to nls as the data argument. The actual data values are present in the environment of the m component.
call	the matched call with several components, notably algorithm.
na.action	the "na.action" attribute (if any) of the model frame.
dataClasses	the "dataClasses" attribute (if any) of the "terms" attribute of the model frame.
model	if model = TRUE, the model frame.
weights	if weights is supplied, the weights.
convInfo	when algorithm is not "port", a list with convergence information.
control	the control list used, see the control argument.
convergence, message	for an algorithm = "port" fit only, a convergence code (0 for convergence) and message.

Note that setting warnOnly = TRUE in the control argument (see nls.control) returns a non-converged object (since R version 2.5.0) which might be useful for further convergence analysis, *but **not** for inference*.

Warning

Do not use nls on artificial "zero-residual" data.

The nls function uses a relative-offset convergence criterion that compares the numerical imprecision at the current parameter estimates to the residual sum-of-squares. This performs well on data of the form

$$y = f(x, \theta) + \epsilon$$

(with var(eps) > 0). It fails to indicate convergence on data of the form

$$y = f(x, \theta)$$

because the criterion amounts to comparing two components of the round-off error. If you wish to test nls on artificial data please add a noise component, as shown in the example below.

The algorithm = "port" code appears unfinished, and does not even check that the starting value is within the bounds. Use with caution, especially where bounds are supplied.

Author(s)

Douglas M. Bates and Saikat DebRoy

References

Bates, D. M. and Watts, D. G. (1988) *Nonlinear Regression Analysis and Its Applications*, Wiley

Bates, D. M. and Chambers, J. M. (1992) *Nonlinear models.* Chapter 10 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

http://www.netlib.org/port/ for the Port library documentation.

See Also

summary.nls, predict.nls, profile.nls.

Examples

```
require(graphics)

DNase1 <- subset(DNase, Run == 1)

## using a selfStart model
fm1DNase1 <- nls(density ~ SSlogis(log(conc), Asym, xmid, scal), DNase1)
summary(fm1DNase1)
## the coefficients only:
coef(fm1DNase1)
## including their SE, etc:
coef(summary(fm1DNase1))

## using conditional linearity
fm2DNase1 <- nls(density ~ 1/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(xmid = 0, scal = 1),
                 algorithm = "plinear", trace = TRUE)
summary(fm2DNase1)

## without conditional linearity
fm3DNase1 <- nls(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(Asym = 3, xmid = 0, scal = 1),
                 trace = TRUE)
summary(fm3DNase1)

## using Port's nl2sol algorithm
fm4DNase1 <- nls(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(Asym = 3, xmid = 0, scal = 1),
                 trace = TRUE, algorithm = "port")
summary(fm4DNase1)

## weighted nonlinear regression
Treated <- Puromycin[Puromycin$state == "treated", ]
weighted.MM <- function(resp, conc, Vm, K)
{
    ## Purpose: exactly as white book p. 451 -- RHS for nls()
    ##  Weighted version of Michaelis-Menten model
```

```
    ## ------------------------------------------------------------
    ## Arguments: 'y', 'x' and the two parameters (see book)
    ## ------------------------------------------------------------
    ## Author: Martin Maechler, Date: 23 Mar 2001

    pred <- (Vm * conc)/(K + conc)
    (resp - pred) / sqrt(pred)
}

Pur.wt <- nls( ~ weighted.MM(rate, conc, Vm, K), data = Treated,
              start = list(Vm = 200, K = 0.1),
              trace = TRUE)
summary(Pur.wt)

## Passing arguments using a list that can not be coerced to a data.frame
lisTreat <- with(Treated,
                 list(conc1 = conc[1], conc.1 = conc[-1], rate = rate))

weighted.MM1 <- function(resp, conc1, conc.1, Vm, K)
{
     conc <- c(conc1, conc.1)
     pred <- (Vm * conc)/(K + conc)
    (resp - pred) / sqrt(pred)
}
Pur.wt1 <- nls( ~ weighted.MM1(rate, conc1, conc.1, Vm, K),
               data = lisTreat, start = list(Vm = 200, K = 0.1))
stopifnot(all.equal(coef(Pur.wt), coef(Pur.wt1)))

## Chambers and Hastie (1992) Statistical Models in S  (p. 537):
## If the value of the right side [of formula] has an attribute called
## 'gradient' this should be a matrix with the number of rows equal
## to the length of the response and one column for each parameter.

weighted.MM.grad <- function(resp, conc1, conc.1, Vm, K)
{
  conc <- c(conc1, conc.1)

  K.conc <- K+conc
  dy.dV <- conc/K.conc
  dy.dK <- -Vm*dy.dV/K.conc
  pred <- Vm*dy.dV
  pred.5 <- sqrt(pred)
  dev <- (resp - pred) / pred.5
  Ddev <- -0.5*(resp+pred)/(pred.5*pred)
  attr(dev, "gradient") <- Ddev * cbind(Vm = dy.dV, K = dy.dK)
  dev
}

Pur.wt.grad <- nls( ~ weighted.MM.grad(rate, conc1, conc.1, Vm, K),
                   data = lisTreat, start = list(Vm = 200, K = 0.1))

rbind(coef(Pur.wt), coef(Pur.wt1), coef(Pur.wt.grad))

## In this example, there seems no advantage to providing the gradient.
## In other cases, there might be.

## The two examples below show that you can fit a model to
```

```
## artificial data with noise but not to artificial data
## without noise.
x <- 1:10
y <- 2*x + 3                              # perfect fit
yeps <- y + rnorm(length(y), sd = 0.01) # added noise
nls(yeps ~ a + b*x, start = list(a = 0.12345, b = 0.54321),
    trace = TRUE)
## Not run:
## terminates in an error, because convergence cannot be confirmed:
nls(y ~ a + b*x, start = list(a = 0.12345, b = 0.54321),
     trace = TRUE)
## End(Not run)

## the nls() internal cheap guess for starting values can be sufficient:

x <- -(1:100)/10
y <- 100 + 10 * exp(x / 2) + rnorm(x)/10
nlmod <- nls(y ~  Const + A * exp(B * x), trace=TRUE)

plot(x,y, main = "nls(*), data, true function and fit, n=100")
curve(100 + 10 * exp(x / 2), col=4, add = TRUE)
lines(x, predict(nlmod), col=2)

## The muscle dataset in MASS is from an experiment on muscle
## contraction on 21 animals.  The observed variables are Strip
## (identifier of muscle), Conc (Cacl concentration) and Length
## (resulting length of muscle section).
utils::data(muscle, package = "MASS")

## The non linear model considered is
##       Length = alpha + beta*exp(-Conc/theta) + error
## where theta is constant but alpha and beta may vary with Strip.

with(muscle, table(Strip)) # 2,3 or 4 obs per strip

## We first use the plinear algorithm to fit an overall model,
## ignoring that alpha and beta might vary with Strip.

musc.1 <- nls(Length ~ cbind(1, exp(-Conc/th)), muscle,
              start = list(th=1), algorithm="plinear")
summary(musc.1)

## Then we use nls' indexing feature for parameters in non-linear
## models to use the conventional algorithm to fit a model in which
## alpha and beta vary with Strip.  The starting values are provided
## by the previously fitted model.
## Note that with indexed parameters, the starting values must be
## given in a list (with names):
b <- coef(musc.1)
musc.2 <- nls(Length ~ a[Strip] + b[Strip]*exp(-Conc/th),
              muscle,
              start = list(a=rep(b[2],21), b=rep(b[3],21), th=b[1]))
summary(musc.2)
```

`nls.control` *Control the Iterations in nls*

Description

Allow the user to set some characteristics of the `nls` nonlinear least squares algorithm.

Usage

```
nls.control(maxiter = 50, tol = 1e-05, minFactor = 1/1024,
            printEval = FALSE, warnOnly = FALSE)
```

Arguments

`maxiter`	A positive integer specifying the maximum number of iterations allowed.
`tol`	A positive numeric value specifying the tolerance level for the relative offset convergence criterion.
`minFactor`	A positive numeric value specifying the minimum step-size factor allowed on any step in the iteration. The increment is calculated with a Gauss-Newton algorithm and successively halved until the residual sum of squares has been decreased or until the step-size factor has been reduced below this limit.
`printEval`	a logical specifying whether the number of evaluations (steps in the gradient direction taken each iteration) is printed.
`warnOnly`	a logical specifying whether `nls()` should return instead of signalling an error in the case of termination before convergence. Termination before convergence happens upon completion of `maxiter` iterations, in the case of a singular gradient, and in the case that the step-size factor is reduced below `minFactor`.

Value

A `list` with exactly five components:

`maxiter`

`tol`

`minFactor`

`printEval`

`warnOnly`

with meanings as explained under 'Arguments'.

Author(s)

Douglas Bates and Saikat DebRoy

References

Bates and Watts (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley.

See Also

`nls`

Examples

```
nls.control(minFactor = 1/2048)
```

NLSstAsymptotic *Fit the Asymptotic Regression Model*

Description

Fits the asymptotic regression model, in the form `b0 + b1*(1-exp(-exp(lrc) * x)` to the `xy` data. This can be used as a building block in determining starting estimates for more complicated models.

Usage

```
NLSstAsymptotic(xy)
```

Arguments

`xy`	a `sortedXyData` object

Value

A numeric value of length 3 with components labelled `b0`, `b1`, and `lrc`. `b0` is the estimated intercept on the `y`-axis, `b1` is the estimated difference between the asymptote and the `y`-intercept, and `lrc` is the estimated logarithm of the rate constant.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`SSasymp`

Examples

```
Lob.329 <- Loblolly[ Loblolly$Seed == "329", ]
NLSstAsymptotic(sortedXyData(expression(age), expression(height), Lob.329))
```

NLSstClosestX *Inverse Interpolation*

Description

Use inverse linear interpolation to approximate the `x` value at which the function represented by `xy` is equal to `yval`.

Usage

```
NLSstClosestX(xy, yval)
```

Arguments

xy	a sortedXyData object
yval	a numeric value on the y scale

Value

A single numeric value on the x scale.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

sortedXyData, NLSstLfAsymptote, NLSstRtAsymptote, selfStart

Examples

```
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData(expression(log(conc)), expression(density), DNase.2)
NLSstClosestX(DN.srt, 1.0)
```

NLSstLfAsymptote *Horizontal Asymptote on the Left Side*

Description

Provide an initial guess at the horizontal asymptote on the left side (i.e., small values of x) of the graph of y versus x from the xy object. Primarily used within initial functions for self-starting nonlinear regression models.

Usage

```
NLSstLfAsymptote(xy)
```

Arguments

xy	a sortedXyData object

Value

A single numeric value estimating the horizontal asymptote for small x.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

sortedXyData, NLSstClosestX, NLSstRtAsymptote, selfStart

Examples

```
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData( expression(log(conc)), expression(density), DNase.2 )
NLSstLfAsymptote( DN.srt )
```

NLSstRtAsymptote *Horizontal Asymptote on the Right Side*

Description

Provide an initial guess at the horizontal asymptote on the right side (i.e., large values of `x`) of the graph of `y` versus `x` from the `xy` object. Primarily used within `initial` functions for self-starting nonlinear regression models.

Usage

```
NLSstRtAsymptote(xy)
```

Arguments

`xy`	a `sortedXyData` object

Value

A single numeric value estimating the horizontal asymptote for large `x`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`sortedXyData`, `NLSstClosestX`, `NLSstRtAsymptote`, `selfStart`

Examples

```
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData( expression(log(conc)), expression(density), DNase.2 )
NLSstRtAsymptote( DN.srt )
```

`Normal` *The Normal Distribution*

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to `mean` and standard deviation equal to `sd`.

Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Arguments

`x,q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`mean`	vector of means.
`sd`	vector of standard deviations.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `mean` or `sd` are not specified they assume the default values of `0` and `1`, respectively.

The normal distribution has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$$

where μ is the mean of the distribution and σ the standard deviation.

`qnorm` is based on Wichura's algorithm AS 241 which provides precise results up to about 16 digits.

Value

`dnorm` gives the density, `pnorm` gives the distribution function, `qnorm` gives the quantile function, and `rnorm` generates random deviates.

Source

For `pnorm`, based on

Cody, W. D. (1993) Algorithm 715: SPECFUN – A portable FORTRAN package of special function routines and test drivers. *ACM Transactions on Mathematical Software* **19**, 22–32.

For `qnorm`, the code is a C translation of

Wichura, M. J. (1988) Algorithm AS 241: The Percentage Points of the Normal Distribution. *Applied Statistics*, **37**, 477–484.

For `rnorm`, see RNG for how to select the algorithm and for references to the supplied methods.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 13. Wiley, New York.

See Also

`runif` and `.Random.seed` about random number generation, and `dlnorm` for the *Log*normal distribution.

Examples

```
require(graphics)

dnorm(0) == 1/ sqrt(2*pi)
dnorm(1) == exp(-1/2)/ sqrt(2*pi)
dnorm(1) == 1/ sqrt(2*pi*exp(1))

## Using "log = TRUE" for an extended range :
par(mfrow=c(2,1))
plot(function(x) dnorm(x, log=TRUE), -60, 50,
     main = "log { Normal density }")
curve(log(dnorm(x)), add=TRUE, col="red",lwd=2)
mtext("dnorm(x, log=TRUE)", adj=0)
mtext("log(dnorm(x))", col="red", adj=1)

plot(function(x) pnorm(x, log.p=TRUE), -50, 10,
     main = "log { Normal Cumulative }")
curve(log(pnorm(x)), add=TRUE, col="red",lwd=2)
mtext("pnorm(x, log=TRUE)", adj=0)
mtext("log(pnorm(x))", col="red", adj=1)

## if you want the so-called 'error function'
erf <- function(x) 2 * pnorm(x * sqrt(2)) - 1
## (see Abramowitz and Stegun 29.2.29)
## and the so-called 'complementary error function'
erfc <- function(x) 2 * pnorm(x * sqrt(2), lower = FALSE)
## and the inverses
erfinv <- function (x) qnorm((1 + x)/2)/sqrt(2)
erfcinv <- function (x) qnorm(x/2, lower = FALSE)/sqrt(2)
```

numericDeriv *Evaluate derivatives numerically*

Description

numericDeriv numerically evaluates the gradient of an expression.

Usage

```
numericDeriv(expr, theta, rho = parent.frame(), dir = 1.0)
```

Arguments

expr	The expression to be differentiated. The value of this expression should be a numeric vector.
theta	A character vector of names of numeric variables used in expr.
rho	An environment containing all the variables needed to evaluate expr.
dir	A numeric vector of directions to use for the finite differences.

Details

This is a front end to the C function numeric_deriv, which is described in *Writing R Extensions*.

The numeric variables must be of type real and not integer.

Value

The value of eval(expr, envir = rho) plus a matrix attribute called gradient. The columns of this matrix are the derivatives of the value with respect to the variables listed in theta.

Author(s)

Saikat DebRoy ⟨saikat@stat.wisc.edu⟩

Examples

```
myenv <- new.env()
assign("mean", 0., envir = myenv)
assign("sd", 1., envir = myenv)
assign("x", seq(-3., 3., len = 31), envir = myenv)
numericDeriv(quote(pnorm(x, mean, sd)), c("mean", "sd"), myenv)
```

offset *Include an Offset in a Model Formula*

Description

An offset is a term to be added to a linear predictor, such as in a generalised linear model, with known coefficient 1 rather than an estimated coefficient.

Usage

```
offset(object)
```

Arguments

object	An offset to be included in a model frame

Details

There can be more than one offset in a model formula, but - is not supported for `offset` terms (and is equivalent to +).

Value

The input value.

See Also

`model.offset`, `model.frame`.

For examples see `glm` and `Insurance` in package **MASS**.

oneway.test *Test for Equal Means in a One-Way Layout*

Description

Test whether two or more samples from normal distributions have the same means. The variances are not necessarily assumed to be equal.

Usage

```
oneway.test(formula, data, subset, na.action, var.equal = FALSE)
```

Arguments

formula	a formula of the form lhs ~ rhs where lhs gives the sample values and rhs the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
var.equal	a logical variable indicating whether to treat the variances in the samples as equal. If TRUE, then a simple F test for the equality of means in a one-way analysis of variance is performed. If FALSE, an approximate method of Welch (1951) is used, which generalizes the commonly known 2-sample Welch test to the case of arbitrarily many samples.

Value

A list with class "htest" containing the following components:

statistic	the value of the test statistic.
parameter	the degrees of freedom of the exact or approximate F distribution of the test statistic.
p.value	the p-value of the test.
method	a character string indicating the test performed.
data.name	a character string giving the names of the data.

References

B. L. Welch (1951), On the comparison of several mean values: an alternative approach. *Biometrika*, **38**, 330–336.

See Also

The standard t test (t.test) as the special case for two samples; the Kruskal-Wallis test kruskal.test for a nonparametric test for equal location parameters in a one-way layout.

Examples

```
## Not assuming equal variances
oneway.test(extra ~ group, data = sleep)
## Assuming equal variances
oneway.test(extra ~ group, data = sleep, var.equal = TRUE)
## which gives the same result as
anova(lm(extra ~ group, data = sleep))
```

`optim` *General-purpose Optimization*

Description

General-purpose optimization based on Nelder–Mead, quasi-Newton and conjugate-gradient algorithms. It includes an option for box-constrained optimization and simulated annealing.

Usage

```
optim(par, fn, gr = NULL, ...,
      method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN"),
      lower = -Inf, upper = Inf,
      control = list(), hessian = FALSE)
```

Arguments

`par`	Initial values for the parameters to be optimized over.
`fn`	A function to be minimized (or maximized), with first argument the vector of parameters over which minimization is to take place. It should return a scalar result.
`gr`	A function to return the gradient for the `"BFGS"`, `"CG"` and `"L-BFGS-B"` methods. If it is `NULL`, a finite-difference approximation will be used. For the `"SANN"` method it specifies a function to generate a new candidate point. If it is `NULL` a default Gaussian Markov kernel is used.
`...`	Further arguments to be passed to `fn` and `gr`.
`method`	The method to be used. See 'Details'.
`lower, upper`	Bounds on the variables for the `"L-BFGS-B"` method.
`control`	A list of control parameters. See 'Details'.
`hessian`	Logical. Should a numerically differentiated Hessian matrix be returned?

Details

Note that arguments after `...` must be matched exactly.

By default this function performs minimization, but it will maximize if `control$fnscale` is negative.

The default method is an implementation of that of Nelder and Mead (1965), that uses only function values and is robust but relatively slow. It will work reasonably well for non-differentiable functions.

Method `"BFGS"` is a quasi-Newton method (also known as a variable metric algorithm), specifically that published simultaneously in 1970 by Broyden, Fletcher, Goldfarb and Shanno. This uses function values and gradients to build up a picture of the surface to be optimized.

Method `"CG"` is a conjugate gradients method based on that by Fletcher and Reeves (1964) (but with the option of Polak–Ribiere or Beale–Sorenson updates). Conjugate gradient methods will generally be more fragile than the BFGS method, but as they do not store a matrix they may be successful in much larger optimization problems.

Method `"L-BFGS-B"` is that of Byrd *et. al.* (1995) which allows *box constraints*, that is each variable can be given a lower and/or upper bound. The initial value must satisfy the constraints.

This uses a limited-memory modification of the BFGS quasi-Newton method. If non-trivial bounds are supplied, this method will be selected, with a warning.

Nocedal and Wright (1999) is a comprehensive reference for the previous three methods.

Method `"SANN"` is by default a variant of simulated annealing given in Belisle (1992). Simulated-annealing belongs to the class of stochastic global optimization methods. It uses only function values but is relatively slow. It will also work for non-differentiable functions. This implementation uses the Metropolis function for the acceptance probability. By default the next candidate point is generated from a Gaussian Markov kernel with scale proportional to the actual temperature. If a function to generate a new candidate point is given, method `"SANN"` can also be used to solve combinatorial optimization problems. Temperatures are decreased according to the logarithmic cooling schedule as given in Belisle (1992, p. 890); specifically, the temperature is set to `temp / log(((t-1) %/% tmax)*tmax + exp(1))`, where `t` is the current iteration step and `temp` and `tmax` are specifiable via `control`, see below. Note that the `"SANN"` method depends critically on the settings of the control parameters. It is not a general-purpose method but can be very useful in getting to a good value on a very rough surface.

Function `fn` can return `NA` or `Inf` if the function cannot be evaluated at the supplied value, but the initial value must have a computable finite value of `fn`. (Except for method `"L-BFGS-B"` where the values should always be finite.)

`optim` can be used recursively, and for a single parameter as well as many. It also accepts a zero-length `par`, and just evaluates the function with that argument.

The `control` argument is a list that can supply any of the following components:

`trace` Non-negative integer. If positive, tracing information on the progress of the optimization is produced. Higher values may produce more tracing information: for method `"L-BFGS-B"` there are six levels of tracing. (To understand exactly what these do see the source code: higher levels give more detail.)

`fnscale` An overall scaling to be applied to the value of `fn` and `gr` during optimization. If negative, turns the problem into a maximization problem. Optimization is performed on `fn(par)/fnscale`.

`parscale` A vector of scaling values for the parameters. Optimization is performed on `par/parscale` and these should be comparable in the sense that a unit change in any element produces about a unit change in the scaled value.

`ndeps` A vector of step sizes for the finite-difference approximation to the gradient, on `par/parscale` scale. Defaults to `1e-3`.

`maxit` The maximum number of iterations. Defaults to `100` for the derivative-based methods, and `500` for `"Nelder-Mead"`. For `"SANN"` `maxit` gives the total number of function evaluations. There is no other stopping criterion. Defaults to `10000`.

`abstol` The absolute convergence tolerance. Only useful for non-negative functions, as a tolerance for reaching zero.

`reltol` Relative convergence tolerance. The algorithm stops if it is unable to reduce the value by a factor of `reltol * (abs(val) + reltol)` at a step. Defaults to `sqrt(.Machine$double.eps)`, typically about `1e-8`.

`alpha, beta, gamma` Scaling parameters for the `"Nelder-Mead"` method. `alpha` is the reflection factor (default 1.0), `beta` the contraction factor (0.5) and `gamma` the expansion factor (2.0).

`REPORT` The frequency of reports for the `"BFGS"`, `"L-BFGS-B"` and `"SANN"` methods if `control$trace` is positive. Defaults to every 10 iterations for `"BFGS"` and `"L-BFGS-B"`, or every 100 temperatures for `"SANN"`.

type for the conjugate-gradients method. Takes value 1 for the Fletcher–Reeves update, 2 for Polak–Ribiere and 3 for Beale–Sorenson.

lmm is an integer giving the number of BFGS updates retained in the "L-BFGS-B" method, It defaults to 5.

factr controls the convergence of the "L-BFGS-B" method. Convergence occurs when the reduction in the objective is within this factor of the machine tolerance. Default is 1e7, that is a tolerance of about 1e-8.

pgtol helps control the convergence of the "L-BFGS-B" method. It is a tolerance on the projected gradient in the current search direction. This defaults to zero, when the check is suppressed.

temp controls the "SANN" method. It is the starting temperature for the cooling schedule. Defaults to 10.

tmax is the number of function evaluations at each temperature for the "SANN" method. Defaults to 10.

Any names given to par will be copied to the vectors passed to fn and gr. Note that no other attributes of par are copied over.

Value

A list with components:

par	The best set of parameters found.
value	The value of fn corresponding to par.
counts	A two-element integer vector giving the number of calls to fn and gr respectively. This excludes those calls needed to compute the Hessian, if requested, and any calls to fn to compute a finite-difference approximation to the gradient.
convergence	An integer code. 0 indicates successful convergence. Error codes are **1** indicates that the iteration limit maxit had been reached. **10** indicates degeneracy of the Nelder–Mead simplex. **51** indicates a warning from the "L-BFGS-B" method; see component message for further details. **52** indicates an error from the "L-BFGS-B" method; see component message for further details.
message	A character string giving any additional information returned by the optimizer, or NULL.
hessian	Only if argument hessian is true. A symmetric matrix giving an estimate of the Hessian at the solution found. Note that this is the Hessian of the unconstrained problem even if the box constraints are active.

Note

optim will work with one-dimensional pars, but the default method does not work well (and will warn). Use optimize instead.

Source

The code for methods "Nelder-Mead", "BFGS" and "CG" was based originally on Pascal code in Nash (1990) that was translated by p2c and then hand-optimized. Dr Nash has agreed that the code can be made freely available.

The code for method "L-BFGS-B" is based on Fortran code by Zhu, Byrd, Lu-Chen and Nocedal obtained from Netlib (file 'opt/lbfgs_bcm.shar': another version is in 'toms/778').

The code for method "SANN" was contributed by A. Trapletti.

References

Belisle, C. J. P. (1992) Convergence theorems for a class of simulated annealing algorithms on R^d. *J Applied Probability*, **29**, 885–895.

Byrd, R. H., Lu, P., Nocedal, J. and Zhu, C. (1995) A limited memory algorithm for bound constrained optimization. *SIAM J. Scientific Computing*, **16**, 1190–1208.

Fletcher, R. and Reeves, C. M. (1964) Function minimization by conjugate gradients. *Computer Journal* **7**, 148–154.

Nash, J. C. (1990) *Compact Numerical Methods for Computers. Linear Algebra and Function Minimisation.* Adam Hilger.

Nelder, J. A. and Mead, R. (1965) A simplex algorithm for function minimization. *Computer Journal* **7**, 308–313.

Nocedal, J. and Wright, S. J. (1999) *Numerical Optimization*. Springer.

See Also

nlm, nlminb.

optimize for one-dimensional minimization and constrOptim for constrained optimization.

Examples

```
require(graphics)

fr <- function(x) {   ## Rosenbrock Banana function
    x1 <- x[1]
    x2 <- x[2]
    100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
grr <- function(x) { ## Gradient of 'fr'
    x1 <- x[1]
    x2 <- x[2]
    c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
       200 *      (x2 - x1 * x1))
}
optim(c(-1.2,1), fr)
optim(c(-1.2,1), fr, grr, method = "BFGS")
optim(c(-1.2,1), fr, NULL, method = "BFGS", hessian = TRUE)
optim(c(-1.2,1), fr, grr, method = "CG")
optim(c(-1.2,1), fr, grr, method = "CG", control=list(type=2))
optim(c(-1.2,1), fr, grr, method = "L-BFGS-B")

flb <- function(x)
    { p <- length(x); sum(c(1, rep(4, p-1)) * (x - c(1, x[-p])^2)^2) }
## 25-dimensional box constrained
```

```
optim(rep(3, 25), flb, NULL, method = "L-BFGS-B",
      lower=rep(2, 25), upper=rep(4, 25)) # par[24] is *not* at boundary

## "wild" function , global minimum at about -15.81515
fw <- function (x)
    10*sin(0.3*x)*sin(1.3*x^2) + 0.00001*x^4 + 0.2*x+80
plot(fw, -50, 50, n=1000, main = "optim() minimising 'wild function'")

res <- optim(50, fw, method="SANN",
             control=list(maxit=20000, temp=20, parscale=20))
res
## Now improve locally {typically only by a small bit}:
(r2 <- optim(res$par, fw, method="BFGS"))
points(r2$par, r2$value, pch = 8, col = "red", cex = 2)

## Combinatorial optimization: Traveling salesman problem
library(stats) # normally loaded

eurodistmat <- as.matrix(eurodist)

distance <- function(sq) {  # Target function
    sq2 <- embed(sq, 2)
    return(sum(eurodistmat[cbind(sq2[,2],sq2[,1])]))
}

genseq <- function(sq) {  # Generate new candidate sequence
    idx <- seq(2, NROW(eurodistmat)-1, by=1)
    changepoints <- sample(idx, size=2, replace=FALSE)
    tmp <- sq[changepoints[1]]
    sq[changepoints[1]] <- sq[changepoints[2]]
    sq[changepoints[2]] <- tmp
    return(sq)
}

sq <- c(1,2:NROW(eurodistmat),1)  # Initial sequence
distance(sq)

set.seed(123) # chosen to get a good soln relatively quickly
res <- optim(sq, distance, genseq, method="SANN",
             control = list(maxit=30000, temp=2000, trace=TRUE, REPORT=500))
res  # Near optimum distance around 12842

loc <- cmdscale(eurodist)
rx <- range(x <- loc[,1])
ry <- range(y <- -loc[,2])
tspinit <- loc[sq,]
tspres <- loc[res$par,]
s <- seq(NROW(tspres)-1)

plot(x, y, type="n", asp=1, xlab="", ylab="",
     main="initial solution of traveling salesman problem")
arrows(tspinit[s,1], -tspinit[s,2], tspinit[s+1,1], -tspinit[s+1,2],
       angle=10, col="green")
text(x, y, labels(eurodist), cex=0.8)

plot(x, y, type="n", asp=1, xlab="", ylab="",
     main="optim() 'solving' traveling salesman problem")
```

```
arrows(tspres[s,1], -tspres[s,2], tspres[s+1,1], -tspres[s+1,2],
       angle=10, col="red")
text(x, y, labels(eurodist), cex=0.8)
```

`optimize` *One Dimensional Optimization*

Description

The function `optimize` searches the interval from `lower` to `upper` for a minimum or maximum of the function `f` with respect to its first argument.

`optimise` is an alias for `optimize`.

Usage

```
optimize(f = , interval = ,  ..., lower = min(interval),
         upper = max(interval), maximum = FALSE,
         tol = .Machine$double.eps^0.25)
optimise(f = , interval = ,  ..., lower = min(interval),
         upper = max(interval), maximum = FALSE,
         tol = .Machine$double.eps^0.25)
```

Arguments

`f`	the function to be optimized. The function is either minimized or maximized over its first argument depending on the value of `maximum`.
`interval`	a vector containing the end-points of the interval to be searched for the minimum.
`...`	additional named or unnamed arguments to be passed to `f`.
`lower`	the lower end point of the interval to be searched.
`upper`	the upper end point of the interval to be searched.
`maximum`	logical. Should we maximize or minimize (the default)?
`tol`	the desired accuracy.

Details

Note that arguments after `...` must be matched exactly.

The method used is a combination of golden section search and successive parabolic interpolation, and was designed for use with continuous functions. Convergence is never much slower than that for a Fibonacci search. If `f` has a continuous second derivative which is positive at the minimum (which is not at `lower` or `upper`), then convergence is superlinear, and usually of the order of about 1.324.

The function `f` is never evaluated at two points closer together than $\epsilon|x_0| + (tol/3)$, where ϵ is approximately `sqrt(.Machine$double.eps)` and x_0 is the final abscissa `optimize()$minimum`.
If `f` is a unimodal function and the computed values of `f` are always unimodal when separated by at least $\epsilon\,|x| + (tol/3)$, then x_0 approximates the abscissa of the global minimum of `f` on the interval `lower,upper` with an error less than $\epsilon|x_0| + tol$.

If `f` is not unimodal, then `optimize()` may approximate a local, but perhaps non-global, minimum to the same accuracy.

The first evaluation of `f` is always at $x_1 = a+(1-\phi)(b-a)$ where `(a,b) = (lower, upper)` and $\phi = (\sqrt{5}-1)/2 = 0.61803..$ is the golden section ratio. Almost always, the second evaluation is at $x_2 = a + \phi(b-a)$. Note that a local minimum inside $[x_1, x_2]$ will be found as solution, even when `f` is constant in there, see the last example.

`f` will be called as `f(x, ...)` for a numeric value of x.

Value

A list with components `minimum` (or `maximum`) and `objective` which give the location of the minimum (or maximum) and the value of the function at that point.

Source

A C translation of Fortran code http://www.netlib.org/fmm/fmin.f based on the Algol 60 procedure `localmin` given in the reference.

References

Brent, R. (1973) *Algorithms for Minimization without Derivatives.* Englewood Cliffs N.J.: Prentice-Hall.

See Also

`nlm`, `uniroot`.

Examples

```
require(graphics)

f <- function (x,a) (x-a)^2
xmin <- optimize(f, c(0, 1), tol = 0.0001, a = 1/3)
xmin

## See where the function is evaluated:
optimize(function(x) x^2*(print(x)-1), lower=0, upper=10)

## "wrong" solution with unlucky interval and piecewise constant f():
f  <- function(x) ifelse(x > -1, ifelse(x < 4, exp(-1/abs(x - 1)), 10), 10)
fp <- function(x) { print(x); f(x) }

plot(f, -2,5, ylim = 0:1, col = 2)
optimize(fp, c(-4, 20))# doesn't see the minimum
optimize(fp, c(-7, 20))# ok
```

`order.dendrogram` *Ordering or Labels of the Leaves in a Dendrogram*

Description

Theses functions return the order (index) or the `"label"` attribute for the leaves in a dendrogram. These indices can then be used to access the appropriate components of any additional data.

Usage

```
order.dendrogram(x)

## S3 method for class 'dendrogram':
labels(object, ...)
```

Arguments

`x, object`	a dendrogram (see `as.dendrogram`).
`...`	additional arguments

Details

The indices or labels for the leaves in left to right order are retrieved.

Value

A vector with length equal to the number of leaves in the dendrogram is returned. From `r <- order.dendrogram()`, each element is the index into the original data (from which the dendrogram was computed).

Author(s)

R. Gentleman (`order.dendrogram` and Martin Maechler (`labels.dendrogram`).

See Also

`reorder`, `dendrogram`.

Examples

```
set.seed(123)
x <- rnorm(10)
hc <- hclust(dist(x))
hc$order
dd <- as.dendrogram(hc)
order.dendrogram(dd) ## the same :
stopifnot(hc$order == order.dendrogram(dd))

d2 <- as.dendrogram(hclust(dist(USArrests)))
labels(d2) ## in this case the same as
stopifnot(labels(d2) == rownames(USArrests)[order.dendrogram(d2)])
```

`p.adjust` *Adjust P-values for Multiple Comparisons*

Description

Given a set of p-values, returns p-values adjusted using one of several methods.

Usage

```
p.adjust(p, method = p.adjust.methods, n = length(p))

p.adjust.methods
# c("holm", "hochberg", "hommel", "bonferroni", "BH", "BY",
#   "fdr", "none")
```

Arguments

`p`	vector of p-values (possibly with `NA`s).
`method`	correction method
`n`	number of comparisons, must be at least `length(p)`; only set this (to non-default) when you know what you are doing!

Details

The adjustment methods include the Bonferroni correction (`"bonferroni"`) in which the p-values are multiplied by the number of comparisons. Less conservative corrections are also included by Holm (1979) (`"holm"`), Hochberg (1988) (`"hochberg"`), Hommel (1988) (`"hommel"`), Benjamini & Hochberg (1995) (`"BH"`), and Benjamini & Yekutieli (2001) (`"BY"`), respectively. A pass-through option (`"none"`) is also included. The set of methods are contained in the `p.adjust.methods` vector for the benefit of methods that need to have the method as an option and pass it on to `p.adjust`.

The first four methods are designed to give strong control of the family wise error rate. There seems no reason to use the unmodified Bonferroni correction because it is dominated by Holm's method, which is also valid under arbitrary assumptions.

Hochberg's and Hommel's methods are valid when the hypothesis tests are independent or when they are non-negatively associated (Sarkar, 1998; Sarkar and Chang, 1997). Hommel's method is more powerful than Hochberg's, but the difference is usually small and the Hochberg p-values are faster to compute.

The `"BH"` and `"BY"` method of Benjamini, Hochberg, and Yekutieli control the false discovery rate, the expected proportion of false discoveries amongst the rejected hypotheses. The false discovery rate is a less stringent condition than the family wise error rate, so these methods are more powerful than the others.

Note that you can set `n` larger than `length(p)` which means the unobserved p-values are assumed to be greater than all the observed p for `"bonferroni"` and `"holm"` methods and equal to 1 for the other methods.

Value

A vector of corrected p-values (same length as `p`).

References

Benjamini, Y., and Hochberg, Y. (1995). Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society Series* B, **57**, 289–300.

Benjamini, Y., and Yekutieli, D. (2001). The control of the false discovery rate in multiple testing under dependency. *Annals of Statistics* **29**, 1165–1188.

Holm, S. (1979). A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, **6**, 65–70.

Hommel, G. (1988). A stagewise rejective multiple test procedure based on a modified Bonferroni test. *Biometrika*, **75**, 383–386.

Hochberg, Y. (1988). A sharper Bonferroni procedure for multiple tests of significance. *Biometrika*, **75**, 800–803.

Shaffer, J. P. (1995). Multiple hypothesis testing. *Annual Review of Psychology*, **46**, 561–576. (An excellent review of the area.)

Sarkar, S. (1998). Some probability inequalities for ordered MTP2 random variables: a proof of Simes conjecture. *Annals of Statistics*, **26**, 494–504.

Sarkar, S., and Chang, C. K. (1997). Simes' method for multiple hypothesis testing with positively dependent test statistics. *Journal of the American Statistical Association*, **92**, 1601–1608.

Wright, S. P. (1992). Adjusted P-values for simultaneous inference. *Biometrics*, **48**, 1005–1013. (Explains the adjusted P-value approach.)

See Also

`pairwise.*` functions such as `pairwise.t.test`.

Examples

```
require(graphics)

set.seed(123)
x <- rnorm(50, mean=c(rep(0,25),rep(3,25)))
p <- 2*pnorm( sort(-abs(x)))

round(p, 3)
round(p.adjust(p), 3)
round(p.adjust(p,"BH"), 3)

## or all of them at once (dropping the "fdr" alias):
p.adjust.M <- p.adjust.methods[p.adjust.methods != "fdr"]
p.adj <- sapply(p.adjust.M, function(meth) p.adjust(p, meth))
round(p.adj, 3)
## or a bit nicer:
noquote(apply(p.adj, 2, format.pval, digits = 3))

## and a graphic:
matplot(p, p.adj, ylab="p.adjust(p, meth)", type = "l", asp=1, lty=1:6,
        main = "P-value adjustments")
legend(.7,.6, p.adjust.M, col=1:6, lty=1:6)

## Can work with NA's:
pN <- p; iN <- c(46,47); pN[iN] <- NA
pN.a <- sapply(p.adjust.M, function(meth) p.adjust(pN, meth))
## The smallest 20 P-values all affected by the NA's :
```

```
round((pN.a / p.adj)[1:20, ] , 4)
```

---

pairwise.prop.test *Pairwise comparisons for proportions*

Description

Calculate pairwise comparisons between pairs of proportions with correction for multiple testing

Usage

```
pairwise.prop.test(x, n, p.adjust.method = p.adjust.methods, ...)
```

Arguments

x	Vector of counts of successes or a matrix with 2 columns giving the counts of successes and failures, respectively.
n	Vector of counts of trials; ignored if x is a matrix.
p.adjust.method	Method for adjusting p values (see p.adjust)
...	Additional arguments to pass to prop.test

Value

Object of class "pairwise.htest"

See Also

prop.test, p.adjust

Examples

```
smokers  <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
pairwise.prop.test(smokers, patients)
```

---

pairwise.t.test *Pairwise t tests*

Description

Calculate pairwise comparisons between group levels with corrections for multiple testing

Usage

```
pairwise.t.test(x, g, p.adjust.method = p.adjust.methods,
                pool.sd = !paired, paired = FALSE,
                alternative = c("two.sided", "less", "greater"),  ...)
```

Arguments

x	response vector.
g	grouping vector or factor.
p.adjust.method	Method for adjusting p values (see p.adjust).
pool.sd	switch to allow/disallow the use of a pooled SD
paired	a logical indicating whether you want paired t-tests.
alternative	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less".
...	additional arguments to pass to t.test.

Details

The pool.SD switch calculates a common SD for all groups and used that for all comparisons (this can be useful if some groups are small). This method does not actually call t.test, so extra arguments are ignored. Pooling does not generalize to paired tests so pool.SD and paired cannot both be TRUE.

Only the lower triangle of the matrix of possible comparisons is being calculated, so setting alternative to anything other than "two.sided" requires that the levels of g are ordered sensibly.

Value

Object of class "pairwise.htest"

See Also

t.test, p.adjust

Examples

```
attach(airquality)
Month <- factor(Month, labels = month.abb[5:9])
pairwise.t.test(Ozone, Month)
pairwise.t.test(Ozone, Month, p.adj = "bonf")
pairwise.t.test(Ozone, Month, pool.sd = FALSE)
detach()
```

---

pairwise.table *Tabulate p values for pairwise comparisons*

Description

Creates table of p values for pairwise comparisons with corrections for multiple testing.

Usage

```
pairwise.table(compare.levels, level.names, p.adjust.method)
```

Arguments

compare.levels	Function to compute (raw) p value given indices i and j
level.names	Names of the group levels
p.adjust.method	Method for multiple testing adjustment

Details

Functions that do multiple group comparisons create separate compare.levels functions (assumed to be symmetrical in i and j) and passes them to this function.

Value

Table of p values in lower triangular form.

See Also

pairwise.t.test, et al.

---

pairwise.wilcox.test

Pairwise Wilcoxon rank sum tests

Description

Calculate pairwise comparisons between group levels with corrections for multiple testing.

Usage

```
pairwise.wilcox.test(x, g, p.adjust.method = p.adjust.methods,
                     paired=FALSE, ...)
```

Arguments

x	response vector.
g	grouping vector or factor.
p.adjust.method	method for adjusting p values (see p.adjust).
paired	a logical indicating whether you want a paired test.
...	additional arguments to pass to wilcox.test.

Details

Extra arguments that are passed on to wilcox.test may or may not be sensible in this context. In particular, only the lower triangle of the matrix of possible comparisons is being calculated, so setting alternative to anything other than "two.sided" requires that the levels of g are ordered sensibly.

Value

Object of class "pairwise.htest"

See Also

wilcox.test, p.adjust

Examples

```
attach(airquality)
Month <- factor(Month, labels = month.abb[5:9])
## These give warnings because of ties :
pairwise.wilcox.test(Ozone, Month)
pairwise.wilcox.test(Ozone, Month, p.adj = "bonf")
detach()
```

---

plot.acf *Plot Autocovariance and Autocorrelation Functions*

Description

Plot method for objects of class "acf".

Usage

```
## S3 method for class 'acf':
plot(x, ci = 0.95, type = "h", xlab = "Lag", ylab = NULL,
     ylim = NULL, main = NULL,
     ci.col = "blue", ci.type = c("white", "ma"),
     max.mfrow = 6, ask = Npgs > 1 && dev.interactive(),
     mar = if(nser > 2) c(3,2,2,0.8) else par("mar"),
     oma = if(nser > 2) c(1,1.2,1,1) else par("oma"),
     mgp = if(nser > 2) c(1.5,0.6,0) else par("mgp"),
     xpd = par("xpd"),
     cex.main = if(nser > 2) 1 else par("cex.main"),
     verbose = getOption("verbose"),
     ...)
```

Arguments

x	an object of class "acf".
ci	coverage probability for confidence interval. Plotting of the confidence interval is suppressed if ci is zero or negative.
type	the type of plot to be drawn, default to histogram like vertical lines.
xlab	the x label of the plot.
ylab	the y label of the plot.
ylim	numeric of length 2 giving the y limits for the plot.
main	overall title for the plot.
ci.col	colour to plot the confidence interval lines.

`ci.type`	should the confidence limits assume a white noise input or for lag k an MA($k-1$) input?
`max.mfrow`	positive integer; for multivariate `x` indicating how many rows and columns of plots should be put on one page, using `par(mfrow = c(m,m))`.
`ask`	logical; if `TRUE`, the user is asked before a new page is started.
`mar, oma, mgp, xpd, cex.main`	graphics parameters as in `par(*)`, by default adjusted to use smaller than default margins for multivariate `x` only.
`verbose`	logical. Should R report extra information on progress?
`...`	graphics parameters to be passed to the plotting routines.

Note

The confidence interval plotted in `plot.acf` is based on an *uncorrelated* series and should be treated with appropriate caution. Using `ci.type = "ma"` may be less potentially misleading.

See Also

`acf` which calls `plot.acf` by default.

Examples

```
require(graphics)

z4  <- ts(matrix(rnorm(400), 100, 4), start=c(1961, 1), frequency=12)
z7  <- ts(matrix(rnorm(700), 100, 7), start=c(1961, 1), frequency=12)
acf(z4)
acf(z7, max.mfrow = 7)# squeeze on 1 page
acf(z7) # multi-page
```

`plot.density` *Plot Method for Kernel Density Estimation*

Description

The `plot` method for density objects.

Usage

```
## S3 method for class 'density':
plot(x, main = NULL, xlab = NULL, ylab = "Density", type = "l",
     zero.line = TRUE, ...)
```

Arguments

`x`	a `"density"` object.
`main, xlab, ylab, type`	plotting parameters with useful defaults.
`...`	further plotting parameters.
`zero.line`	logical; if `TRUE`, add a base line at $y = 0$

Value

None.

See Also

density.

`plot.HoltWinters` *Plot function for HoltWinters objects*

Description

Produces a chart of the original time series along with the fitted values. Optionally, predicted values (and their confidence bounds) can also be plotted.

Usage

```
## S3 method for class 'HoltWinters':
plot(x, predicted.values = NA, intervals = TRUE,
        separator = TRUE, col = 1, col.predicted = 2,
        col.intervals = 4, col.separator = 1, lty = 1,
        lty.predicted = 1, lty.intervals = 1, lty.separator = 3,
        ylab = "Observed / Fitted",
        main = "Holt-Winters filtering",
        ylim = NULL, ...)
```

Arguments

`x`	Object of class `"HoltWinters"`
`predicted.values`	Predicted values as returned by `predict.HoltWinters`
`intervals`	If `TRUE`, the prediction intervals are plotted (default).
`separator`	If `TRUE`, a separating line between fitted and predicted values is plotted (default).
`col, lty`	Color/line type of original data (default: black solid).
`col.predicted, lty.predicted`	Color/line type of fitted and predicted values (default: red solid).
`col.intervals, lty.intervals`	Color/line type of prediction intervals (default: blue solid).
`col.separator, lty.separator`	Color/line type of observed/predicted values separator (default: black dashed).
`ylab`	Label of the y-axis.
`main`	Main title.
`ylim`	Limits of the y-axis. If `NULL`, the range is chosen such that the plot contains the original series, the fitted values, and the predicted values if any.
`...`	Other graphics parameters.

Author(s)

David Meyer ⟨David.Meyer@wu-wien.ac.at⟩

References

C. C. Holt (1957) Forecasting seasonals and trends by exponentially weighted moving averages, ONR Research Memorandum, Carnigie Institute 52.

P. R. Winters (1960) Forecasting sales by exponentially weighted moving averages, *Management Science* **6**, 324–342.

See Also

`HoltWinters`, `predict.HoltWinters`

`plot.isoreg` *Plot Method for isoreg Objects*

Description

The `plot` and `lines` method for R objects of class `isoreg`.

Usage

```
## S3 method for class 'isoreg':
plot(x, plot.type = c("single", "row.wise", "col.wise"),
      main = paste("Isotonic regression", deparse(x$call)),
      main2 = "Cumulative Data and Convex Minorant",
      xlab = "x0", ylab = "x$y",
      par.fit = list(col = "red", cex = 1.5, pch = 13, lwd = 1.5),
      mar = if (both) 0.1 + c(3.5, 2.5, 1, 1) else par("mar"),
      mgp = if (both) c(1.6, 0.7, 0) else par("mgp"),
      grid = length(x$x) < 12, ...)
## S3 method for class 'isoreg':
lines(x, col = "red", lwd = 1.5,
       do.points = FALSE, cex = 1.5, pch = 13, ...)
```

Arguments

`x`	an `isoreg` object.
`plot.type`	character indicating which type of plot is desired. The first (default) only draws the data and the fit, where the others add a plot of the cumulative data and fit.
`main`	main title of plot, see `title`.
`main2`	title for second (cumulative) plot.
`xlab, ylab`	x- and y- axis annotation.
`par.fit`	a `list` of arguments (for `points` and `lines`) for drawing the fit.
`mar, mgp`	graphical parameters, see `par`, mainly for the case of two plots.

`grid` logical indicating if grid lines should be drawn. If true, `grid()` is used for the first plot, where as vertical lines are drawn at 'touching' points for the cumulative plot.

`do.points` for `lines()`: logical indicating if the step points should be drawn as well (and as they are drawn in `plot()`).

`col, lwd, cex, pch` graphical arguments for `lines()`, where `cex` and `pch` are only used when `do.points` is `TRUE`.

`...` further arguments passed to and from methods.

See Also

`isoreg` for computation of `isoreg` objects.

Examples

```
require(graphics)

utils::example(isoreg) # for the examples there

plot(y3, main = "simple plot(.)  +  lines(<isoreg>)")
lines(ir3)

## 'same' plot as above, "proving" that only ranks of 'x' are important
plot(isoreg(2^(1:9), c(1,0,4,3,3,5,4,2,0)), plot.type = "row", log = "x")

plot(ir3, plot.type = "row", ylab = "y3")
plot(isoreg(y3 - 4), plot.t="r", ylab = "y3 - 4")
plot(ir4, plot.type = "ro",  ylab = "y4", xlab = "x = 1:n")

## experiment a bit with these (C-c C-j):
plot(isoreg(sample(9),  y3), plot.type="row")
plot(isoreg(sample(9),  y3), plot.type="col.wise")

plot(ir <- isoreg(sample(10), sample(10, replace = TRUE)),
                  plot.type = "r")
```

`plot.lm` *Plot Diagnostics for an lm Object*

Description

Six plots (selectable by `which`) are currently available: a plot of residuals against fitted values, a Scale-Location plot of $\sqrt{|residuals|}$ against fitted values, a Normal Q-Q plot, a plot of Cook's distances versus row labels, a plot of residuals against leverages, and a plot of Cook's distances against leverage/(1-leverage). By default, the first three and `5` are provided.

Usage

```
## S3 method for class 'lm':
plot(x, which = c(1:3,5),
     caption = list("Residuals vs Fitted", "Normal Q-Q",
```

```
        "Scale-Location", "Cook's distance",
        "Residuals vs Leverage",
        expression("Cook's dist vs Leverage  " * h[ii] / (1 - h[ii]))),
    panel = if(add.smooth) panel.smooth else points,
    sub.caption = NULL, main = "",
    ask = prod(par("mfcol")) < length(which) && dev.interactive(),
    ...,
    id.n = 3, labels.id = names(residuals(x)), cex.id = 0.75,
    qqline = TRUE, cook.levels = c(0.5, 1.0),
    add.smooth = getOption("add.smooth"), label.pos = c(4,2),
    cex.caption = 1)
```

Arguments

`x`	`lm` object, typically result of `lm` or `glm`.
`which`	if a subset of the plots is required, specify a subset of the numbers `1:6`.
`caption`	captions to appear above the plots; `character` vector or `list` of valid graphics annotations, see `as.graphicsAnnot`. Can be set to `""` or `NA` to suppress all captions.
`panel`	panel function. The useful alternative to `points`, `panel.smooth` can be chosen by `add.smooth = TRUE`.
`sub.caption`	common title—above figures if there are multiple; used as `sub` (s.`title`) otherwise. If `NULL`, as by default, a possible shortened version of `deparse(x$call)` is used.
`main`	title to each plot—in addition to the above `caption`.
`ask`	logical; if `TRUE`, the user is *ask*ed before each plot, see `par(ask=.)`.
`...`	other parameters to be passed through to plotting functions.
`id.n`	number of points to be labelled in each plot, starting with the most extreme.
`labels.id`	vector of labels, from which the labels for extreme points will be chosen. `NULL` uses observation numbers.
`cex.id`	magnification of point labels.
`qqline`	logical indicating if a `qqline()` should be added to the normal Q-Q plot.
`cook.levels`	levels of Cook's distance at which to draw contours.
`add.smooth`	logical indicating if a smoother should be added to most plots; see also `panel` above.
`label.pos`	positioning of labels, for the left half and right half of the graph respectively, for plots 1-3.
`cex.caption`	controls the size of `caption`.

Details

`sub.caption`—by default the function call—is shown as a subtitle (under the x-axis title) on each plot when plots are on separate pages, or as a subtitle in the outer margin (if any) when there are multiple plots per page.

The 'Scale-Location' plot, also called 'Spread-Location' or 'S-L' plot, takes the square root of the absolute residuals in order to diminish skewness ($\sqrt{|E|}$ is much less skewed than $|E|$ for Gaussian zero-mean E).

The 'S-L', the Q-Q, and the Residual-Leverage plot, use *standardized* residuals which have identical variance (under the hypothesis). They are given as $R_i/(s \times \sqrt{1 - h_{ii}})$ where h_{ii} are the diagonal entries of the hat matrix, `influence()$hat`, see also `hat`, and where the Residual-Leverage uses standardized Pearson residuals (`residuals.glm(type = "pearson")`) for $R[i]$.

The Residual-Leverage plot shows contours of equal Cook's distance, for values of `cook.levels` (by default 0.5 and 1) and omits cases with leverage one with a warning. If the leverages are constant (as is typically the case in a balanced `aov` situation) the plot uses factor level combinations instead of the leverages for the x-axis. (The factor levels are ordered by mean fitted value.)

In the Cook's distance vs leverage/(1-leverage) plot, contours of standardized residuals that are equal in magnitude are lines through the origin. The contour lines are labelled with the magnitudes.

Author(s)

John Maindonald and Martin Maechler.

References

Belsley, D. A., Kuh, E. and Welsch, R. E. (1980) *Regression Diagnostics.* New York: Wiley.

Cook, R. D. and Weisberg, S. (1982) *Residuals and Influence in Regression.* London: Chapman and Hall.

Firth, D. (1991) Generalized Linear Models. In Hinkley, D. V. and Reid, N. and Snell, E. J., eds: Pp. 55-82 in Statistical Theory and Modelling. In Honour of Sir David Cox, FRS. London: Chapman and Hall.

Hinkley, D. V. (1975) On power transformations to symmetry. *Biometrika* **62**, 101–111.

McCullagh, P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

See Also

`termplot`, `lm.influence`, `cooks.distance`, `hatvalues`.

Examples

```
require(graphics)

## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
plot(lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings))

## 4 plots on 1 page;
## allow room for printing model formula in outer margin:
par(mfrow = c(2, 2), oma = c(0, 0, 2, 0))
plot(lm.SR)
plot(lm.SR, id.n = NULL)               # no id's
plot(lm.SR, id.n = 5, labels.id = NULL)# 5 id numbers

## Was default in R <= 2.1.x:
## Cook's distances instead of Residual-Leverage plot
plot(lm.SR, which = 1:4)

## Fit a smooth curve, where applicable:
plot(lm.SR, panel = panel.smooth)
## Gives a smoother curve
plot(lm.SR, panel = function(x,y) panel.smooth(x, y, span = 1))
```

```
par(mfrow=c(2,1))# same oma as above
plot(lm.SR, which = 1:2, sub.caption = "Saving Rates, n=50, p=5")
```

`plot.ppr` *Plot Ridge Functions for Projection Pursuit Regression Fit*

Description

Plot ridge functions for projection pursuit regression fit.

Usage

```
## S3 method for class 'ppr':
plot(x, ask, type = "o", ...)
```

Arguments

`x`	A fit of class `"ppr"` as produced by a call to `ppr`.
`ask`	the graphics parameter `ask`: see `par` for details. If set to `TRUE` will ask between the plot of each cross-section.
`type`	the type of line to draw
`...`	further graphical parameters

Value

None

Side Effects

A series of plots are drawn on the current graphical device, one for each term in the fit.

See Also

`ppr`, `par`

Examples

```
require(graphics)

with(rock, {
area1 <- area/10000; peri1 <- peri/10000
par(mfrow=c(3,2))# maybe: , pty="s")
rock.ppr <- ppr(log(perm) ~ area1 + peri1 + shape,
                data = rock, nterms = 2, max.terms = 5)
plot(rock.ppr, main="ppr(log(perm)~ ., nterms=2, max.terms=5)")
plot(update(rock.ppr, bass=5), main = "update(..., bass = 5)")
plot(update(rock.ppr, sm.method="gcv", gcvpen=2),
     main = "update(..., sm.method=\"gcv\", gcvpen=2)")
})
```

`plot.profile.nls` *Plot a profile.nls Object*

Description

Displays a series of plots of the profile t function and interpolated confidence intervals for the parameters in a nonlinear regression model that has been fit with `nls` and profiled with `profile.nls`.

Usage

```
## S3 method for class 'profile.nls':
plot(x, levels, conf= c(99, 95, 90, 80, 50)/100,
     absVal =TRUE, ...)
```

Arguments

`x`	an object of class `"profile.nls"`
`levels`	levels, on the scale of the absolute value of a t statistic, at which to interpolate intervals. Usually `conf` is used instead of giving `levels` explicitly.
`conf`	a numeric vector of confidence levels for profile-based confidence intervals on the parameters. Defaults to `c(0.99, 0.95, 0.90, 0.80, 0.50)`.
`absVal`	a logical value indicating whether or not the plots should be on the scale of the absolute value of the profile t. Defaults to `TRUE`.
`...`	other arguments to the `plot` function can be passed here.

Author(s)

Douglas M. Bates and Saikat DebRoy

References

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley (chapter 6)

See Also

`nls`, `profile`, `profile.nls`

Examples

```
require(graphics)

# obtain the fitted object
fm1 <- nls(demand ~ SSasympOrig(Time, A, lrc), data = BOD)
# get the profile for the fitted model
pr1 <- profile(fm1, alpha = 0.05)
opar <- par(mfrow = c(2,2), oma = c(1.1, 0, 1.1, 0), las = 1)
plot(pr1, conf = c(95, 90, 80, 50)/100)
plot(pr1, conf = c(95, 90, 80, 50)/100, absVal = FALSE)
mtext("Confidence intervals based on the profile sum of squares",
      side = 3, outer = TRUE)
```

```
mtext("BOD data - confidence levels of 50%, 80%, 90% and 95%",
      side = 1, outer = TRUE)
par(opar)
```

`plot.spec` *Plotting Spectral Densities*

Description

Plotting method for objects of class `"spec"`. For multivariate time series it plots the marginal spectra of the series or pairs plots of the coherency and phase of the cross-spectra.

Usage

```
## S3 method for class 'spec':
plot(x, add = FALSE, ci = 0.95, log = c("yes", "dB", "no"),
     xlab = "frequency", ylab = NULL, type = "l",
     ci.col = "blue", ci.lty = 3,
     main = NULL, sub = NULL,
     plot.type = c("marginal", "coherency", "phase"),
     ...)

plot.spec.phase(x, ci = 0.95,
                xlab = "frequency", ylab = "phase",
                ylim = c(-pi, pi), type = "l",
                main = NULL, ci.col = "blue", ci.lty = 3, ...)

plot.spec.coherency(x, ci = 0.95,
                    xlab = "frequency",
                    ylab = "squared coherency",
                    ylim = c(0, 1), type = "l",
                    main = NULL, ci.col = "blue", ci.lty = 3, ...)
```

Arguments

`x`	an object of class `"spec"`.
`add`	logical. If `TRUE`, add to already existing plot. Only valid for `plot.type = "marginal"`.
`ci`	coverage probability for confidence interval. Plotting of the confidence bar/limits is omitted unless `ci` is strictly positive.
`log`	If `"dB"`, plot on log10 (decibel) scale (as S-PLUS), otherwise use conventional log scale or linear scale. Logical values are also accepted. The default is `"yes"` unless `options(ts.S.compat = TRUE)` has been set, when it is `"dB"`. Only valid for `plot.type = "marginal"`.
`xlab`	the x label of the plot.
`ylab`	the y label of the plot. If missing a suitable label will be constructed.
`type`	the type of plot to be drawn, defaults to lines.
`ci.col`	colour for plotting confidence bar or confidence intervals for coherency and phase.

ci.lty	line type for confidence intervals for coherency and phase.
main	overall title for the plot. If missing, a suitable title is constructed.
sub	a sub title for the plot. Only used for plot.type = "marginal". If missing, a description of the smoothing is used.
plot.type	For multivariate time series, the type of plot required. Only the first character is needed.
ylim, ...	Graphical parameters.

See Also

spectrum

---

plot.stepfun *Plot Step Functions*

Description

Method of the generic plot for stepfun objects and utility for plotting piecewise constant functions.

Usage

```
## S3 method for class 'stepfun':
plot(x, xval, xlim, ylim,
     xlab = "x", ylab = "f(x)", main = NULL,
     add = FALSE, verticals = TRUE, do.points = TRUE,
     pch = par("pch"),
     col.points = par("col"), cex.points = par("cex"),
     col.hor = par("col"), col.vert = par("col"),
     lty = par("lty"), lwd = par("lwd"), ...)

## S3 method for class 'stepfun':
lines(x, ...)
```

Arguments

x	an R object inheriting from "stepfun".
xval	numeric vector of abscissa values at which to evaluate x. Defaults to knots(x) restricted to xlim.
xlim,ylim	numeric(2) each; range of x or y values to use. Both have sensible defaults.
xlab,ylab	labels of x and y axis.
main	main title.
add	logical; if TRUE only *add* to an existing plot.
verticals	logical; if TRUE, draw vertical lines at steps.
do.points	logical; if true, also draw points at the (xlim restricted) knot locations.
pch	character; point character if do.points.
col.points	character or integer code; color of points if do.points.

cex.points	numeric; character expansion factor if do.points.
col.hor	color of horizontal lines.
col.vert	color of vertical lines.
lty, lwd	line type and thickness for all lines.
...	further arguments of plot(.), or if (add) segments(.).

Value

A list with two components

t	abscissa (x) values, including the two outermost ones.
y	y values 'in between' the t[].

Author(s)

Martin Maechler ⟨maechler@stat.math.ethz.ch⟩, 1990, 1993; ported to R, 1997.

See Also

ecdf for empirical distribution functions as special step functions, approxfun and splinefun.

Examples

```
require(graphics)

y0 <- c(1,2,4,3)
sfun0  <- stepfun(1:3, y0, f = 0)
sfun.2 <- stepfun(1:3, y0, f = .2)
sfun1  <- stepfun(1:3, y0, right = TRUE)

tt <- seq(0,3, by=0.1)
op <- par(mfrow=c(2,2))
plot(sfun0); plot(sfun0, xval=tt, add=TRUE, col.hor="bisque")
plot(sfun.2);plot(sfun.2,xval=tt, add=TRUE, col.hor="orange")
plot(sfun1);lines(sfun1, xval=tt, col.hor="coral")
##-- This is  revealing :
plot(sfun0, verticals= FALSE,
     main = "stepfun(x, y0, f=f)  for f = 0, .2, 1")
for(i in 1:3)
  lines(list(sfun0,sfun.2,stepfun(1:3,y0,f = 1))[[i]],
        col.hor=i, col.vert=i)
legend(2.5, 1.9, paste("f =", c(0,0.2,1)), col=1:3, lty=1, y.intersp=1)
par(op)

# Extend and/or restrict 'viewport':
plot(sfun0, xlim = c(0,5), ylim = c(0, 3.5),
     main = "plot(stepfun(*), xlim= . , ylim = .)")

##-- this works too (automatic call to  ecdf(.)):
plot.stepfun(rt(50, df=3), col.vert = "gray20")
```

`plot.ts` *Plotting Time-Series Objects*

Description

Plotting method for objects inheriting from class `"ts"`.

Usage

```
## S3 method for class 'ts':
plot(x, y = NULL, plot.type = c("multiple", "single"),
        xy.labels, xy.lines, panel = lines, nc, yax.flip = FALSE,
        mar.multi = c(0, 5.1, 0, if(yax.flip) 5.1 else 2.1),
        oma.multi = c(6, 0, 5, 0), axes = TRUE, ...)

## S3 method for class 'ts':
lines(x, ...)
```

Arguments

`x, y`	time series objects, usually inheriting from class `"ts"`.
`plot.type`	for multivariate time series, should the series by plotted separately (with a common time axis) or on a single plot?
`xy.labels`	logical, indicating if `text()` labels should be used for an x-y plot, *or* character, supplying a vector of labels to be used. The default is to label for up to 150 points, and not for more.
`xy.lines`	logical, indicating if `lines` should be drawn for an x-y plot. Defaults to the value of `xy.labels` if that is logical, otherwise to `TRUE`.
`panel`	a `function(x, col, bg, pch, type, ...)` which gives the action to be carried out in each panel of the display for `plot.type="multiple"`. The default is `lines`.
`nc`	the number of columns to use when `type="multiple"`. Defaults to 1 for up to 4 series, otherwise to 2.
`yax.flip`	logical indicating if the y-axis (ticks and numbering) should flip from side 2 (left) to 4 (right) from series to series when `type="multiple"`.
`mar.multi, oma.multi`	the (default) `par` settings for `plot.type="multiple"`. Modify with care!
`axes`	logical indicating if x- and y- axes should be drawn.
`...`	additional graphical arguments, see `plot`, `plot.default` and `par`.

Details

If `y` is missing, this function creates a time series plot, for multivariate series of one of two kinds depending on `plot.type`.

If `y` is present, both `x` and `y` must be univariate, and a scatter plot `y ~ x` will be drawn, enhanced by using `text` if `xy.labels` is `TRUE` or `character`, and `lines` if `xy.lines` is `TRUE`.

See Also

`ts` for basic time series construction and access functionality.

Examples

```
require(graphics)

## Multivariate
z <- ts(matrix(rt(200 * 8, df = 3), 200, 8),
        start = c(1961, 1), frequency = 12)
plot(z, yax.flip = TRUE)
plot(z, axes = FALSE, ann = FALSE, frame.plot = TRUE,
     mar.multi = c(0,0,0,0), oma.multi = c(1,1,5,1))
title("plot(ts(..), axes=FALSE, ann=FALSE, frame.plot=TRUE, mar..., oma...)")

z <- window(z[,1:3], end = c(1969,12))
plot(z, type = "b")    # multiple
plot(z, plot.type="single", lty=1:3, col=4:2)

## A phase plot:
plot(nhtemp, c(nhtemp[-1], NA), cex = .8, col="blue",
     main = "Lag plot of New Haven temperatures")
## a clearer way to do this would be
## Not run:
plot(nhtemp, lag(nhtemp, 1), cex = .8, col="blue",
     main = "Lag plot of New Haven temperatures")
## End(Not run)

## xy.lines and xy.labels are FALSE for large series:
plot(lag(sunspots, 1), sunspots, pch = ".")

SMI <- EuStockMarkets[, "SMI"]
plot(lag(SMI,  1), SMI, pch = ".")
plot(lag(SMI, 20), SMI, pch = ".", log = "xy",
     main = "4 weeks lagged SMI stocks -- log scale", xy.lines= TRUE)
```

`Poisson` *The Poisson Distribution*

Description

Density, distribution function, quantile function and random generation for the Poisson distribution with parameter `lambda`.

Usage

```
dpois(x, lambda, log = FALSE)
ppois(q, lambda, lower.tail = TRUE, log.p = FALSE)
qpois(p, lambda, lower.tail = TRUE, log.p = FALSE)
rpois(n, lambda)
```

Arguments

`x`	vector of (non-negative integer) quantiles.
`q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of random values to return.
`lambda`	vector of (non-negative) means.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The Poisson distribution has density

$$p(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

for $x = 0, 1, 2, \ldots$. The mean and variance are $E(X) = Var(X) = \lambda$.

If an element of `x` is not integer, the result of `dpois` is zero, with a warning. $p(x)$ is computed using Loader's algorithm, see the reference in `dbinom`.

The quantile is left continuous: `qgeom(q, prob)` is the largest integer x such that $P(X \leq x) < q$.

Setting `lower.tail = FALSE` allows to get much more precise results when the default, `lower.tail = TRUE` would return 1, see the example below.

Value

`dpois` gives the (log) density, `ppois` gives the (log) distribution function, `qpois` gives the quantile function, and `rpois` generates random deviates.

Invalid `lambda` will result in return value `NaN`, with a warning.

Source

`dpois` uses C code contributed by Catherine Loader (see `dbinom`).

`ppois` uses `pgamma`.

`qpois` uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

`rpois` uses

Ahrens, J. H. and Dieter, U. (1982). Computer generation of Poisson deviates from modified normal distributions. *ACM Transactions on Mathematical Software*, **8**, 163–179.

See Also

`dbinom` for the binomial and `dnbinom` for the negative binomial distribution.

Examples

```
require(graphics)

-log(dpois(0:7, lambda=1) * gamma(1+ 0:7)) # == 1
Ni <- rpois(50, lambda = 4); table(factor(Ni, 0:max(Ni)))

1 - ppois(10*(15:25), lambda=100)  # becomes 0 (cancellation)
    ppois(10*(15:25), lambda=100, lower.tail=FALSE)     # no cancellation

par(mfrow = c(2, 1))
x <- seq(-0.01, 5, 0.01)
plot(x, ppois(x, 1), type="s", ylab="F(x)", main="Poisson(1) CDF")
plot(x, pbinom(x, 100, 0.01),type="s", ylab="F(x)",
     main="Binomial(100, 0.01) CDF")
```

---

poly *Compute Orthogonal Polynomials*

Description

Returns or evaluates orthogonal polynomials of degree 1 to `degree` over the specified set of points `x`. These are all orthogonal to the constant polynomial of degree 0. Alternatively, evaluate raw polynomials.

Usage

```
poly(x, ..., degree = 1, coefs = NULL, raw = FALSE)
polym(..., degree = 1, raw = FALSE)

## S3 method for class 'poly':
predict(object, newdata, ...)
```

Arguments

`x, newdata`	a numeric vector at which to evaluate the polynomial. `x` can also be a matrix. Missing values are not allowed in `x`.
`degree`	the degree of the polynomial. Must be less than the number of unique points.
`coefs`	for prediction, coefficients from a previous fit.
`raw`	if true, use raw and not orthogonal polynomials.
`object`	an object inheriting from class `"poly"`, normally the result of a call to `poly` with a single vector argument.
`...`	`poly, polym`: further vectors. `predict.poly`: arguments to be passed to or from other methods.

Details

Although formally `degree` should be named (as it follows `...`), an unnamed second argument of length 1 will be interpreted as the degree.

The orthogonal polynomial is summarized by the coefficients, which can be used to evaluate it via the three-term recursion given in Kennedy & Gentle (1980, pp. 343–4), and used in the `predict` part of the code.

Value

For `poly` with a single vector argument:
A matrix with rows corresponding to points in `x` and columns corresponding to the degree, with attributes `"degree"` specifying the degrees of the columns and (unless `raw = TRUE`) `"coefs"` which contains the centering and normalization constants used in constructing the orthogonal polynomials. The matrix has given class `c("poly", "matrix")`.

Other cases of `poly` and `polym`, and `predict.poly`: a matrix.

Note

This routine is intended for statistical purposes such as `contr.poly`: it does not attempt to orthogonalize to machine accuracy.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

Kennedy, W. J. Jr and Gentle, J. E. (1980) *Statistical Computing* Marcel Dekker.

See Also

`contr.poly`.

`cars` for an example of polynomial regression.

Examples

```
(z <- poly(1:10, 3))
predict(z, seq(2, 4, 0.5))
poly(seq(4, 6, 0.5), 3, coefs = attr(z, "coefs"))

polym(1:4, c(1, 4:6), degree=3) # or just poly()
poly(cbind(1:4, c(1, 4:6)), degree=3)
```

`power` *Create a Power Link Object*

Description

Creates a link object based on the link function $\eta = \mu^{\lambda}$.

Usage

```
power(lambda = 1)
```

Arguments

`lambda`	a real number.

Details

If `lambda` is non-positive, it is taken as zero, and the log link is obtained. The default `lambda = 1` gives the identity link.

Value

A list with components `linkfun`, `linkinv`, `mu.eta`, and `valideta`. See `make.link` for information on their meaning.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S.* Wadsworth & Brooks/Cole.

See Also

`make.link`, `family`

To raise a number to a power, see `Arithmetic`.

To calculate the power of a test, see various functions in the **stats** package, e.g., `power.t.test`.

Examples

```
power()
quasi(link=power(1/3))[c("linkfun", "linkinv")]
```

---

power.anova.test *Power calculations for balanced one-way analysis of variance tests*

Description

Compute power of test or determine parameters to obtain target power.

Usage

```
power.anova.test(groups = NULL, n = NULL,
                 between.var = NULL, within.var = NULL,
                 sig.level = 0.05, power = NULL)
```

Arguments

`groups`	Number of groups
`n`	Number of observations (per group)
`between.var`	Between group variance
`within.var`	Within group variance
`sig.level`	Significance level (Type I error probability)
`power`	Power of test (1 minus Type II error probability)

Details

Exactly one of the parameters `groups`, `n`, `between.var`, `power`, `within.var`, and `sig.level` must be passed as NULL, and that parameter is determined from the others. Notice that `sig.level` has non-NULL default so NULL must be explicitly passed if you want it computed.

Value

Object of class "power.htest", a list of the arguments (including the computed one) augmented with method and note elements.

Note

uniroot is used to solve power equation for unknowns, so you may see errors from it, notably about inability to bracket the root when invalid arguments are given.

Author(s)

Claus Ekstrøm

See Also

anova, lm, uniroot

Examples

```
power.anova.test(groups=4, n=5, between.var=1, within.var=3)
# Power = 0.3535594

power.anova.test(groups=4, between.var=1, within.var=3,
                 power=.80)
# n = 11.92613

## Assume we have prior knowledge of the group means:
groupmeans <- c(120, 130, 140, 150)
power.anova.test(groups = length(groupmeans),
                 between.var=var(groupmeans),
                 within.var=500, power=.90) # n = 15.18834
```

---

power.prop.test *Power calculations two sample test for proportions*

Description

Compute power of test, or determine parameters to obtain target power.

Usage

```
power.prop.test(n = NULL, p1 = NULL, p2 = NULL, sig.level = 0.05,
                power = NULL,
                alternative = c("two.sided", "one.sided"),
                strict = FALSE)
```

Arguments

n	Number of observations (per group)
p1	probability in one group
p2	probability in other group
sig.level	Significance level (Type I error probability)
power	Power of test (1 minus Type II error probability)
alternative	One- or two-sided test
strict	Use strict interpretation in two-sided case

Details

Exactly one of the parameters n, p1, p2, power, and sig.level must be passed as NULL, and that parameter is determined from the others. Notice that sig.level has a non-NULL default so NULL must be explicitly passed if you want it computed.

If strict = TRUE is used, the power will include the probability of rejection in the opposite direction of the true effect, in the two-sided case. Without this the power will be half the significance level if the true difference is zero.

Value

Object of class "power.htest", a list of the arguments (including the computed one) augmented with method and note elements.

Note

uniroot is used to solve power equation for unknowns, so you may see errors from it, notably about inability to bracket the root when invalid arguments are given. If one of them is computed p1 < p2 will hold, although this is not enforced when both are specified.

Author(s)

Peter Dalgaard. Based on previous work by Claus Ekstrøm

See Also

prop.test, uniroot

Examples

```
power.prop.test(n = 50, p1 = .50, p2 = .75)
power.prop.test(p1 = .50, p2 = .75, power = .90)
power.prop.test(n = 50, p1 = .5, power = .90)
```

`power.t.test` *Power calculations for one and two sample t tests*

Description

Compute power of test, or determine parameters to obtain target power.

Usage

```
power.t.test(n = NULL, delta = NULL, sd = 1, sig.level = 0.05,
             power = NULL,
             type = c("two.sample", "one.sample", "paired"),
             alternative = c("two.sided", "one.sided"),
             strict = FALSE)
```

Arguments

`n`	Number of observations (per group)
`delta`	True difference in means
`sd`	Standard deviation
`sig.level`	Significance level (Type I error probability)
`power`	Power of test (1 minus Type II error probability)
`type`	Type of t test
`alternative`	One- or two-sided test
`strict`	Use strict interpretation in two-sided case

Details

Exactly one of the parameters `n`, `delta`, `power`, `sd`, and `sig.level` must be passed as NULL, and that parameter is determined from the others. Notice that the last two have non-NULL defaults so NULL must be explicitly passed if you want to compute them.

If `strict = TRUE` is used, the power will include the probability of rejection in the opposite direction of the true effect, in the two-sided case. Without this the power will be half the significance level if the true difference is zero.

Value

Object of class `"power.htest"`, a list of the arguments (including the computed one) augmented with `method` and `note` elements.

Note

`uniroot` is used to solve power equation for unknowns, so you may see errors from it, notably about inability to bracket the root when invalid arguments are given.

Author(s)

Peter Dalgaard. Based on previous work by Claus Ekstrøm

See Also

t.test, uniroot

Examples

```
power.t.test(n = 20, delta = 1)
power.t.test(power = .90, delta = 1)
power.t.test(power = .90, delta = 1, alternative = "one.sided")
```

PP.test *Phillips-Perron Test for Unit Roots*

Description

Computes the Phillips-Perron test for the null hypothesis that x has a unit root against a stationary alternative.

Usage

```
PP.test(x, lshort = TRUE)
```

Arguments

x	a numeric vector or univariate time series.
lshort	a logical indicating whether the short or long version of the truncation lag parameter is used.

Details

The general regression equation which incorporates a constant and a linear trend is used and the corrected t-statistic for a first order autoregressive coefficient equals one is computed. To estimate sigma^2 the Newey-West estimator is used. If lshort is TRUE, then the truncation lag parameter is set to trunc(4*(n/100)^0.25), otherwise trunc(12*(n/100)^0.25) is used. The p-values are interpolated from Table 4.2, page 103 of Banerjee *et al.* (1993).

Missing values are not handled.

Value

A list with class "htest" containing the following components:

statistic	the value of the test statistic.
parameter	the truncation lag parameter.
p.value	the p-value of the test.
method	a character string indicating what type of test was performed.
data.name	a character string giving the name of the data.

Author(s)

A. Trapletti

References

A. Banerjee, J. J. Dolado, J. W. Galbraith, and D. F. Hendry (1993) *Cointegration, Error Correction, and the Econometric Analysis of Non-Stationary Data*, Oxford University Press, Oxford.

P. Perron (1988) Trends and random walks in macroeconomic time series. *Journal of Economic Dynamics and Control* **12**, 297–332.

Examples

```
x <- rnorm(1000)
PP.test(x)
y <- cumsum(x) # has unit root
PP.test(y)
```

---

ppoints *Ordinates for Probability Plotting*

Description

Generates the sequence of probability points `(1:m - a)/(m + (1-a)-a)` where `m` is either `n`, if `length(n)==1`, or `length(n)`.

Usage

```
ppoints(n, a = ifelse(n <= 10, 3/8, 1/2))
```

Arguments

`n`	either the number of points generated or a vector of observations.
`a`	the offset fraction to be used; typically in $(0, 1)$.

Details

If $0 < a < 1$, the resulting values are within $(0, 1)$ (excluding boundaries). In any case, the resulting sequence is symmetric in $[0, 1]$, i.e., `p + rev(p) == 1`.

`ppoints()` is used in `qqplot` and `qqnorm` to generate the set of probabilities at which to evaluate the inverse distribution.

The choice of `a` follows the documentation of the function of the same name in Becker *et al* (1988), and appears to have been motivated by results from Blom (1958) on approximations to expect normal order statistics (see also `quantile`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Blom, G. (1958) *Statistical Estimates and Transformed Beta Variables.* Wiley

See Also

`qqplot`, `qqnorm`.

Examples

```
ppoints(4) # the same as  ppoints(1:4)
ppoints(10)
ppoints(10, a=1/2)
```

---

ppr *Projection Pursuit Regression*

Description

Fit a projection pursuit regression model.

Usage

```
ppr(x, ...)

## S3 method for class 'formula':
ppr(formula, data, weights, subset, na.action,
    contrasts = NULL, ..., model = FALSE)

## Default S3 method:
ppr(x, y, weights = rep(1,n),
    ww = rep(1,q), nterms, max.terms = nterms, optlevel = 2,
    sm.method = c("supsmu", "spline", "gcvspline"),
    bass = 0, span = 0, df = 5, gcvpen = 1, ...)
```

Arguments

formula	a formula specifying one or more numeric response variables and the explanatory variables.
x	numeric matrix of explanatory variables. Rows represent observations, and columns represent variables. Missing values are not accepted.
y	numeric matrix of response variables. Rows represent observations, and columns represent variables. Missing values are not accepted.
nterms	number of terms to include in the final model.
data	a data frame (or similar: see model.frame) from which variables specified in formula are preferentially to be taken.
weights	a vector of weights w_i for each *case*.
ww	a vector of weights for each *response*, so the fit criterion is the sum over case i and responses j of w_i ww_j (y_ij - fit_ij)^2 divided by the sum of w_i.
subset	an index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
na.action	a function to specify the action to be taken if NAs are found. The default action is given by getOption("na.action"). (NOTE: If given, this argument must be named.)
contrasts	the contrasts to be used when any factor explanatory variables are coded.
max.terms	maximum number of terms to choose from when building the model.

optlevel	integer from 0 to 3 which determines the thoroughness of an optimization routine in the SMART program. See the 'Details' section.
sm.method	the method used for smoothing the ridge functions. The default is to use Friedman's super smoother supsmu. The alternatives are to use the smoothing spline code underlying smooth.spline, either with a specified (equivalent) degrees of freedom for each ridge functions, or to allow the smoothness to be chosen by GCV.
bass	super smoother bass tone control used with automatic span selection (see supsmu); the range of values is 0 to 10, with larger values resulting in increased smoothing.
span	super smoother span control (see supsmu). The default, 0, results in automatic span selection by local cross validation. span can also take a value in (0, 1].
df	if sm.method is "spline" specifies the smoothness of each ridge term via the requested equivalent degrees of freedom.
gcvpen	if sm.method is "gcvspline" this is the penalty used in the GCV selection for each degree of freedom used.
...	arguments to be passed to or from other methods.
model	logical. If true, the model frame is returned.

Details

The basic method is given by Friedman (1984), and is essentially the same code used by S-PLUS's ppreg. This code is extremely sensitive to the compiler used.

The algorithm first adds up to max.terms ridge terms one at a time; it will use less if it is unable to find a term to add that makes sufficient difference. It then removes the least important term at each step until nterms terms are left.

The levels of optimization (argument optlevel) differ in how thoroughly the models are refitted during this process. At level 0 the existing ridge terms are not refitted. At level 1 the projection directions are not refitted, but the ridge functions and the regression coefficients are. Levels 2 and 3 refit all the terms and are equivalent for one response; level 3 is more careful to re-balance the contributions from each regressor at each step and so is a little less likely to converge to a saddle point of the sum of squares criterion.

Value

A list with the following components, many of which are for use by the method functions.

call	the matched call
p	the number of explanatory variables (after any coding)
q	the number of response variables
mu	the argument nterms
ml	the argument max.terms
gof	the overall residual (weighted) sum of squares for the selected model
gofn	the overall residual (weighted) sum of squares against the number of terms, up to max.terms. Will be invalid (and zero) for less than nterms.
df	the argument df
edf	if sm.method is "spline" or "gcvspline" the equivalent number of degrees of freedom for each ridge term used.

`xnames`	the names of the explanatory variables
`ynames`	the names of the response variables
`alpha`	a matrix of the projection directions, with a column for each ridge term
`beta`	a matrix of the coefficients applied for each response to the ridge terms: the rows are the responses and the columns the ridge terms
`yb`	the weighted means of each response
`ys`	the overall scale factor used: internally the responses are divided by `ys` to have unit total weighted sum of squares.
`fitted.values`	the fitted values, as a matrix if `q > 1`.
`residuals`	the residuals, as a matrix if `q > 1`.
`smod`	internal work array, which includes the ridge functions evaluated at the training set points.
`model`	(only if `model=TRUE`) the model frame.

References

Friedman, J. H. and Stuetzle, W. (1981) Projection pursuit regression. *Journal of the American Statistical Association*, **76**, 817–823.

Friedman, J. H. (1984) SMART User's Guide. Laboratory for Computational Statistics, Stanford University Technical Report No. 1.

Venables, W. N. & Ripley, B. D. (2002) *Modern Applied Statistics with S.* Springer.

See Also

`plot.ppr`, `supsmu`, `smooth.spline`

Examples

```
require(graphics)

# Note: your numerical values may differ
attach(rock)
area1 <- area/10000; peri1 <- peri/10000
rock.ppr <- ppr(log(perm) ~ area1 + peri1 + shape,
                data = rock, nterms = 2, max.terms = 5)
rock.ppr
# Call:
# ppr.formula(formula = log(perm) ~ area1 + peri1 + shape, data = rock,
#     nterms = 2, max.terms = 5)
#
# Goodness of fit:
#  2 terms  3 terms  4 terms  5 terms
# 8.737806 5.289517 4.745799 4.490378

summary(rock.ppr)
# .....  (same as above)
# .....
#
# Projection direction vectors:
#       term 1      term 2
# area1  0.34357179  0.37071027
```

```
# peri1 -0.93781471 -0.61923542
# shape  0.04961846  0.69218595
#
# Coefficients of ridge terms:
#    term 1    term 2
# 1.6079271 0.5460971

par(mfrow=c(3,2))# maybe: , pty="s")
plot(rock.ppr, main="ppr(log(perm)~ ., nterms=2, max.terms=5)")
plot(update(rock.ppr, bass=5), main = "update(..., bass = 5)")
plot(update(rock.ppr, sm.method="gcv", gcvpen=2),
     main = "update(..., sm.method=\"gcv\", gcvpen=2)")
cbind(perm=rock$perm, prediction=round(exp(predict(rock.ppr)), 1))
detach()
```

`prcomp` *Principal Components Analysis*

Description

Performs a principal components analysis on the given data matrix and returns the results as an object of class `prcomp`.

Usage

```
prcomp(x, ...)

## S3 method for class 'formula':
prcomp(formula, data = NULL, subset, na.action, ...)

## Default S3 method:
prcomp(x, retx = TRUE, center = TRUE, scale. = FALSE,
       tol = NULL, ...)

## S3 method for class 'prcomp':
predict(object, newdata, ...)
```

Arguments

`formula`	a formula with no response variable, referring only to numeric variables.
`data`	an optional data frame (or similar: see `model.frame`) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.
`subset`	an optional vector used to select rows (observations) of the data matrix `x`.
`na.action`	a function which indicates what should happen when the data contain `NA`s. The default is set by the `na.action` setting of `options`, and is `na.fail` if that is unset. The 'factory-fresh' default is `na.omit`.
`...`	arguments passed to or from other methods. If `x` is a formula one might specify `scale.` or `tol`.
`x`	a numeric or complex matrix (or data frame) which provides the data for the principal components analysis.

retx	a logical value indicating whether the rotated variables should be returned.
center	a logical value indicating whether the variables should be shifted to be zero centered. Alternately, a vector of length equal the number of columns of x can be supplied. The value is passed to scale.
scale.	a logical value indicating whether the variables should be scaled to have unit variance before the analysis takes place. The default is FALSE for consistency with S, but in general scaling is advisable. Alternatively, a vector of length equal the number of columns of x can be supplied. The value is passed to scale.
tol	a value indicating the magnitude below which components should be omitted. (Components are omitted if their standard deviations are less than or equal to tol times the standard deviation of the first component.) With the default null setting, no components are omitted. Other settings for tol could be tol = 0 or tol = sqrt(.Machine$double.eps), which would omit essentially constant components.
object	Object of class inheriting from "prcomp"
newdata	An optional data frame or matrix in which to look for variables with which to predict. If omitted, the scores are used. If the original fit used a formula or a data frame or a matrix with column names, newdata must contain columns with the same names. Otherwise it must contain the same number of columns, to be used in the same order.

Details

The calculation is done by a singular value decomposition of the (centered and possibly scaled) data matrix, not by using eigen on the covariance matrix. This is generally the preferred method for numerical accuracy. The print method for these objects prints the results in a nice format and the plot method produces a scree plot.

Note that scale = TRUE cannot be used if there are zero or constant (for center = TRUE) variables.

Value

prcomp returns a list with class "prcomp" containing the following components:

sdev	the standard deviations of the principal components (i.e., the square roots of the eigenvalues of the covariance/correlation matrix, though the calculation is actually done with the singular values of the data matrix).
rotation	the matrix of variable loadings (i.e., a matrix whose columns contain the eigenvectors). The function princomp returns this in the element loadings.
x	if retx is true the value of the rotated data (the centred (and scaled if requested) data multiplied by the rotation matrix) is returned. Hence, cov(x) is the diagonal matrix diag(sdev^2). For the formula method, napredict is applied to handle the treatment of values omitted by the na.action.
center, scale	the centering and scaling used, or FALSE.

Note

The signs of the columns of the rotation matrix are arbitrary, and so may differ between different programs for PCA, and even between different builds of R.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Mardia, K. V., J. T. Kent, and J. M. Bibby (1979) *Multivariate Analysis*, London: Academic Press.

Venables, W. N. and B. D. Ripley (2002) *Modern Applied Statistics with S*, Springer-Verlag.

See Also

`biplot.prcomp`, `screeplot`, `princomp`, `cor`, `cov`, `svd`, `eigen`.

Examples

```
require(graphics)

## the variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
prcomp(USArrests)  # inappropriate
prcomp(USArrests, scale = TRUE)
prcomp(~ Murder + Assault + Rape, data = USArrests, scale = TRUE)
plot(prcomp(USArrests))
summary(prcomp(USArrests, scale = TRUE))
biplot(prcomp(USArrests, scale = TRUE))
```

`predict` *Model Predictions*

Description

`predict` is a generic function for predictions from the results of various model fitting functions. The function invokes particular *methods* which depend on the `class` of the first argument.

Usage

```
predict (object, ...)
```

Arguments

`object`	a model object for which prediction is desired.
`...`	additional arguments affecting the predictions produced.

Details

Most prediction methods which similar to fitting linear models have an argument `newdata` specifying the first place to look for explanatory variables to be used for prediction. Some considerable attempts are made to match up the columns in `newdata` to those used for fitting, for example that they are of comparable types and that any factors have the same level set in the same order (or can be transformed to be so).

Time series prediction methods in package **stats** have an argument `n.ahead` specifying how many time steps ahead to predict.

Many methods have a logical argument `se.fit` saying if standard errors are to returned.

Value

The form of the value returned by `predict` depends on the class of its argument. See the documentation of the particular methods for details of what is produced by that method.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`predict.glm`, `predict.lm`, `predict.loess`, `predict.nls`, `predict.poly`, `predict.princomp`, `predict.smooth.spline`.

For time-series prediction, `predict.ar`, `predict.Arima`, `predict.arima0`, `predict.HoltWinters`, `predict.StructTS`.

Examples

```
require(utils)

## All the "predict" methods found
## NB most of the methods in the standard packages are hidden.
for(fn in methods("predict"))
   try({
       f <- eval(substitute(getAnywhere(fn)$objs[[1]], list(fn = fn)))
       cat(fn, ":\n\t", deparse(args(f)), "\n")
       }, silent = TRUE)
```

`predict.Arima` *Forecast from ARIMA fits*

Description

Forecast from models fitted by `arima`.

Usage

```
## S3 method for class 'Arima':
predict(object, n.ahead = 1, newxreg = NULL,
        se.fit = TRUE, ...)
```

Arguments

`object`	The result of an `arima` fit.
`n.ahead`	The number of steps ahead for which prediction is required.
`newxreg`	New values of `xreg` to be used for prediction. Must have at least `n.ahead` rows.
`se.fit`	Logical: should standard errors of prediction be returned?
`...`	arguments passed to or from other methods.

Details

Finite-history prediction is used, via `KalmanForecast`. This is only statistically efficient if the MA part of the fit is invertible, so `predict.Arima` will give a warning for non-invertible MA models.

The standard errors of prediction exclude the uncertainty in the estimation of the ARMA model and the regression coefficients. According to Harvey (1993, pp. 58–9) the effect is small.

Value

A time series of predictions, or if `se.fit = TRUE`, a list with components `pred`, the predictions, and `se`, the estimated standard errors. Both components are time series.

References

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press.

Harvey, A. C. and McKenzie, C. R. (1982) Algorithm AS182. An algorithm for finite sample prediction from ARIMA processes. *Applied Statistics* **31**, 180–187.

Harvey, A. C. (1993) *Time Series Models*, 2nd Edition, Harvester Wheatsheaf, sections 3.3 and 4.4.

See Also

`arima`

Examples

```
predict(arima(lh, order = c(3,0,0)), n.ahead = 12)

(fit <- arima(USAccDeaths, order = c(0,1,1),
              seasonal = list(order=c(0,1,1))))
predict(fit, n.ahead = 6)
```

`predict.glm` *Predict Method for GLM Fits*

Description

Obtains predictions and optionally estimates standard errors of those predictions from a fitted generalized linear model object.

Usage

```
## S3 method for class 'glm':
predict(object, newdata = NULL,
            type = c("link", "response", "terms"),
            se.fit = FALSE, dispersion = NULL, terms = NULL,
            na.action = na.pass, ...)
```

Arguments

object	a fitted object of class inheriting from "glm".
newdata	optionally, a data frame in which to look for variables with which to predict. If omitted, the fitted linear predictors are used.
type	the type of prediction required. The default is on the scale of the linear predictors; the alternative "response" is on the scale of the response variable. Thus for a default binomial model the default predictions are of log-odds (probabilities on logit scale) and type = "response" gives the predicted probabilities. The "terms" option returns a matrix giving the fitted values of each term in the model formula on the linear predictor scale. The value of this argument can be abbreviated.
se.fit	logical switch indicating if standard errors are required.
dispersion	the dispersion of the GLM fit to be assumed in computing the standard errors. If omitted, that returned by summary applied to the object is used.
terms	with type="terms" by default all terms are returned. A character vector specifies which terms are to be returned
na.action	function determining what should be done with missing values in newdata. The default is to predict NA.
...	further arguments passed to or from other methods.

Details

If newdata is omitted the predictions are based on the data used for the fit. In that case how cases with missing values in the original fit is determined by the na.action argument of that fit. If na.action = na.omit omitted cases will not appear in the residuals, whereas if na.action = na.exclude they will appear (in predictions and standard errors), with residual value NA. See also napredict.

Value

If se = FALSE, a vector or matrix of predictions. If se = TRUE, a list with components

fit	Predictions
se.fit	Estimated standard errors
residual.scale	A scalar giving the square root of the dispersion used in computing the standard errors.

Note

Variables are first looked for in newdata and then searched for in the usual way (which will include the environment of the formula used in the fit). A warning will be given if the variables found are not of the same length as those in newdata if it was supplied.

See Also

glm, SafePrediction

Examples

```
require(graphics)

## example from Venables and Ripley (2002, pp. 190-2.)
ldose <- rep(0:5, 2)
numdead <- c(1, 4, 9, 13, 18, 20, 0, 2, 6, 10, 12, 16)
sex <- factor(rep(c("M", "F"), c(6, 6)))
SF <- cbind(numdead, numalive=20-numdead)
budworm.lg <- glm(SF ~ sex*ldose, family=binomial)
summary(budworm.lg)

plot(c(1,32), c(0,1), type = "n", xlab = "dose",
     ylab = "prob", log = "x")
text(2^ldose, numdead/20, as.character(sex))
ld <- seq(0, 5, 0.1)
lines(2^ld, predict(budworm.lg, data.frame(ldose=ld,
   sex=factor(rep("M", length(ld)), levels=levels(sex))),
   type = "response"))
lines(2^ld, predict(budworm.lg, data.frame(ldose=ld,
   sex=factor(rep("F", length(ld)), levels=levels(sex))),
   type = "response"))
```

---

predict.HoltWinters

prediction function for fitted Holt-Winters models

Description

Computes predictions and prediction intervals for models fitted by the Holt-Winters method.

Usage

```
## S3 method for class 'HoltWinters':
predict(object, n.ahead=1, prediction.interval = FALSE,
        level = 0.95, ...)
```

Arguments

`object`	An object of class `HoltWinters`.
`n.ahead`	Number of future periods to predict.
`prediction.interval`	logical. If `TRUE`, the lower and upper bounds of the corresponding prediction intervals are computed.
`level`	Confidence level for the prediction interval.
`...`	arguments passed to or from other methods.

Value

A time series of the predicted values. If prediction intervals are requested, a multiple time series is returned with columns `fit`, `lwr` and `upr` for the predicted values and the lower and upper bounds respectively.

Author(s)

David Meyer ⟨David.Meyer@wu-wien.ac.at⟩

References

C. C. Holt (1957) Forecasting seasonals and trends by exponentially weighted moving averages, ONR Research Memorandum, Carnigie Institute 52.

P. R. Winters (1960) Forecasting sales by exponentially weighted moving averages, *Management Science* **6**, 324–342.

See Also

`HoltWinters`

Examples

```
require(graphics)

m <- HoltWinters(co2)
p <- predict(m, 50, prediction.interval = TRUE)
plot(m, p)
```

`predict.lm` *Predict method for Linear Model Fits*

Description

Predicted values based on linear model object.

Usage

```
## S3 method for class 'lm':
predict(object, newdata, se.fit = FALSE, scale = NULL, df = Inf,
        interval = c("none", "confidence", "prediction"),
        level = 0.95, type = c("response", "terms"),
        terms = NULL, na.action = na.pass,
        pred.var = res.var/weights, weights = 1, ...)
```

Arguments

`object`	Object of class inheriting from `"lm"`
`newdata`	An optional data frame in which to look for variables with which to predict. If omitted, the fitted values are used.
`se.fit`	A switch indicating if standard errors are required.
`scale`	Scale parameter for std.err. calculation
`df`	Degrees of freedom for scale
`interval`	Type of interval calculation.
`level`	Tolerance/confidence level
`type`	Type of prediction (response or model term).

terms	If type="terms", which terms (default is all terms)
na.action	function determining what should be done with missing values in newdata. The default is to predict NA.
pred.var	the variance(s) for future observations to be assumed for prediction intervals. See 'Details'.
weights	variance weights for prediction. This can be a numeric vector or a one-sided model formula. In the latter case, it is interpreted as an expression evaluated in newdata
...	further arguments passed to or from other methods.

Details

predict.lm produces predicted values, obtained by evaluating the regression function in the frame newdata (which defaults to model.frame(object). If the logical se.fit is TRUE, standard errors of the predictions are calculated. If the numeric argument scale is set (with optional df), it is used as the residual standard deviation in the computation of the standard errors, otherwise this is extracted from the model fit. Setting intervals specifies computation of confidence or prediction (tolerance) intervals at the specified level, sometimes referred to as narrow vs. wide intervals.

If the fit is rank-deficient, some of the columns of the design matrix will have been dropped. Prediction from such a fit only makes sense if newdata is contained in the same subspace as the original data. That cannot be checked accurately, so a warning is issued.

If newdata is omitted the predictions are based on the data used for the fit. In that case how cases with missing values in the original fit is determined by the na.action argument of that fit. If na.action = na.omit omitted cases will not appear in the residuals, whereas if na.action = na.exclude they will appear (in predictions, standard errors or interval limits), with residual value NA. See also napredict.

The prediction intervals are for a single observation at each case in newdata (or by default, the data used for the fit) with error variance(s) pred.var. This can be a multiple of res.var, the estimated value of σ^2: the default is to assume that future observations have the same error variance as those used for fitting. If weights is supplied, the inverse of this is used as a scale factor. For a weighted fit, if the prediction is for the original data frame, weights defaults to the weights used for the model fit, with a warning since it might not be the intended result. If the fit was weighted and newdata is given, the default is to assume constant prediction variance, with a warning.

Value

predict.lm produces a vector of predictions or a matrix of predictions and bounds with column names fit, lwr, and upr if interval is set. If se.fit is TRUE, a list with the following components is returned:

fit	vector or matrix as above
se.fit	standard error of predicted means
residual.scale	residual standard deviations
df	degrees of freedom for residual

Note

Variables are first looked for in `newdata` and then searched for in the usual way (which will include the environment of the formula used in the fit). A warning will be given if the variables found are not of the same length as those in `newdata` if it was supplied.

Offsets specified by `offset` in the fit by `lm` will not be included in predictions, whereas those specified by an offset term in the formula will be.

Notice that prediction variances and prediction intervals always refer to *future* observations, possibly corresponding to the same predictors as used for the fit. The variance of the *residuals* will be smaller.

Strictly speaking, the formula used for prediction limits assumes that the degrees of freedom for the fit are the same as those for the residual variance. This may not be the case if `res.var` is not obtained from the fit.

See Also

The model fitting function `lm`, `predict`, `SafePrediction`

Examples

```
require(graphics)

## Predictions
x <- rnorm(15)
y <- x + rnorm(15)
predict(lm(y ~ x))
new <- data.frame(x = seq(-3, 3, 0.5))
predict(lm(y ~ x), new, se.fit = TRUE)
pred.w.plim <- predict(lm(y ~ x), new, interval="prediction")
pred.w.clim <- predict(lm(y ~ x), new, interval="confidence")
matplot(new$x,cbind(pred.w.clim, pred.w.plim[,-1]),
        lty=c(1,2,2,3,3), type="l", ylab="predicted y")

## Prediction intervals, special cases
##  The first three of these throw warnings
w <- 1 + x^2
fit <- lm(y ~ x)
wfit <- lm(y ~ x, weights = w)
predict(fit, interval = "prediction")
predict(wfit, interval = "prediction")
predict(wfit, new, interval = "prediction")
predict(wfit, new, interval = "prediction", weights = (new$x)^2)
predict(wfit, new, interval = "prediction", weights = ~x^2)
```

`predict.loess` *Predict Loess Curve or Surface*

Description

Predictions from a `loess` fit, optionally with standard errors.

Usage

```
## S3 method for class 'loess':
predict(object, newdata = NULL, se = FALSE, ...)
```

Arguments

object	an object fitted by loess.
newdata	an optional data frame in which to look for variables with which to predict, or a matrix or vector containing exactly the variables needs for prediction. If missing, the original data points are used.
se	should standard errors be computed?
...	arguments passed to or from other methods.

Details

The standard errors calculation is slower than prediction.

When the fit was made using surface="interpolate" (the default), predict.loess will not extrapolate – so points outside an axis-aligned hypercube enclosing the original data will have missing (NA) predictions and standard errors.

Value

If se = FALSE, a vector giving the prediction for each row of newdata (or the original data). If se = TRUE, a list containing components

fit	the predicted values.
se	an estimated standard error for each predicted value.
residual.scale	the estimated scale of the residuals used in computing the standard errors.
df	an estimate of the effective degrees of freedom used in estimating the residual scale, intended for use with t-based confidence intervals.

If newdata was the result of a call to expand.grid, the predictions (and s.e.'s if requested) will be an array of the appropriate dimensions.

Note

Variables are first looked for in newdata and then searched for in the usual way (which will include the environment of the formula used in the fit). A warning will be given if the variables found are not of the same length as those in newdata if it was supplied.

Author(s)

B. D. Ripley, based on the cloess package of Cleveland, Grosse and Shyu.

See Also

loess

Examples

```
cars.lo <- loess(dist ~ speed, cars)
predict(cars.lo, data.frame(speed=seq(5, 30, 1)), se=TRUE)
# to get extrapolation
cars.lo2 <- loess(dist ~ speed, cars,
  control=loess.control(surface="direct"))
predict(cars.lo2, data.frame(speed=seq(5, 30, 1)), se=TRUE)
```

---

predict.nls *Predicting from Nonlinear Least Squares Fits*

Description

`predict.nls` produces predicted values, obtained by evaluating the regression function in the frame `newdata`. If the logical `se.fit` is `TRUE`, standard errors of the predictions are calculated. If the numeric argument `scale` is set (with optional `df`), it is used as the residual standard deviation in the computation of the standard errors, otherwise this is extracted from the model fit. Setting `intervals` specifies computation of confidence or prediction (tolerance) intervals at the specified `level`.

At present `se.fit` and `interval` are ignored.

Usage

```
## S3 method for class 'nls':
predict(object, newdata , se.fit = FALSE, scale = NULL, df = Inf,
        interval = c("none", "confidence", "prediction"),
        level = 0.95, ...)
```

Arguments

`object`	An object that inherits from class `nls`.
`newdata`	A named list or data frame in which to look for variables with which to predict. If `newdata` is missing the fitted values at the original data points are returned.
`se.fit`	A logical value indicating if the standard errors of the predictions should be calculated. Defaults to `FALSE`. At present this argument is ignored.
`scale`	A numeric scalar. If it is set (with optional `df`), it is used as the residual standard deviation in the computation of the standard errors, otherwise this information is extracted from the model fit. At present this argument is ignored.
`df`	A positive numeric scalar giving the number of degrees of freedom for the `scale` estimate. At present this argument is ignored.
`interval`	A character string indicating if prediction intervals or a confidence interval on the mean responses are to be calculated. At present this argument is ignored.
`level`	A numeric scalar between 0 and 1 giving the confidence level for the intervals (if any) to be calculated. At present this argument is ignored.
`...`	Additional optional arguments. At present no optional arguments are used.

Value

`predict.nls` produces a vector of predictions. When implemented, `interval` will produce a matrix of predictions and bounds with column names `fit`, `lwr`, and `upr`. When implemented, if `se.fit` is `TRUE`, a list with the following components will be returned:

`fit`	vector or matrix as above
`se.fit`	standard error of predictions
`residual.scale`	residual standard deviations
`df`	degrees of freedom for residual

Note

Variables are first looked for in `newdata` and then searched for in the usual way (which will include the environment of the formula used in the fit). A warning will be given if the variables found are not of the same length as those in `newdata` if it was supplied.

See Also

The model fitting function `nls`, `predict`.

Examples

```
require(graphics)

fm <- nls(demand ~ SSasympOrig(Time, A, lrc), data = BOD)
predict(fm)              # fitted values at observed times
## Form data plot and smooth line for the predictions
opar <- par(las = 1)
plot(demand ~ Time, data = BOD, col = 4,
     main = "BOD data and fitted first-order curve",
     xlim = c(0,7), ylim = c(0, 20) )
tt <- seq(0, 8, length = 101)
lines(tt, predict(fm, list(Time = tt)))
par(opar)
```

---

predict.smooth.spline

Predict from Smoothing Spline Fit

Description

Predict a smoothing spline fit at new points, return the derivative if desired. The predicted fit is linear beyond the original data.

Usage

```
## S3 method for class 'smooth.spline':
predict(object, x, deriv = 0, ...)
```

Arguments

object	a fit from smooth.spline.
x	the new values of x.
deriv	integer; the order of the derivative required.
...	further arguments passed to or from other methods.

Value

A list with components

x	The input x.
y	The fitted values or derivatives at x.

See Also

smooth.spline

Examples

```
require(graphics)

attach(cars)
cars.spl <- smooth.spline(speed, dist, df=6.4)


## "Proof" that the derivatives are okay, by comparing with approximation
diff.quot <- function(x,y) {
  ## Difference quotient (central differences where available)
  n <- length(x); i1 <- 1:2; i2 <- (n-1):n
  c(diff(y[i1]) / diff(x[i1]), (y[-i1] - y[-i2]) / (x[-i1] - x[-i2]),
    diff(y[i2]) / diff(x[i2]))
}

xx <- unique(sort(c(seq(0,30, by = .2), kn <- unique(speed))))
i.kn <- match(kn, xx)# indices of knots within xx
op <- par(mfrow = c(2,2))
plot(speed, dist, xlim = range(xx), main = "Smooth.spline & derivatives")
lines(pp <- predict(cars.spl, xx), col = "red")
points(kn, pp$y[i.kn], pch = 3, col="dark red")
mtext("s(x)", col = "red")
for(d in 1:3){
  n <- length(pp$x)
  plot(pp$x, diff.quot(pp$x,pp$y), type = 'l', xlab="x", ylab="",
       col = "blue", col.main = "red",
       main= paste("s",paste(rep("'",d), collapse=""),"(x)", sep=""))
  mtext("Difference quotient approx.(last)", col = "blue")
  lines(pp <- predict(cars.spl, xx, deriv = d), col = "red")

  points(kn, pp$y[i.kn], pch = 3, col="dark red")
  abline(h=0, lty = 3, col = "gray")
}
detach(); par(op)
```

preplot *Pre-computations for a Plotting Objeect*

Description

Compute an object to be used for plots relating to the given model object.

Usage

```
preplot(object, ...)
```

Arguments

`object`	a fitted model object.
`...`	additional arguments for specific methods.

Details

Only the generic function is currently provided in base R, but some add-on packages have methods. Principally here for S compatibility.

Value

An object set up to make a plot that describes `object`.

princomp *Principal Components Analysis*

Description

`princomp` performs a principal components analysis on the given numeric data matrix and returns the results as an object of class `princomp`.

Usage

```
princomp(x, ...)

## S3 method for class 'formula':
princomp(formula, data = NULL, subset, na.action, ...)

## Default S3 method:
princomp(x, cor = FALSE, scores = TRUE, covmat = NULL,
         subset = rep(TRUE, nrow(as.matrix(x))), ...)

## S3 method for class 'princomp':
predict(object, newdata, ...)
```

Arguments

formula	a formula with no response variable, referring only to numeric variables.
data	an optional data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector used to select rows (observations) of the data matrix x.
na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The 'factory-fresh' default is na.omit.
x	a numeric matrix or data frame which provides the data for the principal components analysis.
cor	a logical value indicating whether the calculation should use the correlation matrix or the covariance matrix. (The correlation matrix can only be used if there are no constant variables.)
scores	a logical value indicating whether the score on each principal component should be calculated.
covmat	a covariance matrix, or a covariance list as returned by cov.wt (and cov.mve or cov.mcd from package **MASS**). If supplied, this is used rather than the covariance matrix of x.
...	arguments passed to or from other methods. If x is a formula one might specify cor or scores.
object	Object of class inheriting from "princomp"
newdata	An optional data frame or matrix in which to look for variables with which to predict. If omitted, the scores are used. If the original fit used a formula or a data frame or a matrix with column names, newdata must contain columns with the same names. Otherwise it must contain the same number of columns, to be used in the same order.

Details

princomp is a generic function with "formula" and "default" methods.

The calculation is done using eigen on the correlation or covariance matrix, as determined by cor. This is done for compatibility with the S-PLUS result. A preferred method of calculation is to use svd on x, as is done in prcomp.

Note that the default calculation uses divisor N for the covariance matrix.

The print method for these objects prints the results in a nice format and the plot method produces a scree plot (screeplot). There is also a biplot method.

If x is a formula then the standard NA-handling is applied to the scores (if requested): see napredict.

princomp only handles so-called R-mode PCA, that is feature extraction of variables. If a data matrix is supplied (possibly via a formula) it is required that there are at least as many units as variables. For Q-mode PCA use prcomp.

Value

princomp returns a list with class "princomp" containing the following components:

sdev	the standard deviations of the principal components.

loadings	the matrix of variable loadings (i.e., a matrix whose columns contain the eigenvectors). This is of class "loadings": see loadings for its print method.
center	the means that were subtracted.
scale	the scalings applied to each variable.
n.obs	the number of observations.
scores	if scores = TRUE, the scores of the supplied data on the principal components. These are non-null only if x was supplied, and if covmat was also supplied if it was a covariance list. For the formula method, napredict is applied to handle the treatment of values omitted by the na.action.
call	the matched call.
na.action	If relevant.

Note

The signs of the columns of the loadings and scores are arbitrary, and so may differ between different programs for PCA, and even between different builds of R.

References

Mardia, K. V., J. T. Kent and J. M. Bibby (1979). *Multivariate Analysis*, London: Academic Press.

Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S*, Springer-Verlag.

See Also

summary.princomp, screeplot, biplot.princomp, prcomp, cor, cov, eigen.

Examples

```
require(graphics)

## The variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
(pc.cr <- princomp(USArrests))  # inappropriate
princomp(USArrests, cor = TRUE) # =^= prcomp(USArrests, scale=TRUE)
## Similar, but different:
## The standard deviations differ by a factor of sqrt(49/50)

summary(pc.cr <- princomp(USArrests, cor = TRUE))
loadings(pc.cr)  ## note that blank entries are small but not zero
plot(pc.cr) # shows a screeplot.
biplot(pc.cr)

## Formula interface
princomp(~ ., data = USArrests, cor = TRUE)
# NA-handling
USArrests[1, 2] <- NA
pc.cr <- princomp(~ Murder + Assault + UrbanPop,
                  data = USArrests, na.action=na.exclude, cor = TRUE)
pc.cr$scores
```

print.power.htest *Print method for power calculation object*

Description

Print object of class "power.htest" in nice layout.

Usage

```
## S3 method for class 'power.htest':
print(x, ...)
```

Arguments

x	Object of class "power.htest".
...	further arguments to be passed to or from methods.

Details

A power.htest object is just a named list of numbers and character strings, supplemented with method and note elements. The method is displayed as a title, the note as a footnote, and the remaining elements are given in an aligned 'name = value' format.

Value

none

Author(s)

Peter Dalgaard

See Also

power.t.test, power.prop.test

print.ts *Printing Time-Series Objects*

Description

Print method for time series objects.

Usage

```
## S3 method for class 'ts':
print(x, calendar, ...)
```

Arguments

x	a time series object.
calendar	enable/disable the display of information about month names, quarter names or year when printing. The default is TRUE for a frequency of 4 or 12, FALSE otherwise.
...	additional arguments to print.

Details

This is the print methods for objects inheriting from class "ts".

See Also

print, ts.

Examples

```
print(ts(1:10, frequency = 7, start = c(12, 2)), calendar = TRUE)
```

---

printCoefmat *Print Coefficient Matrices*

Description

Utility function to be used in higher-level print methods, such as print.summary.lm, print.summary.glm and print.anova. The goal is to provide a flexible interface with smart defaults such that often, only x needs to be specified.

Usage

```
printCoefmat(x, digits=max(3, getOption("digits") - 2),
             signif.stars = getOption("show.signif.stars"),
             signif.legend = signif.stars,
             dig.tst = max(1, min(5, digits - 1)),
             cs.ind = 1:k, tst.ind = k + 1, zap.ind = integer(0),
             P.values = NULL,
             has.Pvalue = nc >= 4 &&
                          substr(colnames(x)[nc],1,3) == "Pr(",
             eps.Pvalue = .Machine$double.eps,
             na.print = "NA", ...)
```

Arguments

x	a numeric matrix like object, to be printed.
digits	minimum number of significant digits to be used for most numbers.
signif.stars	logical; if TRUE, P-values are additionally encoded visually as 'significance stars' in order to help scanning of long coefficient tables. It defaults to the show.signif.stars slot of options.

`signif.legend` logical; if `TRUE`, a legend for the 'significance stars' is printed provided `signif.stars=TRUE`.

`dig.tst` minimum number of significant digits for the test statistics, see `tst.ind`.

`cs.ind` indices (integer) of column numbers which are (like) **c**oefficients and **s**tandard errors to be formatted together.

`tst.ind` indices (integer) of column numbers for test statistics.

`zap.ind` indices (integer) of column numbers which should be formatted by `zapsmall`, i.e., by 'zapping' values close to 0.

`P.values` logical or `NULL`; if `TRUE`, the last column of `x` is formatted by `format.pval` as P values. If `P.values = NULL`, the default, it is set to `TRUE` only if `options("show.coef.Pvalue")` is `TRUE` *and* `x` has at least 4 columns *and* the last column name of `x` starts with `"Pr("`.

`has.Pvalue` logical; if `TRUE`, the last column of `x` contains P values; in that case, it is printed if and only if `P.values` (above) is true.

`eps.Pvalue` number,..

`na.print` a character string to code `NA` values in printed output.

`...` further arguments for `print`.

Value

Invisibly returns its argument, `x`.

Author(s)

Martin Maechler

See Also

`print.summary.lm`, `format.pval`, `format`.

Examples

```
cmat <- cbind(rnorm(3, 10), sqrt(rchisq(3, 12)))
cmat <- cbind(cmat, cmat[,1]/cmat[,2])
cmat <- cbind(cmat, 2*pnorm(-cmat[,3]))
colnames(cmat) <- c("Estimate", "Std.Err", "Z value", "Pr(>z)")
printCoefmat(cmat[,1:3])
printCoefmat(cmat)
options(show.coef.Pvalues = FALSE)
printCoefmat(cmat, digits=2)
printCoefmat(cmat, digits=2, P.values = TRUE)
options(show.coef.Pvalues = TRUE)# revert
```

profile *Generic Function for Profiling Models*

Description

Investigates behavior of objective function near the solution represented by `fitted`.

See documentation on method functions for further details.

Usage

```
profile(fitted, ...)
```

Arguments

`fitted`	the original fitted model object.
`...`	additional parameters. See documentation on individual methods.

Value

A list with an element for each parameter being profiled. See the individual methods for further details.

See Also

`profile.nls`, `profile.glm` in package **MASS**, ...

For profiling R code, see `Rprof`.

profile.nls *Method for Profiling nls Objects*

Description

Investigates the profile log-likelihood function for a fitted model of class `"nls"`.

Usage

```
## S3 method for class 'nls':
profile(fitted, which = 1:npar, maxpts = 100, alphamax = 0.01,
        delta.t = cutoff/5, ...)
```

Arguments

`fitted`	the original fitted model object.
`which`	the original model parameters which should be profiled. This can be a numeric or character vector. By default, all non-linear parameters are profiled.
`maxpts`	maximum number of points to be used for profiling each parameter.
`alphamax`	highest significance level allowed for the profile t-statistics.
`delta.t`	suggested change on the scale of the profile t-statistics. Default value chosen to allow profiling at about 10 parameter values.
`...`	further arguments passed to or from other methods.

Details

The profile t-statistics is defined as the square root of change in sum-of-squares divided by residual standard error with an appropriate sign.

Value

A list with an element for each parameter being profiled. The elements are data-frames with two variables

`par.vals`	a matrix of parameter values for each fitted model.
`tau`	the profile t-statistics.

Author(s)

Of the original version, Douglas M. Bates and Saikat DebRoy

References

Bates, D. M. and Watts, D. G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley (chapter 6).

See Also

`nls`, `profile`, `plot.profile.nls`

Examples

```
# obtain the fitted object
fm1 <- nls(demand ~ SSasympOrig(Time, A, lrc), data = BOD)
# get the profile for the fitted model: default level is too extreme
pr1 <- profile(fm1, alpha = 0.05)
# profiled values for the two parameters
pr1$A
pr1$lrc
# see also example(plot.profile.nls)
```

`proj` *Projections of Models*

Description

`proj` returns a matrix or list of matrices giving the projections of the data onto the terms of a linear model. It is most frequently used for `aov` models.

Usage

```
proj(object, ...)

## S3 method for class 'aov':
proj(object, onedf = FALSE, unweighted.scale = FALSE, ...)

## S3 method for class 'aovlist':
```

```
proj(object, onedf = FALSE, unweighted.scale = FALSE, ...)

## Default S3 method:
proj(object, onedf = TRUE, ...)

## S3 method for class 'lm':
proj(object, onedf = FALSE, unweighted.scale = FALSE, ...)
```

Arguments

`object`	An object of class `"lm"` or a class inheriting from it, or an object with a similar structure including in particular components `qr` and `effects`.
`onedf`	A logical flag. If `TRUE`, a projection is returned for all the columns of the model matrix. If `FALSE`, the single-column projections are collapsed by terms of the model (as represented in the analysis of variance table).
`unweighted.scale`	If the fit producing `object` used weights, this determines if the projections correspond to weighted or unweighted observations.
`...`	Swallow and ignore any other arguments.

Details

A projection is given for each stratum of the object, so for `aov` models with an `Error` term the result is a list of projections.

Value

A projection matrix or (for multi-stratum objects) a list of projection matrices.

Each projection is a matrix with a row for each observations and either a column for each term (`onedf = FALSE`) or for each coefficient (`onedf = TRUE`). Projection matrices from the default method have orthogonal columns representing the projection of the response onto the column space of the Q matrix from the QR decomposition. The fitted values are the sum of the projections, and the sum of squares for each column is the reduction in sum of squares from fitting that column (after those to the left of it).

The methods for `lm` and `aov` models add a column to the projection matrix giving the residuals (the projection of the data onto the orthogonal complement of the model space).

Strictly, when `onedf = FALSE` the result is not a projection, but the columns represent sums of projections onto the columns of the model matrix corresponding to that term. In this case the matrix does not depend on the coding used.

Author(s)

The design was inspired by the S function of the same name described in Chambers *et al.* (1992).

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`aov`, `lm`, `model.tables`

Examples

```
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)
npk.aov <- aov(yield ~ block + N*P*K, npk)
proj(npk.aov)

## as a test, not particularly sensible
options(contrasts=c("contr.helmert", "contr.treatment"))
npk.aovE <- aov(yield ~  N*P*K + Error(block), npk)
proj(npk.aovE)
```

---

prop.test *Test of Equal or Given Proportions*

Description

`prop.test` can be used for testing the null that the proportions (probabilities of success) in several groups are the same, or that they equal certain given values.

Usage

```
prop.test(x, n, p = NULL,
          alternative = c("two.sided", "less", "greater"),
          conf.level = 0.95, correct = TRUE)
```

Arguments

`x`	a vector of counts of successes or a matrix with 2 columns giving the counts of successes and failures, respectively.
`n`	a vector of counts of trials; ignored if `x` is a matrix.
`p`	a vector of probabilities of success. The length of `p` must be the same as the number of groups specified by `x`, and its elements must be greater than 0 and less than 1.
`alternative`	a character string specifying the alternative hypothesis, must be one of `"two.sided"` (default), `"greater"` or `"less"`. You can specify just the initial letter. Only used for testing the null that a single proportion equals a given value, or that two proportions are equal; ignored otherwise.
`conf.level`	confidence level of the returned confidence interval. Must be a single number between 0 and 1. Only used when testing the null that a single proportion equals a given value, or that two proportions are equal; ignored otherwise.
`correct`	a logical indicating whether Yates' continuity correction should be applied.

Details

Only groups with finite numbers of successes and failures are used. Counts of successes and failures must be nonnegative and hence not greater than the corresponding numbers of trials which must be positive. All finite counts should be integers.

If `p` is `NULL` and there is more than one group, the null tested is that the proportions in each group are the same. If there are two groups, the alternatives are that the probability of success in the first group is less than, not equal to, or greater than the probability of success in the second group, as specified by `alternative`. A confidence interval for the difference of proportions with confidence level as specified by `conf.level` and clipped to $[-1, 1]$ is returned. Continuity correction is used only if it does not exceed the difference of the sample proportions in absolute value. Otherwise, if there are more than 2 groups, the alternative is always `"two.sided"`, the returned confidence interval is `NULL`, and continuity correction is never used.

If there is only one group, then the null tested is that the underlying probability of success is `p`, or .5 if `p` is not given. The alternative is that the probability of success if less than, not equal to, or greater than `p` or 0.5, respectively, as specified by `alternative`. A confidence interval for the underlying proportion with confidence level as specified by `conf.level` and clipped to $[0, 1]$ is returned. Continuity correction is used only if it does not exceed the difference between sample and null proportions in absolute value. The confidence interval is computed by inverting the score test.

Finally, if `p` is given and there are more than 2 groups, the null tested is that the underlying probabilities of success are those given by `p`. The alternative is always `"two.sided"`, the returned confidence interval is `NULL`, and continuity correction is never used.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of Pearson's chi-squared test statistic.
`parameter`	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
`p.value`	the p-value of the test.
`estimate`	a vector with the sample proportions `x/n`.
`conf.int`	a confidence interval for the true proportion if there is one group, or for the difference in proportions if there are 2 groups and `p` is not given, or `NULL` otherwise. In the cases where it is not `NULL`, the returned confidence interval has an asymptotic confidence level as specified by `conf.level`, and is appropriate to the specified alternative hypothesis.
`null.value`	the value of `p` if specified by the null, or `NULL` otherwise.
`alternative`	a character string describing the alternative.
`method`	a character string indicating the method used, and whether Yates' continuity correction was applied.
`data.name`	a character string giving the names of the data.

References

Wilson, E.B. (1927) Probable inference, the law of succession, and statistical inference. *J. Am. Stat. Assoc.*, **22**, 209–212.

Newcombe R.G. (1998) Two-Sided Confidence Intervals for the Single Proportion: Comparison of Seven Methods. *Statistics in Medicine* **17**, 857–872.

Newcombe R.G. (1998) Interval Estimation for the Difference Between Independent Proportions: Comparison of Eleven Methods. *Statistics in Medicine* **17**, 873–890.

See Also

binom.test for an *exact* test of a binomial hypothesis.

Examples

```
heads <- rbinom(1, size=100, prob = .5)
prop.test(heads, 100)          # continuity correction TRUE by default
prop.test(heads, 100, correct = FALSE)

## Data from Fleiss (1981), p. 139.
## H0: The null hypothesis is that the four populations from which
##     the patients were drawn have the same true proportion of smokers.
## A:  The alternative is that this proportion is different in at
##     least one of the populations.

smokers  <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
prop.test(smokers, patients)
```

---

prop.trend.test *Test for trend in proportions*

Description

Performs chi-squared test for trend in proportions, i.e., a test asymptotically optimal for local alternatives where the log odds vary in proportion with score. By default, score is chosen as the group numbers.

Usage

```
prop.trend.test(x, n, score = 1:length(x))
```

Arguments

x	Number of events
n	Number of trials
score	Group score

Value

An object of class "htest" with title, test statistic, p-value, etc.

Note

This really should get integrated with prop.test

Author(s)

Peter Dalgaard

See Also

prop.test

Examples

```
smokers  <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
prop.test(smokers, patients)
prop.trend.test(smokers, patients)
prop.trend.test(smokers, patients,c(0,0,0,1))
```

---

qqnorm *Quantile-Quantile Plots*

Description

`qqnorm` is a generic function the default method of which produces a normal QQ plot of the values in `y`. `qqline` adds a line to a normal quantile-quantile plot which passes through the first and third quartiles.

`qqplot` produces a QQ plot of two datasets.

Graphical parameters may be given as arguments to `qqnorm`, `qqplot` and `qqline`.

Usage

```
qqnorm(y, ...)
## Default S3 method:
qqnorm(y, ylim, main = "Normal Q-Q Plot",
       xlab = "Theoretical Quantiles", ylab = "Sample Quantiles",
       plot.it = TRUE, datax = FALSE, ...)

qqline(y, datax = FALSE, ...)

qqplot(x, y, plot.it = TRUE, xlab = deparse(substitute(x)),
       ylab = deparse(substitute(y)), ...)
```

Arguments

`x`	The first sample for `qqplot`.
`y`	The second or only data sample.
`xlab, ylab, main`	plot labels. The `xlab` and `ylab` refer to the y and x axes respectively if `datax = TRUE`.
`plot.it`	logical. Should the result be plotted?
`datax`	logical. Should data values be on the x-axis?
`ylim, ...`	graphical parameters.

Value

For `qqnorm` and `qqplot`, a list with components

`x`	The x coordinates of the points that were/would be plotted
`y`	The original `y` vector, i.e., the corresponding y coordinates *including* `NA`*s*.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

ppoints, used by qqnorm to generate approximations to expected order statistics for a normal distribution.

Examples

```
require(graphics)

y <- rt(200, df = 5)
qqnorm(y); qqline(y, col = 2)
qqplot(y, rt(300, df = 5))

qqnorm(precip, ylab = "Precipitation [in/yr] for 70 US cities")
```

---

quade.test *Quade Test*

Description

Performs a Quade test with unreplicated blocked data.

Usage

```
quade.test(y, ...)

## Default S3 method:
quade.test(y, groups, blocks, ...)

## S3 method for class 'formula':
quade.test(formula, data, subset, na.action, ...)
```

Arguments

y	either a numeric vector of data values, or a data matrix.
groups	a vector giving the group for the corresponding elements of y if this is a vector; ignored if y is a matrix. If not a factor object, it is coerced to one.
blocks	a vector giving the block for the corresponding elements of y if this is a vector; ignored if y is a matrix. If not a factor object, it is coerced to one.
formula	a formula of the form a ~ b | c, where a, b and c give the data values and corresponding groups and blocks, respectively.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

`quade.test` can be used for analyzing unreplicated complete block designs (i.e., there is exactly one observation in `y` for each combination of levels of `groups` and `blocks`) where the normality assumption may be violated.

The null hypothesis is that apart from an effect of `blocks`, the location parameter of `y` is the same in each of the `groups`.

If `y` is a matrix, `groups` and `blocks` are obtained from the column and row indices, respectively. `NA`'s are not allowed in `groups` or `blocks`; if `y` contains `NA`'s, corresponding blocks are removed.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of Quade's F statistic.
`parameter`	a vector with the numerator and denominator degrees of freedom of the approximate F distribution of the test statistic.
`p.value`	the p-value of the test.
`method`	the character string `"Quade test"`.
`data.name`	a character string giving the names of the data.

References

D. Quade (1979), Using weighted rankings in the analysis of complete blocks with additive block effects. *Journal of the American Statistical Association*, **74**, 680–683.

William J. Conover (1999), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 373–380.

See Also

`friedman.test`.

Examples

```
## Conover (1999, p. 375f):
## Numbers of five brands of a new hand lotion sold in seven stores
## during one week.
y <- matrix(c( 5,  4,  7, 10, 12,
               1,  3,  1,  0,  2,
              16, 12, 22, 22, 35,
               5,  4,  3,  5,  4,
              10,  9,  7, 13, 10,
              19, 18, 28, 37, 58,
              10,  7,  6,  8,  7),
            nrow = 7, byrow = TRUE,
            dimnames =
            list(Store = as.character(1:7),
                 Brand = LETTERS[1:5]))
y
quade.test(y)
```

`quantile` *Sample Quantiles*

Description

The generic function `quantile` produces sample quantiles corresponding to the given probabilities. The smallest observation corresponds to a probability of 0 and the largest to a probability of 1.

Usage

```
quantile(x, ...)

## Default S3 method:
quantile(x, probs = seq(0, 1, 0.25), na.rm = FALSE,
         names = TRUE, type = 7, ...)
```

Arguments

`x`	numeric vectors whose sample quantiles are wanted. Missing values are ignored.
`probs`	numeric vector of probabilities with values in $[0, 1]$. (As from R 2.8.0 values up to 2e-14 outside that range are accepted and moved to the nearby endpoint.
`na.rm`	logical; if true, any `NA` and `NaN`'s are removed from `x` before the quantiles are computed.
`names`	logical; if true, the result has a `names` attribute. Set to `FALSE` for speedup with many `probs`.
`type`	an integer between 1 and 9 selecting one of the nine quantile algorithms detailed below to be used.
`...`	further arguments passed to or from other methods.

Details

A vector of length `length(probs)` is returned; if `names = TRUE`, it has a `names` attribute.

`NA` and `NaN` values in `probs` are propagated to the result.

Types

`quantile` returns estimates of underlying distribution quantiles based on one or two order statistics from the supplied elements in `x` at probabilities in `probs`. One of the nine quantile algorithms discussed in Hyndman and Fan (1996), selected by `type`, is employed.

Sample quantiles of type i are defined by

$$Q_i(p) = (1 - \gamma)x_j + \gamma x_{j+1}$$

where $1 \leq i \leq 9$, $\frac{j-m}{n} \leq p < \frac{j-m+1}{n}$, x_j is the jth order statistic, n is the sample size, and m is a constant determined by the sample quantile type. Here γ depends on the fractional part of $g = np + m - j$.

For the continuous sample quantile types (4 through 9), the sample quantiles can be obtained by linear interpolation between the kth order statistic and $p(k)$:

$$p(k) = \frac{k - \alpha}{n - \alpha - \beta + 1}$$

where α and β are constants determined by the type. Further, $m = \alpha + p\,(1 - \alpha - \beta)$, and $\gamma = g$.

Discontinuous sample quantile types 1, 2, and 3

Type 1 Inverse of empirical distribution function.

Type 2 Similar to type 1 but with averaging at discontinuities.

Type 3 SAS definition: nearest even order statistic.

Continuous sample quantile types 4 through 9

Type 4 $p(k) = \frac{k}{n}$. That is, linear interpolation of the empirical cdf.

Type 5 $p(k) = \frac{k-0.5}{n}$. That is a piecewise linear function where the knots are the values midway through the steps of the empirical cdf. This is popular amongst hydrologists.

Type 6 $p(k) = \frac{k}{n+1}$. Thus $p(k) = \mathrm{E}[F(x_k)]$. This is used by Minitab and by SPSS.

Type 7 $p(k) = \frac{k-1}{n-1}$. In this case, $p(k) = \mathrm{mode}[F(x_k)]$. This is used by S.

Type 8 $p(k) = \frac{k-\frac{1}{3}}{n+\frac{1}{3}}$. Then $p(k) \approx \mathrm{median}[F(x_k)]$. The resulting quantile estimates are approximately median-unbiased regardless of the distribution of `x`.

Type 9 $p(k) = \frac{k-\frac{3}{8}}{n+\frac{1}{4}}$. The resulting quantile estimates are approximately unbiased for the expected order statistics if `x` is normally distributed.

Hyndman and Fan (1996) recommend type 8. The default method is type 7, as used by S and by R < 2.0.0.

Author(s)

of the version used in R >= 2.0.0, Ivan Frohne and Rob J Hyndman.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Hyndman, R. J. and Fan, Y. (1996) Sample quantiles in statistical packages, *American Statistician*, **50**, 361–365.

See Also

`ecdf` for empirical distributions of which `quantile` is an inverse; `boxplot.stats` and `fivenum` for computing other versions of quartiles, etc.

Examples

```
quantile(x <- rnorm(1001))# Extremes & Quartiles by default
quantile(x,  probs=c(.1,.5,1,2,5,10,50, NA)/100)

### Compare different types
p <- c(0.1,0.5,1,2,5,10,50)/100
res <- matrix(as.numeric(NA), 9, 7)
for(type in 1:9) res[type, ] <- y <- quantile(x,  p, type=type)
dimnames(res) <- list(1:9, names(y))
round(res, 3)
```

---

r2dtable *Random 2-way Tables with Given Marginals*

Description

Generate random 2-way tables with given marginals using Patefield's algorithm.

Usage

```
r2dtable(n, r, c)
```

Arguments

n	a non-negative numeric giving the number of tables to be drawn.
r	a non-negative vector of length at least 2 giving the row totals, to be coerced to integer. Must sum to the same as c.
c	a non-negative vector of length at least 2 giving the column totals, to be coerced to integer.

Value

A list of length n containing the generated tables as its components.

References

Patefield, W. M. (1981) Algorithm AS159. An efficient method of generating r x c tables with given row and column totals. *Applied Statistics* **30**, 91–97.

Examples

```
## Fisher's Tea Drinker data.
TeaTasting <-
matrix(c(3, 1, 1, 3),
       nrow = 2,
       dimnames = list(Guess = c("Milk", "Tea"),
                       Truth = c("Milk", "Tea")))
## Simulate permutation test for independence based on the maximum
## Pearson residuals (rather than their sum).
rowTotals <- rowSums(TeaTasting)
colTotals <- colSums(TeaTasting)
nOfCases <- sum(rowTotals)
```

```
expected <- outer(rowTotals, colTotals, "*") / nOfCases
maxSqResid <- function(x) max((x - expected) ^ 2 / expected)
simMaxSqResid <-
    sapply(r2dtable(1000, rowTotals, colTotals), maxSqResid)
sum(simMaxSqResid >= maxSqResid(TeaTasting)) / 1000
## Fisher's exact test gives p = 0.4857 ...
```

`read.ftable` *Manipulate Flat Contingency Tables*

Description

Read, write and coerce 'flat' contingency tables.

Usage

```
read.ftable(file, sep = "", quote = "\"",
            row.var.names, col.vars, skip = 0)

write.ftable(x, file = "", quote = TRUE, append = FALSE,
             digits = getOption("digits"))

## S3 method for class 'ftable':
format(x, quote = TRUE, digits = getOption("digits"), ...)
```

Arguments

`file`	either a character string naming a file or a connection which the data are to be read from or written to. `""` indicates input from the console for reading and output to the console for writing.
`sep`	the field separator string. Values on each line of the file are separated by this string.
`quote`	a character string giving the set of quoting characters for `read.ftable`; to disable quoting altogether, use `quote=""`. For `write.table`, a logical indicating whether strings in the data will be surrounded by double quotes.
`row.var.names`	a character vector with the names of the row variables, in case these cannot be determined automatically.
`col.vars`	a list giving the names and levels of the column variables, in case these cannot be determined automatically.
`skip`	the number of lines of the data file to skip before beginning to read data.
`x`	an object of class `"ftable"`.
`append`	logical. If `TRUE` and `file` is the name of a file (and not a connection or `"|cmd"`), the output from `write.ftable` is appended to the file. If `FALSE`, the contents of `file` will be overwritten.
`digits`	an integer giving the number of significant digits to use for (the cell entries of) `x`.
`...`	further arguments to be passed to or from methods.

Details

`read.ftable` reads in a flat-like contingency table from a file. If the file contains the written representation of a flat table (more precisely, a header with all information on names and levels of column variables, followed by a line with the names of the row variables), no further arguments are needed. Similarly, flat tables with only one column variable the name of which is the only entry in the first line are handled automatically. Other variants can be dealt with by skipping all header information using `skip`, and providing the names of the row variables and the names and levels of the column variable using `row.var.names` and `col.vars`, respectively. See the examples below.

Note that flat tables are characterized by their 'ragged' display of row (and maybe also column) labels. If the full grid of levels of the row variables is given, one should instead use `read.table` to read in the data, and create the contingency table from this using `xtabs`.

`write.ftable` writes a flat table to a file, which is useful for generating 'pretty' ASCII representations of contingency tables.

References

Agresti, A. (1990) *Categorical data analysis*. New York: Wiley.

See Also

`ftable` for more information on flat contingency tables.

Examples

```
## Agresti (1990), page 157, Table 5.8.
## Not in ftable standard format, but o.k.
file <- tempfile()
cat("             Intercourse\n",
    "Race  Gender     Yes  No\n",
    "White Male        43 134\n",
    "      Female      26 149\n",
    "Black Male        29  23\n",
    "      Female      22  36\n",
    file = file)
file.show(file)
ft <- read.ftable(file)
ft
unlink(file)

## Agresti (1990), page 297, Table 8.16.
## Almost o.k., but misses the name of the row variable.
file <- tempfile()
cat("                       \"Tonsil Size\"\n",
    "            \"Not Enl.\" \"Enl.\" \"Greatly Enl.\"\n",
    "Noncarriers       497     560           269\n",
    "Carriers           19      29            24\n",
    file = file)
file.show(file)
ft <- read.ftable(file, skip = 2,
                  row.var.names = "Status",
                  col.vars = list("Tonsil Size" =
                      c("Not Enl.", "Enl.", "Greatly Enl.")))
ft
```

```
unlink(file)

ft22 <- ftable(Titanic, row.vars = 2:1, col.vars = 4:3)
write.ftable(ft22, quote = FALSE)
```

`rect.hclust` *Draw Rectangles Around Hierarchical Clusters*

Description

Draws rectangles around the branches of a dendrogram highlighting the corresponding clusters. First the dendrogram is cut at a certain level, then a rectangle is drawn around selected branches.

Usage

```
rect.hclust(tree, k = NULL, which = NULL, x = NULL, h = NULL,
            border = 2, cluster = NULL)
```

Arguments

`tree`	an object of the type produced by `hclust`.
`k, h`	Scalar. Cut the dendrogram such that either exactly `k` clusters are produced or by cutting at height `h`.
`which, x`	A vector selecting the clusters around which a rectangle should be drawn. `which` selects clusters by number (from left to right in the tree), `x` selects clusters containing the respective horizontal coordinates. Default is `which = 1:k`.
`border`	Vector with border colors for the rectangles.
`cluster`	Optional vector with cluster memberships as returned by `cutree(hclust.obj, k = k)`, can be specified for efficiency if already computed.

Value

(Invisibly) returns a list where each element contains a vector of data points contained in the respective cluster.

See Also

`hclust`, `identify.hclust`.

Examples

```
require(graphics)

hca <- hclust(dist(USArrests))
plot(hca)
rect.hclust(hca, k=3, border="red")
x <- rect.hclust(hca, h=50, which=c(2,7), border=3:4)
x
```

relevel *Reorder Levels of Factor*

Description

The levels of a factor are re-ordered so that the level specified by `ref` is first and the others are moved down. This is useful for `contr.treatment` contrasts which take the first level as the reference.

Usage

```
relevel(x, ref, ...)
```

Arguments

`x`	An unordered factor.
`ref`	The reference level.
`...`	Additional arguments for future methods.

Value

A factor of the same length as `x`.

See Also

`factor`, `contr.treatment`, `levels`, `reorder`.

Examples

```
warpbreaks$tension <- relevel(warpbreaks$tension, ref="M")
summary(lm(breaks ~ wool + tension, data=warpbreaks))
```

reorder.dendrogram *Reorder a Dendrogram*

Description

A method for the generic function `reorder`.

There are many different orderings of a dendrogram that are consistent with the structure imposed. This function takes a dendrogram and a vector of values and reorders the dendrogram in the order of the supplied vector, maintaining the constraints on the dendrogram.

Usage

```
## S3 method for class 'dendrogram':
reorder(x, wts, agglo.FUN = sum, ...)
```

Arguments

x	the (dendrogram) object to be reordered
wts	numeric weights (arbitrary values) for reordering.
agglo.FUN	a function for weights agglomeration, see below.
...	additional arguments

Details

Using the weights wts, the leaves of the dendrogram are reordered so as to be in an order as consistent as possible with the weights. At each node, the branches are ordered in increasing weights where the weight of a branch is defined as $f(w_j)$ where f is agglo.FUN and w_j is the weight of the j-th sub branch).

Value

A dendrogram where each node has a further attribute value with its corresponding weight.

Author(s)

R. Gentleman and M. Maechler

See Also

reorder.

rev.dendrogram which simply reverses the nodes' order; heatmap, cophenetic.

Examples

```
require(graphics)

set.seed(123)
x <- rnorm(10)
hc <- hclust(dist(x))
dd <- as.dendrogram(hc)
dd.reorder <- reorder(dd, 10:1)
plot(dd, main = "random dendrogram 'dd'")

op <- par(mfcol = 1:2)
plot(dd.reorder, main = "reorder(dd, 10:1)")
plot(reorder(dd,10:1, agglo.FUN= mean),
     main = "reorder(dd, 10:1, mean)")
par(op)
```

---

reorder.factor *Reorder Levels of a Factor*

Description

reorder is a generic function. Its "factor" method reorders the levels of a factor depending on values of a second variable, usually numeric. The "character" method is a convenient alias.

Usage

```
reorder(x, ...)

## S3 method for class 'factor':
reorder(x, X, FUN = mean, ...,
        order = is.ordered(x))

## S3 method for class 'character':
reorder(x, X, FUN = mean, ...,
        order = is.ordered(x))
```

Arguments

`x`	a character vector or factor (possibly ordered) or whose levels will be reordered.
`X`	a vector of the same length as `x`, whose subset of values for each unique level of `x` determines the eventual order of that level.
`FUN`	a function whose first argument is a vector and returns a scalar, to be applied to each subset of `X` determined by the levels of `x`.
`...`	optional: extra arguments supplied to `FUN`
`order`	logical, whether return value will be an ordered factor rather than a factor.

Value

A factor or an ordered factor (depending on the value of `order`), with the order of the levels determined by `FUN` applied to `X` grouped by `x`. The levels are ordered such that the values returned by `FUN` are in increasing order.

Additionally, the values of `FUN` applied to the subsets of `X` (in the original order of the levels of `x`) is returned as the `"scores"` attribute.

Author(s)

Deepayan Sarkar ⟨deepayan.sarkar@r-project.org⟩

See Also

`reorder.dendrogram`, `levels`, `relevel`.

Examples

```
require(graphics)

bymedian <- with(InsectSprays, reorder(spray, count, median))
boxplot(count ~ bymedian, data = InsectSprays,
        xlab = "Type of spray", ylab = "Insect count",
        main = "InsectSprays data", varwidth = TRUE,
        col = "lightgray")
```

`replications` *Number of Replications of Terms*

Description

Returns a vector or a list of the number of replicates for each term in the formula.

Usage

```
replications(formula, data=NULL, na.action)
```

Arguments

`formula`	a formula or a terms object or a data frame.
`data`	a data frame used to find the objects in `formula`.
`na.action`	function for handling missing values. Defaults to a `na.action` attribute of `data`, then a setting of the option `na.action`, or `na.fail` if that is not set.

Details

If `formula` is a data frame and `data` is missing, `formula` is used for `data` with the formula ~ ..

Value

A vector or list with one entry for each term in the formula giving the number(s) of replications for each level. If all levels are balanced (have the same number of replications) the result is a vector, otherwise it is a list with a component for each terms, as a vector, matrix or array as required.

A test for balance is `!is.list(replications(formula,data))`.

Author(s)

The design was inspired by the S function of the same name described in Chambers *et al.* (1992).

References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments.* Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`model.tables`

Examples

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)
replications(~ . - yield, npk)
```

---

reshape *Reshape Grouped Data*

Description

This function reshapes a data frame between 'wide' format with repeated measurements in separate columns of the same record and 'long' format with the repeated measurements in separate records.

Usage

```
reshape(data, varying = NULL, v.names = NULL, timevar = "time",
        idvar = "id", ids = 1:NROW(data),
        times = seq_along(varying[[1]]),
        drop = NULL, direction, new.row.names = NULL,
        sep = ".",
        split = if (sep==""){
            list(regexp="[A-Za-z][0-9]",include=TRUE)
        } else {
            list(regexp=sep, include= FALSE, fixed=TRUE)}
        )
```

Arguments

`data`	a data frame
`varying`	names of sets of variables in the wide format that correspond to single variables in long format ('time-varying'). This is canonically a list of vectors of variable names, but it can optionally be a matrix of names, or a single vector of names. In each case, the names can be replaced by indexes which are interpreted as referring to `names(data)`. See below for more details and options.
`v.names`	names of variables in the long format that correspond to multiple variables in the wide format. See below for details.
`timevar`	the variable in long format that differentiates multiple records from the same group or individual.
`idvar`	Names of one or more variables in long format that identify multiple records from the same group/individual. These variables may also be present in wide format
`ids`	the values to use for a newly created `idvar` variable in long format.

times	the values to use for a newly created timevar variable in long format. See below for details.
drop	a vector of names of variables to drop before reshaping
direction	character string, either "wide" to reshape to wide format, or "long" to reshape to long format.
new.row.names	logical; if TRUE and direction="wide", create new row names in long format from the values of the id and time variables.
sep	A character vector of length 1, indicating a separating character in the variable names in the wide format. This is used for guessing v.names and times arguments based on the names in varying. If sep=="", the split is just before the first numeral that follows an alphabetic character.
split	A list with three components, regexp, include, and (optionally) fixed. This allows an extended interface to variable name splitting. See below for details.

Details

The arguments to this function are described in terms of longitudinal data, as that is the application motivating the functions. A 'wide' longitudinal dataset will have one record for each individual with some time-constant variables that occupy single columns and some time-varying variables that occupy a column for each time point. In 'long' format there will be multiple records for each individual, with some variables being constant across these records and others varying across the records. A 'long' format dataset also needs a 'time' variable identifying which time point each record comes from and an 'id' variable showing which records refer to the same person.

If the data frame resulted from a previous reshape then the operation can be reversed simply by reshape(a). The direction argument is optional and the other arguments are stored as attributes on the data frame.

If direction="wide" and no varying or v.names arguments are supplied it is assumed that all variables except idvar and timevar are time-varying. They are all expanded into multiple variables in wide format.

If direction="long" the varying argument can be a vector of column names (or a corresponding index). The function will attempt to guess the v.names and times from these names. The default is variable names like x.1, x.2, where sep="." specifies to split at the dot and drop it from the name. To have alphabetic followed by numeric times use sep="".

Variable name splitting as described above is only attempted in the case where varying is an atomic vector, if it is a list or a matrix, v.names and times will generally need to be specified, although they will default to, respectively, the first variable name in each set, and sequential times.

Also, guessing is not attempted if v.names is given explicitly. Notice that the order of variables in varying is like x.1,y.1,x.2,y.2.

The split argument should not usually be necessary. The split$regexp component is passed to either strsplit() or regexp(), where the latter is used if split$include is TRUE, in which case the splitting occurs after the first character of the matched string. In the strsplit() case, the separator is not included in the result, and it is possible to specify fixed-string matching using split$fixed.

Value

The reshaped data frame with added attributes to simplify reshaping back to the original form.

See Also

stack, aperm; relist for reshaping the result of unlist.

Examples

```
summary(Indometh)
wide <- reshape(Indometh, v.names="conc", idvar="Subject",
                timevar="time", direction="wide")
wide

reshape(wide, direction="long")
reshape(wide, idvar="Subject", varying=list(2:12),
        v.names="conc", direction="long")

## times need not be numeric
df <- data.frame(id=rep(1:4,rep(2,4)),
                 visit=I(rep(c("Before","After"),4)),
                 x=rnorm(4), y=runif(4))
df
reshape(df, timevar="visit", idvar="id", direction="wide")
## warns that y is really varying
reshape(df, timevar="visit", idvar="id", direction="wide", v.names="x")

##  unbalanced 'long' data leads to NA fill in 'wide' form
df2 <- df[1:7,]
df2
reshape(df2, timevar="visit", idvar="id", direction="wide")

## Alternative regular expressions for guessing names
df3 <- data.frame(id=1:4, age=c(40,50,60,50), dose1=c(1,2,1,2),
                  dose2=c(2,1,2,1), dose4=c(3,3,3,3))
reshape(df3, direction="long", varying=3:5, sep="")

## an example that isn't longitudinal data
state.x77 <- as.data.frame(state.x77)
long <- reshape(state.x77, idvar="state", ids=row.names(state.x77),
                times=names(state.x77), timevar="Characteristic",
                varying=list(names(state.x77)), direction="long")

reshape(long, direction="wide")

reshape(long, direction="wide", new.row.names=unique(long$state))

## multiple id variables
df3 <- data.frame(school=rep(1:3,each=4), class=rep(9:10,6),
                  time=rep(c(1,1,2,2),3),
score=rnorm(12))
wide <- reshape(df3, idvar=c("school","class"), direction="wide")
wide
## transform back
reshape(wide)
```

residuals *Extract Model Residuals*

Description

`residuals` is a generic function which extracts model residuals from objects returned by modeling functions.

The abbreviated form `resid` is an alias for `residuals`. It is intended to encourage users to access object components through an accessor function rather than by directly referencing an object slot.

All object classes which are returned by model fitting functions should provide a `residuals` method. (Note that the method is for '`residuals`' and not '`resid`'.)

Methods can make use of `naresid` methods to compensate for the omission of missing values. The default, `nls` and `smooth.spline` methods do.

Usage

```
residuals(object, ...)
resid(object, ...)
```

Arguments

`object`	an object for which the extraction of model residuals is meaningful.
`...`	other arguments.

Value

Residuals extracted from the object `object`.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

`coefficients`, `fitted.values`, `glm`, `lm`.

influence.measures for standardized (`rstandard`) and studentized (`rstudent`) residuals.

runmed *Running Medians – Robust Scatter Plot Smoothing*

Description

Compute running medians of odd span. This is the 'most robust' scatter plot smoothing possible. For efficiency (and historical reason), you can use one of two different algorithms giving identical results.

Usage

```
runmed(x, k, endrule = c("median", "keep", "constant"),
       algorithm = NULL, print.level = 0)
```

Arguments

`x`	numeric vector, the 'dependent' variable to be smoothed.
`k`	integer width of median window; must be odd. Turlach had a default of `k <- 1 + 2 * min((n-1)%/% 2, ceiling(0.1*n))`. Use `k = 3` for 'minimal' robust smoothing eliminating isolated outliers.
`endrule`	character string indicating how the values at the beginning and the end (of the data) should be treated.
	`"keep"` keeps the first and last k_2 values at both ends, where k_2 is the half-bandwidth `k2 = k %/% 2`, i.e., `y[j] = x[j]` for $j \in \{1, \ldots, k_2; n - k_2 + 1, \ldots, n\}$;
	`"constant"` copies `median(y[1:k2])` to the first values and analogously for the last ones making the smoothed ends *constant*;
	`"median"` the default, smooths the ends by using symmetrical medians of subsequently smaller bandwidth, but for the very first and last value where Tukey's robust end-point rule is applied, see `smoothEnds`.
`algorithm`	character string (partially matching `"Turlach"` or `"Stuetzle"`) or the default `NULL`, specifying which algorithm should be applied. The default choice depends on `n = length(x)` and `k` where `"Turlach"` will be used for larger problems.
`print.level`	integer, indicating verboseness of algorithm; should rarely be changed by average users.

Details

Apart from the end values, the result `y = runmed(x, k)` simply has `y[j] = median(x[(j-k2):(j+k2)])` (k = 2*k2+1), computed very efficiently.

The two algorithms are internally entirely different:

"Turlach" is the Härdle-Steiger algorithm (see Ref.) as implemented by Berwin Turlach. A tree algorithm is used, ensuring performance $O(n \log k)$ where `n <- length(x)` which is asymptotically optimal.

"Stuetzle" is the (older) Stuetzle-Friedman implementation which makes use of median *updating* when one observation enters and one leaves the smoothing window. While this performs as $O(n \times k)$ which is slower asymptotically, it is considerably faster for small k or n.

Value

vector of smoothed values of the same length as `x` with an `attr`ibute `k` containing (the 'oddified') `k`.

Author(s)

Martin Maechler ⟨maechler@stat.math.ethz.ch⟩, based on Fortran code from Werner Stuetzle and S-plus and C code from Berwin Turlach.

References

Härdle, W. and Steiger, W. (1995) [Algorithm AS 296] Optimal median smoothing, *Applied Statistics* **44**, 258–264.

Jerome H. Friedman and Werner Stuetzle (1982) *Smoothing of Scatterplots*; Report, Dep. Statistics, Stanford U., Project Orion 003.

Martin Maechler (2003) Fast Running Medians: Finite Sample and Asymptotic Optimality; working paper available from the author.

See Also

`smoothEnds` which implements Tukey's end point rule and is called by default from `runmed(*, endrule = "median")`. `smooth` uses running medians of 3 for its compound smoothers.

Examples

```
require(graphics)

utils::example(nhtemp)
myNHT <- as.vector(nhtemp)
myNHT[20] <- 2 * nhtemp[20]
plot(myNHT, type="b", ylim = c(48,60), main = "Running Medians Example")
lines(runmed(myNHT, 7), col = "red")

## special: multiple y values for one x
plot(cars, main = "'cars' data and runmed(dist, 3)")
lines(cars, col = "light gray", type = "c")
with(cars, lines(speed, runmed(dist, k = 3), col = 2))

## nice quadratic with a few outliers
y <- ys <- (-20:20)^2
y [c(1,10,21,41)] <- c(150, 30, 400, 450)
all(y == runmed(y, 1)) # 1-neighbourhood <==> interpolation
plot(y) ## lines(y, lwd=.1, col="light gray")
lines(lowess(seq(y),y, f = .3), col = "brown")
lines(runmed(y, 7), lwd=2, col = "blue")
lines(runmed(y,11), lwd=2, col = "red")

## Lowess is not robust
y <- ys ; y[21] <- 6666 ; x <- seq(y)
col <- c("black", "brown","blue")
plot(y, col=col[1])
lines(lowess(x,y, f = .3), col = col[2])
lines(runmed(y, 7),      lwd=2, col = col[3])
legend(length(y),max(y), c("data", "lowess(y, f = 0.3)", "runmed(y, 7)"),
       xjust = 1, col = col, lty = c(0, 1,1), pch = c(1,NA,NA))
```

`scatter.smooth` *Scatter Plot with Smooth Curve Fitted by Loess*

Description

Plot and add a smooth curve computed by `loess` to a scatter plot.

Usage

```
scatter.smooth(x, y = NULL, span = 2/3, degree = 1,
    family = c("symmetric", "gaussian"),
    xlab = NULL, ylab = NULL,
    ylim = range(y, prediction$y, na.rm = TRUE),
    evaluation = 50, ...)

loess.smooth(x, y, span = 2/3, degree = 1,
    family = c("symmetric", "gaussian"), evaluation = 50, ...)
```

Arguments

x,y	the x and y arguments provide the x and y coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function xy.coords for details.
span	smoothness parameter for loess.
degree	degree of local polynomial used.
family	if "gaussian" fitting is by least-squares, and if family="symmetric" a re-descending M estimator is used.
xlab	label for x axis.
ylab	label for y axis.
ylim	the y limits of the plot.
evaluation	number of points at which to evaluate the smooth curve.
...	graphical parameters.

Details

loess.smooth is an auxiliary function which evaluates the loess smooth at evaluation equally spaced points covering the range of x.

Value

For scatter.smooth, none.

For loess.smooth, a list with two components, x (the grid of evaluation points) and y (the smoothed values at the grid points).

See Also

loess

Examples

```
require(graphics)

with(cars, scatter.smooth(speed, dist))
```

`screeplot` *Screeplots*

Description

`screeplot.default` plots the variances against the number of the principal component. This is also the `plot` method for classes `"princomp"` and `"prcomp"`.

Usage

```
## Default S3 method:
screeplot(x, npcs = min(10, length(x$sdev)),
          type = c("barplot", "lines"),
          main = deparse(substitute(x)), ...)
```

Arguments

`x`	an object containing a `sdev` component, such as that returned by `princomp()` and `prcomp()`.
`npcs`	the number of components to be plotted.
`type`	the type of plot.
`main, ...`	graphics parameters.

References

Mardia, K. V., J. T. Kent and J. M. Bibby (1979). *Multivariate Analysis*, London: Academic Press.

Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S*, Springer-Verlag.

See Also

`princomp` and `prcomp`.

Examples

```
require(graphics)

## The variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
(pc.cr <- princomp(USArrests, cor = TRUE))  # inappropriate
screeplot(pc.cr)

fit <- princomp(covmat=Harman74.cor)
screeplot(fit)
screeplot(fit, npcs=24, type="lines")
```

sd *Standard Deviation*

Description

This function computes the standard deviation of the values in `x`. If `na.rm` is `TRUE` then missing values are removed before computation proceeds. If `x` is a matrix or a data frame, a vector of the standard deviation of the columns is returned.

Usage

```
sd(x, na.rm = FALSE)
```

Arguments

`x`	a numeric vector, matrix or data frame. An object which is not a vector, matrix or data frame is coerced (if possible) by `as.vector`.
`na.rm`	logical. Should missing values be removed?

Details

Like `var` this uses denominator $n - 1$.

The standard deviation of a zero-length vector (after removal of `NA`s if `na.rm = TRUE`) is not defined and gives an error. The standard deviation of a length-one vector is `NA`.

See Also

`var` for its square, and `mad`, the most robust alternative.

Examples

```
sd(1:2) ^ 2
```

se.contrast *Standard Errors for Contrasts in Model Terms*

Description

Returns the standard errors for one or more contrasts in an `aov` object.

Usage

```
se.contrast(object, ...)
## S3 method for class 'aov':
se.contrast(object, contrast.obj,
            coef = contr.helmert(ncol(contrast))[, 1],
            data = NULL, ...)
```

Arguments

`object`	A suitable fit, usually from `aov`.
`contrast.obj`	The contrasts for which standard errors are requested. This can be specified via a list or via a matrix. A single contrast can be specified by a list of logical vectors giving the cells to be contrasted. Multiple contrasts should be specified by a matrix, each column of which is a numerical contrast vector (summing to zero).
`coef`	used when `contrast.obj` is a list; it should be a vector of the same length as the list with zero sum. The default value is the first Helmert contrast, which contrasts the first and second cell means specified by the list.
`data`	The data frame used to evaluate `contrast.obj`.
`...`	further arguments passed to or from other methods.

Details

Contrasts are usually used to test if certain means are significantly different; it can be easier to use `se.contrast` than compute them directly from the coefficients.

In multistratum models, the contrasts can appear in more than one stratum, in which case the standard errors are computed in the lowest stratum and adjusted for efficiencies and comparisons between strata. (See the comments in the note in the help for `aov` about using orthogonal contrasts.) Such standard errors are often conservative.

Suitable matrices for use with `coef` can be found by calling `contrasts` and indexing the columns by a factor.

Value

A vector giving the standard errors for each contrast.

See Also

`contrasts`, `model.tables`

Examples

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block = gl(6,4), N = factor(N), P = factor(P),
                  K = factor(K), yield = yield)
## Set suitable contrasts.
options(contrasts=c("contr.helmert", "contr.poly"))
npk.aov1 <- aov(yield ~ block + N + K, data=npk)
se.contrast(npk.aov1, list(N == "0", N == "1"), data=npk)
# or via a matrix
cont <- matrix(c(-1,1), 2, 1, dimnames=list(NULL, "N"))
se.contrast(npk.aov1, cont[N, , drop=FALSE]/12, data=npk)

## test a multi-stratum model
npk.aov2 <- aov(yield ~ N + K + Error(block/(N + K)), data=npk)
```

```
se.contrast(npk.aov2, list(N == "0", N == "1"))

## an example looking at an interaction contrast
## Dataset from R.E. Kirk (1995)
## 'Experimental Design: procedures for the behavioral sciences'
score <- c(12, 8,10, 6, 8, 4,10,12, 8, 6,10,14, 9, 7, 9, 5,11,12,
           7,13, 9, 9, 5,11, 8, 7, 3, 8,12,10,13,14,19, 9,16,14)
A <- gl(2, 18, labels=c("a1", "a2"))
B <- rep(gl(3, 6, labels=c("b1", "b2", "b3")), 2)
fit <- aov(score ~ A*B)
cont <- c(1, -1)[A] * c(1, -1, 0)[B]
sum(cont)       # 0
sum(cont*score) # value of the contrast
se.contrast(fit, as.matrix(cont))
(t.stat <- sum(cont*score)/se.contrast(fit, as.matrix(cont)))
summary(fit, split=list(B=1:2), expand.split = TRUE)
## t.stat^2 is the F value on the A:B: C1 line (with Helmert contrasts)
## Now look at all three interaction contrasts
cont <- c(1, -1)[A] * cbind(c(1, -1, 0), c(1, 0, -1), c(0, 1, -1))[B,]
se.contrast(fit, cont)  # same, due to balance.
rm(A,B,score)

## multi-stratum example where efficiencies play a role
utils::example(eff.aovlist)
fit <- aov(Yield ~ A + B * C + Error(Block), data = aovdat)
cont1 <- c(-1, 1)[A]/32  # Helmert contrasts
cont2 <- c(-1, 1)[B] * c(-1, 1)[C]/32
cont <- cbind(A=cont1, BC=cont2)
colSums(cont*Yield) # values of the contrasts
se.contrast(fit, as.matrix(cont))
## Not run:
# comparison with lme
library(nlme)
fit2 <- lme(Yield ~ A + B*C, random = ~1 | Block, data = aovdat)
summary(fit2)$tTable # same estimates, similar (but smaller) se's.
## End(Not run)
```

`selfStart` *Construct Self-starting Nonlinear Models*

Description

Construct self-starting nonlinear models.

Usage

```
selfStart(model, initial, parameters, template)
```

Arguments

`model`	a function object defining a nonlinear model or a nonlinear formula object of the form `~expression`.

initial	a function object, taking three arguments: mCall, data, and LHS, representing, respectively, a matched call to the function model, a data frame in which to interpret the variables in mCall, and the expression from the left-hand side of the model formula in the call to nls. This function should return initial values for the parameters in model.
parameters	a character vector specifying the terms on the right hand side of model for which initial estimates should be calculated. Passed as the namevec argument to the deriv function.
template	an optional prototype for the calling sequence of the returned object, passed as the function.arg argument to the deriv function. By default, a template is generated with the covariates in model coming first and the parameters in model coming last in the calling sequence.

Details

This function is generic; methods functions can be written to handle specific classes of objects.

Value

a function object of class "selfStart", for the formula method obtained by applying deriv to the right hand side of the model formula. An initial attribute (defined by the initial argument) is added to the function to calculate starting estimates for the parameters in the model automatically.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

nls

Examples

```
## self-starting logistic model

SSlogis <- selfStart(~ Asym/(1 + exp((xmid - x)/scal)),
  function(mCall, data, LHS)
  {
    xy <- sortedXyData(mCall[["x"]], LHS, data)
    if(nrow(xy) < 4) {
      stop("Too few distinct x values to fit a logistic")
    }
    z <- xy[["y"]]
    if (min(z) <= 0) { z <- z + 0.05 * max(z) } # avoid zeroes
    z <- z/(1.05 * max(z))              # scale to within unit height
    xy[["z"]] <- log(z/(1 - z))         # logit transformation
    aux <- coef(lm(x ~ z, xy))
    parameters(xy) <- list(xmid = aux[1], scal = aux[2])
    pars <- as.vector(coef(nls(y ~ 1/(1 + exp((xmid - x)/scal)),
                               data = xy, algorithm = "plinear")))
    value <- c(pars[3], pars[1], pars[2])
    names(value) <- mCall[c("Asym", "xmid", "scal")]
    value
  }, c("Asym", "xmid", "scal"))
```

```
# 'first.order.log.model' is a function object defining a first order
# compartment model
# 'first.order.log.initial' is a function object which calculates initial
# values for the parameters in 'first.order.log.model'

# self-starting first order compartment model
## Not run:
SSfol <- selfStart(first.order.log.model, first.order.log.initial)
## End(Not run)
```

`setNames` *Set the Names in an Object*

Description

This is a convenience function that sets the names on an object and returns the object. It is most useful at the end of a function definition where one is creating the object to be returned and would prefer not to store it under a name just so the names can be assigned.

Usage

```
setNames(object, nm)
```

Arguments

`object`	an object for which a `names` attribute will be meaningful
`nm`	a character vector of names to assign to the object

Value

An object of the same sort as `object` with the new names assigned.

Author(s)

Douglas M. Bates and Saikat DebRoy

See Also

`clearNames`

Examples

```
setNames( 1:3, c("foo", "bar", "baz") )
# this is just a short form of
tmp <- 1:3
names(tmp) <-  c("foo", "bar", "baz")
tmp
```

`shapiro.test` *Shapiro-Wilk Normality Test*

Description

Performs the Shapiro-Wilk test of normality.

Usage

```
shapiro.test(x)
```

Arguments

`x`	a numeric vector of data values. Missing values are allowed, but the number of non-missing values must be between 3 and 5000.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the Shapiro-Wilk statistic.
`p.value`	an approximate p-value for the test. This is said in Royston (1995) to be adequate for `p.value < 0.1`.
`method`	the character string `"Shapiro-Wilk normality test"`.
`data.name`	a character string giving the name(s) of the data.

Source

The algorithm used is a C translation of the Fortran code described in Royston (1995) and found at `http://lib.stat.cmu.edu/apstat/R94`. The calculation of the p value is exact for $n = 3$, otherwise approximations are used, separately for $4 \leq n \leq 11$ and $n \geq 12$.

References

Patrick Royston (1982) An extension of Shapiro and Wilk's W test for normality to large samples. *Applied Statistics*, **31**, 115–124.

Patrick Royston (1982) Algorithm AS 181: The W test for Normality. *Applied Statistics*, **31**, 176–180.

Patrick Royston (1995) Remark AS R94: A remark on Algorithm AS 181: The W test for normality. *Applied Statistics*, **44**, 547–551.

See Also

`qqnorm` for producing a normal quantile-quantile plot.

Examples

```
shapiro.test(rnorm(100, mean = 5, sd = 3))
shapiro.test(runif(100, min = 2, max = 4))
```

SignRank *Distribution of the Wilcoxon Signed Rank Statistic*

Description

Density, distribution function, quantile function and random generation for the distribution of the Wilcoxon Signed Rank statistic obtained from a sample with size n.

Usage

```
dsignrank(x, n, log = FALSE)
psignrank(q, n, lower.tail = TRUE, log.p = FALSE)
qsignrank(p, n, lower.tail = TRUE, log.p = FALSE)
rsignrank(nn, n)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.
nn	number of observations. If length(nn) > 1, the length is taken to be the number required.
n	number(s) of observations in the sample(s). A positive integer, or a vector of such integers.
log, log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

This distribution is obtained as follows. Let x be a sample of size n from a continuous distribution symmetric about the origin. Then the Wilcoxon signed rank statistic is the sum of the ranks of the absolute values x[i] for which x[i] is positive. This statistic takes values between 0 and $n(n+1)/2$, and its mean and variance are $n(n+1)/4$ and $n(n+1)(2n+1)/24$, respectively.

If either of the first two arguments is a vector, the recycling rule is used to do the calculations for all combinations of the two up to the length of the longer vector.

Value

dsignrank gives the density, psignrank gives the distribution function, qsignrank gives the quantile function, and rsignrank generates random deviates.

Author(s)

Kurt Hornik; efficiency improvement by Ivo Ugrina.

See Also

wilcox.test to calculate the statistic from data, find p values and so on.

dwilcox etc, for the distribution of *two-sample* Wilcoxon rank sum statistic.

Examples

```
require(graphics)

par(mfrow=c(2,2))
for(n in c(4:5,10,40)) {
  x <- seq(0, n*(n+1)/2, length=501)
  plot(x, dsignrank(x,n=n), type='l', main=paste("dsignrank(x,n=",n,")"))
}
```

---

simulate *Simulate Responses*

Description

Simulate one or more response vectors from the theoretical distribution corresponding to a fitted model object.

Usage

```
simulate(object, nsim, seed, ...)
```

Arguments

`object`	an object representing a fitted model.
`nsim`	number of response vectors to simulate. Defaults to 1.
`seed`	an object specifying if and how the random number generator should be initialized ('seeded'). For the "lm" method, either `NULL` or an integer that will be used in a call to `set.seed` before simulating the response vectors. If set, the value is saved as the `"seed"` attribute of the returned value. The default, `NULL` will not change the random generator state, and return `.Random.seed` as `"seed"` attribute, see below.
`...`	additional optional arguments.

Details

This is a generic function with a method for `lm` objects. Consult the individual modeling functions for details on how to use this function.

Value

Typically, a list of length `nsim` of simulated response vectors. When appropriate the result can be a data frame (which is a special type of list).

For the `"lm"` method, the result is a data frame with an attribute `"seed"` containing the `seed` argument and `as.list(RNGkind())` if `seed` was not `NULL`, or the value of `.Random.seed` before the simulation was started when `seed` was NULL as by default.

See Also

`fitted.values` and `residuals` for related methods; `glm`, `lm` for model fitting.

Examples

```
x <- 1:5
mod1 <- lm(c(1:3,7,6) ~ x)
S1 <- simulate(mod1, nsim = 4)
## repeat the simulation:
.Random.seed <- attr(S1, "seed")
identical(S1, simulate(mod1, nsim = 4))

S2 <- simulate(mod1, nsim = 200, seed = 101)
rowMeans(S2) # should be about
fitted(mod1)

## repeat identically:
(sseed <- attr(S2, "seed")) # seed; RNGkind as attribute
stopifnot(identical(S2, simulate(mod1, nsim = 200, seed = sseed)))

## To be sure about the proper RNGkind, e.g., after
RNGversion("2.7.0")
## first set the RNG kind, then simulate
do.call(RNGkind, attr(sseed, "kind"))
identical(S2, simulate(mod1, nsim = 200, seed = sseed))
```

`smooth` *Tukey's (Running Median) Smoothing*

Description

Tukey's smoothers, *3RS3R*, *3RSS*, *3R*, etc.

Usage

```
smooth(x, kind = c("3RS3R", "3RSS", "3RSR", "3R", "3", "S"),
       twiceit = FALSE, endrule = "Tukey", do.ends = FALSE)
```

Arguments

`x`	a vector or time series
`kind`	a character string indicating the kind of smoother required; defaults to `"3RS3R"`.
`twiceit`	logical, indicating if the result should be 'twiced'. Twicing a smoother $S(y)$ means $S(y) + S(y - S(y))$, i.e., adding smoothed residuals to the smoothed values. This decreases bias (increasing variance).
`endrule`	a character string indicating the rule for smoothing at the boundary. Either `"Tukey"` (default) or `"copy"`.
`do.ends`	logical, indicating if the 3-splitting of ties should also happen at the boundaries (ends). This is only used for `kind = "S"`.

Details

3 is Tukey's short notation for running medians of length **3**,
3R stands for **R**epeated *3* until convergence, and
S for **S**plitting of horizontal stretches of length 2 or 3.

Hence, *3RS3R* is a concatenation of 3R, S and 3R, *3RSS* similarly, whereas *3RSR* means first 3R and then (S and 3) **R**epeated until convergence – which can be bad.

Value

An object of class "tukeysmooth" (which has print and summary methods) and is a vector or time series containing the smoothed values with additional attributes.

Note

S and S-PLUS use a different (somewhat better) Tukey smoother in smooth(*). Note that there are other smoothing methods which provide rather better results. These were designed for hand calculations and may be used mainly for didactical purposes.

Since R version 1.2, smooth *does* really implement Tukey's end-point rule correctly (see argument endrule).

kind = "3RSR" has been the default till R-1.1, but it can have very bad properties, see the examples.

Note that repeated application of smooth(*) *does* smooth more, for the "3RS*" kinds.

References

Tukey, J. W. (1977). *Exploratory Data Analysis*, Reading Massachusetts: Addison-Wesley.

See Also

lowess; loess, supsmu and smooth.spline.

Examples

```
require(graphics)

## see also   demo(smooth) !

x1 <- c(4, 1, 3, 6, 6, 4, 1, 6, 2, 4, 2) # very artificial
(x3R <- smooth(x1, "3R")) # 2 iterations of "3"
smooth(x3R, kind = "S")

sm.3RS <- function(x, ...)
   smooth(smooth(x, "3R", ...), "S", ...)

y <- c(1,1, 19:1)
plot(y, main = "misbehaviour of \"3RSR\"", col.main = 3)
lines(sm.3RS(y))
lines(smooth(y))
lines(smooth(y, "3RSR"), col = 3, lwd = 2)# the horror

x <- c(8:10,10, 0,0, 9,9)
plot(x, main = "breakdown of  3R  and  S  and hence  3RSS")
matlines(cbind(smooth(x,"3R"),smooth(x,"S"), smooth(x,"3RSS"),smooth(x)))
```

```
presidents[is.na(presidents)] <- 0 # silly
summary(sm3 <- smooth(presidents, "3R"))
summary(sm2 <- smooth(presidents,"3RSS"))
summary(sm  <- smooth(presidents))

all.equal(c(sm2),c(smooth(smooth(sm3, "S"), "S"))) # 3RSS  === 3R S S
all.equal(c(sm), c(smooth(smooth(sm3, "S"), "3R")))# 3RS3R === 3R S 3R

plot(presidents, main = "smooth(presidents0, *) :  3R and default 3RS3R")
lines(sm3,col = 3, lwd = 1.5)
lines(sm, col = 2, lwd = 1.25)
```

`smooth.spline` *Fit a Smoothing Spline*

Description

Fits a cubic smoothing spline to the supplied data.

Usage

```
smooth.spline(x, y = NULL, w = NULL, df, spar = NULL,
              cv = FALSE, all.knots = FALSE, nknots = NULL,
              keep.data = TRUE, df.offset = 0, penalty = 1,
              control.spar = list())
```

Arguments

`x`	a vector giving the values of the predictor variable, or a list or a two-column matrix specifying x and y.
`y`	responses. If `y` is missing, the responses are assumed to be specified by `x`.
`w`	optional vector of weights of the same length as `x`; defaults to all 1.
`df`	the desired equivalent number of degrees of freedom (trace of the smoother matrix).
`spar`	smoothing parameter, typically (but not necessarily) in $(0, 1]$. The coefficient λ of the integral of the squared second derivative in the fit (penalized log likelihood) criterion is a monotone function of `spar`, see the details below.
`cv`	ordinary (`TRUE`) or 'generalized' cross-validation (GCV) when `FALSE`.
`all.knots`	if `TRUE`, all distinct points in `x` are used as knots. If `FALSE` (default), a subset of `x[]` is used, specifically `x[j]` where the `nknots` indices are evenly spaced in `1:n`, see also the next argument `nknots`.
`nknots`	integer giving the number of knots to use when `all.knots=FALSE`. Per default, this is less than n, the number of unique `x` values for $n > 49$.
`keep.data`	logical specifying if the input data should be kept in the result. If `TRUE` (as per default), fitted values and residuals are available from the result.
`df.offset`	allows the degrees of freedom to be increased by `df.offset` in the GCV criterion.
`penalty`	the coefficient of the penalty for degrees of freedom in the GCV criterion.

`control.spar` optional list with named components controlling the root finding when the smoothing parameter `spar` is computed, i.e., missing or `NULL`, see below.

Note that this is partly *experimental* and may change with general spar computation improvements!

low: lower bound for `spar`; defaults to -1.5 (used to implicitly default to 0 in R versions earlier than 1.4).

high: upper bound for `spar`; defaults to +1.5.

tol: the absolute precision (**tol**erance) used; defaults to 1e-4 (formerly 1e-3).

eps: the relative precision used; defaults to 2e-8 (formerly 0.00244).

trace: logical indicating if iterations should be traced.

maxit: integer giving the maximal number of iterations; defaults to 500.

Note that `spar` is only searched for in the interval $[low, high]$.

Details

The `x` vector should contain at least four distinct values. *Distinct* here means 'distinct after rounding to 6 significant digits', i.e., `x` will be transformed to `unique(sort(signif(x, 6)))`, and `y` and `w` are pooled accordingly.

The computational λ used (as a function of $s = spar$) is $\lambda = r * 256^{3s-1}$ where $r = tr(X'WX)/tr(\Sigma)$, Σ is the matrix given by $\Sigma_{ij} = \int B_i''(t)B_j''(t)dt$, X is given by $X_{ij} = B_j(x_i)$, W is the diagonal matrix of weights (scaled such that its trace is n, the original number of observations) and $B_k(.)$ is the k-th B-spline.

Note that with these definitions, $f_i = f(x_i)$, and the B-spline basis representation $f = Xc$ (i.e., c is the vector of spline coefficients), the penalized log likelihood is $L = (y - f)'W(y - f) + \lambda c'\Sigma c$, and hence c is the solution of the (ridge regression) $(X'WX + \lambda\Sigma)c = X'Wy$.

If `spar` is missing or `NULL`, the value of `df` is used to determine the degree of smoothing. If both are missing, leave-one-out cross-validation (ordinary or 'generalized' as determined by `cv`) is used to determine λ. Note that from the above relation, `spar` is $s = s0 + 0.0601 * \log \lambda$, which is intentionally *different* from the S-plus implementation of `smooth.spline` (where `spar` is proportional to λ). In R's ($\log \lambda$) scale, it makes more sense to vary `spar` linearly.

Note however that currently the results may become very unreliable for `spar` values smaller than about -1 or -2. The same may happen for values larger than 2 or so. Don't think of setting `spar` or the controls `low` and `high` outside such a safe range, unless you know what you are doing!

The 'generalized' cross-validation method will work correctly when there are duplicated points in `x`. However, it is ambiguous what leave-one-out cross-validation means with duplicated points, and the internal code uses an approximation that involves leaving out groups of duplicated points. `cv=TRUE` is best avoided in that case.

Value

An object of class `"smooth.spline"` with components

`x`	the *distinct* `x` values in increasing order, see the 'Details' above.
`y`	the fitted values corresponding to `x`.
`w`	the weights used at the unique values of `x`.
`yin`	the y values used at the unique `y` values.
`data`	only if `keep.data = TRUE`: itself a `list` with components `x`, `y` and `w` of the same length. These are the original $(x_i, y_i, w_i), i = 1, \ldots, n$, values where `data$x` may have repeated values and hence be longer than the above `x` component; see details.

lev	leverages, the diagonal values of the smoother matrix.
cv.crit	cross-validation score, 'generalized' or true, depending on cv.
pen.crit	penalized criterion
crit	the criterion value minimized in the underlying .Fortran routine 'sslvrg'.
df	equivalent degrees of freedom used. Note that (currently) this value may become quite imprecise when the true df is between and 1 and 2.
spar	the value of spar computed or given.
lambda	the value of λ corresponding to spar, see the details above.
iparms	named integer(3) vector where ..$ipars["iter"] gives number of spar computing iterations used.
fit	list for use by predict.smooth.spline, with components **knot:** the knot sequence (including the repeated boundary knots). **nk:** number of coefficients or number of 'proper' knots plus 2. **coef:** coefficients for the spline basis used. **min, range:** numbers giving the corresponding quantities of x.
call	the matched call.

Note

The default all.knots = FALSE and nknots = NULL entails using only $O(n^{0.2})$ knots instead of n for $n > 49$. This cuts speed and memory requirements, but not drastically anymore since R version 1.5.1 where it is only $O(n_k) + O(n)$ where n_k is the number of knots. In this case where not all unique x values are used as knots, the result is not a smoothing spline in the strict sense, but very close unless a small smoothing parameter (or large df) is used.

Author(s)

R implementation by B. D. Ripley and Martin Maechler (spar/lambda, etc).

This function is based on code in the GAMFIT Fortran program by T. Hastie and R. Tibshirani (http://lib.stat.cmu.edu/general/), which makes use of spline code by Finbarr O'Sullivan. Its design parallels the smooth.spline function of Chambers & Hastie (1992).

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*, Wadsworth & Brooks/Cole.

Green, P. J. and Silverman, B. W. (1994) *Nonparametric Regression and Generalized Linear Models: A Roughness Penalty Approach.* Chapman and Hall.

Hastie, T. J. and Tibshirani, R. J. (1990) *Generalized Additive Models.* Chapman and Hall.

See Also

predict.smooth.spline for evaluating the spline and its derivatives.

Examples

```
require(graphics)

attach(cars)
plot(speed, dist, main = "data(cars)  &  smoothing splines")
cars.spl <- smooth.spline(speed, dist)
(cars.spl)
## This example has duplicate points, so avoid cv=TRUE

lines(cars.spl, col = "blue")
lines(smooth.spline(speed, dist, df=10), lty=2, col = "red")
legend(5,120,c(paste("default [C.V.] => df =",round(cars.spl$df,1)),
               "s( * , df = 10)"), col = c("blue","red"), lty = 1:2,
       bg='bisque')
detach()

## Residual (Tukey Anscombe) plot:
plot(residuals(cars.spl) ~ fitted(cars.spl))
abline(h = 0, col="gray")

## consistency check:
stopifnot(all.equal(cars$dist,
                    fitted(cars.spl) + residuals(cars.spl)))

##-- artificial example
y18 <- c(1:3,5,4,7:3,2*(2:5),rep(10,4))
xx  <- seq(1,length(y18), len=201)
(s2  <- smooth.spline(y18)) # GCV
(s02 <- smooth.spline(y18, spar = 0.2))
plot(y18, main=deparse(s2$call), col.main=2)
lines(s2, col = "gray"); lines(predict(s2, xx), col = 2)
lines(predict(s02, xx), col = 3); mtext(deparse(s02$call), col = 3)

## The following shows the problematic behavior of 'spar' searching:
(s2  <- smooth.spline(y18, control = list(trace=TRUE,tol=1e-6, low= -1.5)))
(s2m <- smooth.spline(y18, cv = TRUE,
                      control = list(trace=TRUE,tol=1e-6, low= -1.5)))
## both above do quite similarly (Df = 8.5 +- 0.2)
```

`smoothEnds` *End Points Smoothing (for Running Medians)*

Description

Smooth end points of a vector `y` using subsequently smaller medians and Tukey's end point rule at the very end. (of odd span),

Usage

```
smoothEnds(y, k = 3)
```

Arguments

y	dependent variable to be smoothed (vector).
k	width of largest median window; must be odd.

Details

`smoothEnds` is used to only do the 'end point smoothing', i.e., change at most the observations closer to the beginning/end than half the window `k`. The first and last value are computed using *Tukey's end point rule*, i.e., `sm[1] = median(y[1], sm[2], 3*sm[2] - 2*sm[3])`.

Value

vector of smoothed values, the same length as `y`.

Author(s)

Martin Maechler

References

John W. Tukey (1977) *Exploratory Data Analysis*, Addison.

Velleman, P.F., and Hoaglin, D.C. (1981) *ABC of EDA (Applications, Basics, and Computing of Exploratory Data Analysis)*; Duxbury.

See Also

`runmed(*, endrule = "median")` which calls `smoothEnds()`.

Examples

```
require(graphics)

y <- ys <- (-20:20)^2
y [c(1,10,21,41)] <-  c(100, 30, 400, 470)
s7k <- runmed(y,7, endrule = "keep")
s7. <- runmed(y,7, endrule = "const")
s7m <- runmed(y,7)
col3 <- c("midnightblue","blue","steelblue")
plot(y, main = "Running Medians -- runmed(*, k=7, end.rule = X)")
lines(ys, col = "light gray")
matlines(cbind(s7k,s7.,s7m), lwd= 1.5, lty = 1, col = col3)
legend(1,470, paste("endrule",c("keep","constant","median"),sep=" = "),
       col = col3, lwd = 1.5, lty = 1)

stopifnot(identical(s7m, smoothEnds(s7k, 7)))
```

sortedXyData *Create a sortedXyData object*

Description

This is a constructor function for the class of `sortedXyData` objects. These objects are mostly used in the `initial` function for a self-starting nonlinear regression model, which will be of the `selfStart` class.

Usage

```
sortedXyData(x, y, data)
```

Arguments

`x`	a numeric vector or an expression that will evaluate in `data` to a numeric vector
`y`	a numeric vector or an expression that will evaluate in `data` to a numeric vector
`data`	an optional data frame in which to evaluate expressions for `x` and `y`, if they are given as expressions

Value

A `sortedXyData` object. This is a data frame with exactly two numeric columns, named `x` and `y`. The rows are sorted so the `x` column is in increasing order. Duplicate `x` values are eliminated by averaging the corresponding `y` values.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`selfStart`, `NLSstClosestX`, `NLSstLfAsymptote`, `NLSstRtAsymptote`

Examples

```
DNase.2 <- DNase[ DNase$Run == "2", ]
sortedXyData( expression(log(conc)), expression(density), DNase.2 )
```

spec.ar *Estimate Spectral Density of a Time Series from AR Fit*

Description

Fits an AR model to `x` (or uses the existing fit) and computes (and by default plots) the spectral density of the fitted model.

Usage

```
spec.ar(x, n.freq, order = NULL, plot = TRUE, na.action = na.fail,
        method = "yule-walker", ...)
```

Arguments

x	A univariate (not yet:or multivariate) time series or the result of a fit by ar.
n.freq	The number of points at which to plot.
order	The order of the AR model to be fitted. If omitted, the order is chosen by AIC.
plot	Plot the periodogram?
na.action	NA action function.
method	method for ar fit.
...	Graphical arguments passed to plot.spec.

Value

An object of class "spec". The result is returned invisibly if plot is true.

Warning

Some authors, for example Thomson (1990), warn strongly that AR spectra can be misleading.

Note

The multivariate case is not yet implemented.

References

Thompson, D.J. (1990) Time series analysis of Holocene climate data. *Phil. Trans. Roy. Soc. A* **330**, 601–616.

Venables, W.N. and Ripley, B.D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer. (Especially page 402.)

See Also

ar, spectrum.

Examples

```
require(graphics)

spec.ar(lh)

spec.ar(ldeaths)
spec.ar(ldeaths, method="burg")
```

`spec.pgram` *Estimate Spectral Density of a Time Series by a Smoothed Periodogram*

Description

`spec.pgram` calculates the periodogram using a fast Fourier transform, and optionally smooths the result with a series of modified Daniell smoothers (moving averages giving half weight to the end values).

Usage

```
spec.pgram(x, spans = NULL, kernel, taper = 0.1,
           pad = 0, fast = TRUE, demean = FALSE, detrend = TRUE,
           plot = TRUE, na.action = na.fail, ...)
```

Arguments

`x`	univariate or multivariate time series.
`spans`	vector of odd integers giving the widths of modified Daniell smoothers to be used to smooth the periodogram.
`kernel`	alternatively, a kernel smoother of class `"tskernel"`.
`taper`	specifies the proportion of data to taper. A split cosine bell taper is applied to this proportion of the data at the beginning and end of the series.
`pad`	proportion of data to pad. Zeros are added to the end of the series to increase its length by the proportion `pad`.
`fast`	logical; if `TRUE`, pad the series to a highly composite length.
`demean`	logical. If `TRUE`, subtract the mean of the series.
`detrend`	logical. If `TRUE`, remove a linear trend from the series. This will also remove the mean.
`plot`	plot the periodogram?
`na.action`	`NA` action function.
`...`	graphical arguments passed to `plot.spec`.

Details

The raw periodogram is not a consistent estimator of the spectral density, but adjacent values are asymptotically independent. Hence a consistent estimator can be derived by smoothing the raw periodogram, assuming that the spectral density is smooth.

The series will be automatically padded with zeros until the series length is a highly composite number in order to help the Fast Fourier Transform. This is controlled by the `fast` and not the `pad` argument.

The periodogram at zero is in theory zero as the mean of the series is removed (but this may be affected by tapering): it is replaced by an interpolation of adjacent values during smoothing, and no value is returned for that frequency.

Value

A list object of class "spec" (see spectrum) with the following additional components:

kernel	The kernel argument, or the kernel constructed from spans.
df	The distribution of the spectral density estimate can be approximated by a (scaled) chi square distribution with df degrees of freedom.
bandwidth	The equivalent bandwidth of the kernel smoother as defined by Bloomfield (1976, page 201).
taper	The value of the taper argument.
pad	The value of the pad argument.
detrend	The value of the detrend argument.
demean	The value of the demean argument.

The result is returned invisibly if plot is true.

Author(s)

Originally Martyn Plummer; kernel smoothing by Adrian Trapletti, synthesis by B.D. Ripley

References

Bloomfield, P. (1976) *Fourier Analysis of Time Series: An Introduction.* Wiley.

Brockwell, P.J. and Davis, R.A. (1991) *Time Series: Theory and Methods.* Second edition. Springer.

Venables, W.N. and Ripley, B.D. (2002) *Modern Applied Statistics with S.* Fourth edition. Springer. (Especially pp. 392–7.)

See Also

spectrum, spec.taper, plot.spec, fft

Examples

```
require(graphics)

## Examples from Venables & Ripley
spectrum(ldeaths)
spectrum(ldeaths, spans = c(3,5))
spectrum(ldeaths, spans = c(5,7))
spectrum(mdeaths, spans = c(3,3))
spectrum(fdeaths, spans = c(3,3))

## bivariate example
mfdeaths.spc <- spec.pgram(ts.union(mdeaths, fdeaths), spans = c(3,3))
# plots marginal spectra: now plot coherency and phase
plot(mfdeaths.spc, plot.type = "coherency")
plot(mfdeaths.spc, plot.type = "phase")

## now impose a lack of alignment
mfdeaths.spc <- spec.pgram(ts.intersect(mdeaths, lag(fdeaths, 4)),
   spans = c(3,3), plot = FALSE)
plot(mfdeaths.spc, plot.type = "coherency")
plot(mfdeaths.spc, plot.type = "phase")
```

```
stocks.spc <- spectrum(EuStockMarkets, kernel("daniell", c(30,50)),
                       plot = FALSE)
plot(stocks.spc, plot.type = "marginal") # the default type
plot(stocks.spc, plot.type = "coherency")
plot(stocks.spc, plot.type = "phase")

sales.spc <- spectrum(ts.union(BJsales, BJsales.lead),
                      kernel("modified.daniell", c(5,7)))
plot(sales.spc, plot.type = "coherency")
plot(sales.spc, plot.type = "phase")
```

`spec.taper` *Taper a Time Series by a Cosine Bell*

Description

Apply a cosine-bell taper to a time series.

Usage

```
spec.taper(x, p = 0.1)
```

Arguments

`x`	A univariate or multivariate time series
`p`	The proportion to be tapered at each end of the series, either a scalar (giving the proportion for all series) or a vector of the length of the number of series (giving the proportion for each series..

Details

The cosine-bell taper is applied to the first and last `p[i]` observations of time series `x[, i]`.

Value

A new time series object.

See Also

`spec.pgram`, `cpgram`

spectrum *Spectral Density Estimation*

Description

The spectrum function estimates the spectral density of a time series.

Usage

```
spectrum(x, ..., method = c("pgram", "ar"))
```

Arguments

x	A univariate or multivariate time series.
method	String specifying the method used to estimate the spectral density. Allowed methods are "pgram" (the default) and "ar".
...	Further arguments to specific spec methods or plot.spec.

Details

spectrum is a wrapper function which calls the methods spec.pgram and spec.ar.

The spectrum here is defined with scaling 1/frequency(x), following S-PLUS. This makes the spectral density a density over the range (-frequency(x)/2, +frequency(x)/2], whereas a more common scaling is 2π and range $(-0.5, 0.5]$ (e.g., Bloomfield) or 1 and range $(-\pi, \pi]$.

If available, a confidence interval will be plotted by plot.spec: this is asymmetric, and the width of the centre mark indicates the equivalent bandwidth.

Value

An object of class "spec", which is a list containing at least the following components:

freq	vector of frequencies at which the spectral density is estimated. (Possibly approximate Fourier frequencies.) The units are the reciprocal of cycles per unit time (and not per observation spacing): see 'Details' below.
spec	Vector (for univariate series) or matrix (for multivariate series) of estimates of the spectral density at frequencies corresponding to freq.
coh	NULL for univariate series. For multivariate time series, a matrix containing the *squared* coherency between different series. Column $i + (j-1) * (j-2)/2$ of coh contains the squared coherency between columns i and j of x, where $i < j$.
phase	NULL for univariate series. For multivariate time series a matrix containing the cross-spectrum phase between different series. The format is the same as coh.
series	The name of the time series.
snames	For multivariate input, the names of the component series.
method	The method used to calculate the spectrum.

The result is returned invisibly if plot is true.

Note

The default plot for objects of class "spec" is quite complex, including an error bar and default title, subtitle and axis labels. The defaults can all be overridden by supplying the appropriate graphical parameters.

Author(s)

Martyn Plummer, B.D. Ripley

References

Bloomfield, P. (1976) *Fourier Analysis of Time Series: An Introduction.* Wiley.

Brockwell, P. J. and Davis, R. A. (1991) *Time Series: Theory and Methods.* Second edition. Springer.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S-PLUS.* Fourth edition. Springer. (Especially pages 392–7.)

See Also

spec.ar, spec.pgram; plot.spec.

Examples

```
require(graphics)

## Examples from Venables & Ripley
## spec.pgram
par(mfrow=c(2,2))
spectrum(lh)
spectrum(lh, spans=3)
spectrum(lh, spans=c(3,3))
spectrum(lh, spans=c(3,5))

spectrum(ldeaths)
spectrum(ldeaths, spans=c(3,3))
spectrum(ldeaths, spans=c(3,5))
spectrum(ldeaths, spans=c(5,7))
spectrum(ldeaths, spans=c(5,7), log="dB", ci=0.8)

# for multivariate examples see the help for spec.pgram

## spec.ar
spectrum(lh, method="ar")
spectrum(ldeaths, method="ar")
```

---

splinefun *Interpolating Splines*

Description

Perform cubic (or Hermite) spline interpolation of given data points, returning either a list of points obtained by the interpolation or a *function* performing the interpolation.

Usage

```
splinefun(x, y = NULL, method = c("fmm", "periodic", "natural", "monoH.FC"),
          ties = mean)

spline(x, y = NULL, n = 3*length(x), method = "fmm",
       xmin = min(x), xmax = max(x), xout, ties = mean)

splinefunH(x, y, m)
```

Arguments

`x,y`	vectors giving the coordinates of the points to be interpolated. Alternatively a single plotting structure can be specified: see `xy.coords`.
`m`	(for `splinefunH()`): vector of *slopes* m_i at the points (x_i, y_i); these together determine the **H**ermite "spline" which is piecewise cubic, (only) *once* differentiable continuously.
`method`	specifies the type of spline to be used. Possible values are `"fmm"`, `"natural"`, `"periodic"` and `"monoH.FC"`.
`n`	if `xout` is left unspecified, interpolation takes place at `n` equally spaced points spanning the interval [`xmin`, `xmax`].
`xmin, xmax`	left-hand and right-hand endpoint of the interpolation interval (when `xout` is unspecified).
`xout`	an optional set of values specifying where interpolation is to take place.
`ties`	Handling of tied `x` values. Either a function with a single vector argument returning a single number result or the string `"ordered"`.

Details

The inputs can contain missing values which are deleted, so at least one complete `(x, y)` pair is required. If `method = "fmm"`, the spline used is that of Forsythe, Malcolm and Moler (an exact cubic is fitted through the four points at each end of the data, and this is used to determine the end conditions). Natural splines are used when `method = "natural"`, and periodic splines when `method = "periodic"`.

The new (R 2.8.0) method `"monoH.FC"` computes a *monotone* Hermite spline according to the method of Fritsch and Carlson. It does so by determining slopes such that the Hermite spline, determined by (x_i, y_i, m_i), is monotone (increasing or decreasing) **iff** the data are.

These interpolation splines can also be used for extrapolation, that is prediction at points outside the range of `x`. Extrapolation makes little sense for `method = "fmm"`; for natural splines it is linear using the slope of the interpolating curve at the nearest data point.

Value

`spline` returns a list containing components `x` and `y` which give the ordinates where interpolation took place and the interpolated values.

`splinefun` returns a function with formal arguments `x` and `deriv`, the latter defaulting to zero. This function can be used to evaluate the interpolating cubic spline (`deriv`=0), or its derivatives (`deriv`=1,2,3) at the points `x`, where the spline function interpolates the data points originally specified. This is often more useful than `spline`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Forsythe, G. E., Malcolm, M. A. and Moler, C. B. (1977) *Computer Methods for Mathematical Computations*.

Fritsch, F. N. and Carlson, R. E. (1980) Monotone piecewise cubic interpolation, *SIAM Journal on Numerical Analysis* **17**, 238–246.

See Also

`approx` and `approxfun` for constant and linear interpolation.

Package **splines**, especially `interpSpline` and `periodicSpline` for interpolation splines. That package also generates spline bases that can be used for regression splines.

`smooth.spline` for smoothing splines.

Examples

```
require(graphics)

op <- par(mfrow = c(2,1), mgp = c(2,.8,0), mar = .1+c(3,3,3,1))
n <- 9
x <- 1:n
y <- rnorm(n)
plot(x, y, main = paste("spline[fun](.) through", n, "points"))
lines(spline(x, y))
lines(spline(x, y, n = 201), col = 2)

y <- (x-6)^2
plot(x, y, main = "spline(.) -- 3 methods")
lines(spline(x, y, n = 201), col = 2)
lines(spline(x, y, n = 201, method = "natural"), col = 3)
lines(spline(x, y, n = 201, method = "periodic"), col = 4)
legend(6,25, c("fmm","natural","periodic"), col=2:4, lty=1)

y <- sin((x-0.5)*pi)
f <- splinefun(x, y)
ls(envir = environment(f))
splinecoef <- get("z", envir = environment(f))
curve(f(x), 1, 10, col = "green", lwd = 1.5)
points(splinecoef, col = "purple", cex = 2)
curve(f(x, deriv=1), 1, 10, col = 2, lwd = 1.5)
curve(f(x, deriv=2), 1, 10, col = 2, lwd = 1.5, n = 401)
curve(f(x, deriv=3), 1, 10, col = 2, lwd = 1.5, n = 401)
par(op)

## Manual spline evaluation --- demo the coefficients :
.x <- splinecoef$x
u <- seq(3,6, by = 0.25)
(ii <- findInterval(u, .x))
dx <- u - .x[ii]
f.u <- with(splinecoef,
            y[ii] + dx*(b[ii] + dx*(c[ii] + dx* d[ii])))
stopifnot(all.equal(f(u), f.u))
```

```
## An example with ties (non-unique  x values):
set.seed(1); x <- round(rnorm(30), 1); y <- sin(pi * x) + rnorm(30)/10
plot(x,y, main="spline(x,y)  when x has ties")
lines(spline(x,y, n= 201), col = 2)
## visualizes the non-unique ones:
tx <- table(x); mx <- as.numeric(names(tx[tx > 1]))
ry <- matrix(unlist(tapply(y, match(x,mx), range, simplify=FALSE)),
             ncol=2, byrow=TRUE)
segments(mx, ry[,1], mx, ry[,2], col = "blue", lwd = 2)

## An example of  monotone  interpolation
n <- 20
set.seed(11)
x. <- sort(runif(n)) ; y. <- cumsum(abs(rnorm(n)))
plot(x.,y.)
curve(splinefun(x.,y.)(x),                add=TRUE, col=2, n=1001)
curve(splinefun(x.,y., method="mono")(x), add=TRUE, col=3, n=1001)
legend("topleft", paste("splinefun( \"", c("fmm", "monoH.CS"), "\" )", sep=''),
        col=2:3, lty=1)
```

`SSasymp` *Asymptotic Regression Model*

Description

This `selfStart` model evaluates the asymptotic regression function and its gradient. It has an `initial` attribute that will evaluate initial estimates of the parameters `Asym`, `R0`, and `lrc` for a given set of data.

Usage

```
SSasymp(input, Asym, R0, lrc)
```

Arguments

`input`	a numeric vector of values at which to evaluate the model.
`Asym`	a numeric parameter representing the horizontal asymptote on the right side (very large values of `input`).
`R0`	a numeric parameter representing the response when `input` is zero.
`lrc`	a numeric parameter representing the natural logarithm of the rate constant.

Value

a numeric vector of the same length as `input`. It is the value of the expression `Asym+(R0-Asym)*exp(-exp(lrc)*input)`. If all of the arguments `Asym`, `R0`, and `lrc` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

nls, selfStart

Examples

```
Lob.329 <- Loblolly[ Loblolly$Seed == "329", ]
SSasymp( Lob.329$age, 100, -8.5, -3.2 )  # response only
Asym <- 100
resp0 <- -8.5
lrc <- -3.2
SSasymp( Lob.329$age, Asym, resp0, lrc ) # response and gradient
getInitial(height ~ SSasymp( age, Asym, resp0, lrc), data = Lob.329)
## Initial values are in fact the converged values
fm1 <- nls(height ~ SSasymp( age, Asym, resp0, lrc), data = Lob.329)
summary(fm1)
```

SSasympOff *Asymptotic Regression Model with an Offset*

Description

This selfStart model evaluates an alternative parametrization of the asymptotic regression function and the gradient with respect to those parameters. It has an initial attribute that creates initial estimates of the parameters Asym, lrc, and c0.

Usage

```
SSasympOff(input, Asym, lrc, c0)
```

Arguments

input	a numeric vector of values at which to evaluate the model.
Asym	a numeric parameter representing the horizontal asymptote on the right side (very large values of input).
lrc	a numeric parameter representing the natural logarithm of the rate constant.
c0	a numeric parameter representing the input for which the response is zero.

Value

a numeric vector of the same length as input. It is the value of the expression Asym*(1 - exp(-exp(lrc)*(input - c0))). If all of the arguments Asym, lrc, and c0 are names of objects, the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

nls, selfStart; example(SSasympOff) gives graph showing the SSasympOff parametrization, where ϕ_1 is Asymp, ϕ_3 is c0, and $t_{0.5}$ is

Examples

```
CO2.Qn1 <- CO2[CO2$Plant == "Qn1", ]
SSasympOff( CO2.Qn1$conc, 32, -4, 43 )  # response only
Asym <- 32; lrc <- -4; c0 <- 43
SSasympOff( CO2.Qn1$conc, Asym, lrc, c0 ) # response and gradient
getInitial(uptake ~ SSasympOff( conc, Asym, lrc, c0), data = CO2.Qn1)
## Initial values are in fact the converged values
fm1 <- nls(uptake ~ SSasympOff( conc, Asym, lrc, c0), data = CO2.Qn1)
summary(fm1)
```

SSasympOrig *Asymptotic Regression Model through the Origin*

Description

This selfStart model evaluates the asymptotic regression function through the origin and its gradient. It has an initial attribute that will evaluate initial estimates of the parameters Asym and lrc for a given set of data.

Usage

```
SSasympOrig(input, Asym, lrc)
```

Arguments

input	a numeric vector of values at which to evaluate the model.
Asym	a numeric parameter representing the horizontal asymptote.
lrc	a numeric parameter representing the natural logarithm of the rate constant.

Value

a numeric vector of the same length as input. It is the value of the expression Asym*(1 - exp(-exp(lrc)*input)). If all of the arguments Asym and lrc are names of objects, the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

nls, selfStart

Examples

```
Lob.329 <- Loblolly[ Loblolly$Seed == "329", ]
SSasympOrig( Lob.329$age, 100, -3.2 )  # response only
Asym <- 100; lrc <- -3.2
SSasympOrig( Lob.329$age, Asym, lrc ) # response and gradient
getInitial(height ~ SSasympOrig(age, Asym, lrc), data = Lob.329)
## Initial values are in fact the converged values
fm1 <- nls(height ~ SSasympOrig( age, Asym, lrc), data = Lob.329)
summary(fm1)
```

SSbiexp *Biexponential model*

Description

This `selfStart` model evaluates the biexponential model function and its gradient. It has an `initial` attribute that creates initial estimates of the parameters `A1`, `lrc1`, `A2`, and `lrc2`.

Usage

```
SSbiexp(input, A1, lrc1, A2, lrc2)
```

Arguments

`input`	a numeric vector of values at which to evaluate the model.
`A1`	a numeric parameter representing the multiplier of the first exponential.
`lrc1`	a numeric parameter representing the natural logarithm of the rate constant of the first exponential.
`A2`	a numeric parameter representing the multiplier of the second exponential.
`lrc2`	a numeric parameter representing the natural logarithm of the rate constant of the second exponential.

Value

a numeric vector of the same length as `input`. It is the value of the expression `A1*exp(-exp(lrc1)*input)+A2*exp(-exp(lrc2)*input)`. If all of the arguments `A1`, `lrc1`, `A2`, and `lrc2` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`nls`, `selfStart`

Examples

```
Indo.1 <- Indometh[Indometh$Subject == 1, ]
SSbiexp( Indo.1$time, 3, 1, 0.6, -1.3 )  # response only
A1 <- 3; lrc1 <- 1; A2 <- 0.6; lrc2 <- -1.3
SSbiexp( Indo.1$time, A1, lrc1, A2, lrc2 ) # response and gradient
getInitial(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2), data = Indo.1)
## Initial values are in fact the converged values
fm1 <- nls(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2), data = Indo.1)
summary(fm1)
```

SSD *SSD Matrix and Estimated Variance Matrix in Multivariate Models*

Description

Functions to compute matrix of residual sums of squares and products, or the estimated variance matrix for multivariate linear models.

Usage

```
# S3 method for class 'mlm'
SSD(object, ...)

# S3 methods for class 'SSD' and 'mlm'
estVar(object, ...)
```

Arguments

`object`	`object` of class `"mlm"`, or `"SSD"` in the case of `estVar`.
`...`	Unused

Value

`SSD()` returns a list of class `"SSD"` containing the following components

`SSD`	The residual sums of squares and products matrix
`df`	Degrees of freedom
`call`	Copied from `object`

`estVar` returns a matrix with the estimated variances and covariances.

See Also

`mauchly.test`, `anova.mlm`

Examples

```
# Lifted from Baron+Li:
# "Notes on the use of R for psychology experiments and questionnaires"
# Maxwell and Delaney, p. 497
reacttime <- matrix(c(
420, 420, 480, 480, 600, 780,
420, 480, 480, 360, 480, 600,
480, 480, 540, 660, 780, 780,
420, 540, 540, 480, 780, 900,
540, 660, 540, 480, 660, 720,
360, 420, 360, 360, 480, 540,
480, 480, 600, 540, 720, 840,
480, 600, 660, 540, 720, 900,
540, 600, 540, 480, 720, 780,
480, 420, 540, 540, 660, 780),
ncol = 6, byrow = TRUE,
dimnames=list(subj=1:10,
              cond=c("deg0NA", "deg4NA", "deg8NA",
                     "deg0NP", "deg4NP", "deg8NP")))

mlmfit <- lm(reacttime~1)
SSD(mlmfit)
estVar(mlmfit)
```

SSfol *First-order Compartment Model*

Description

This `selfStart` model evaluates the first-order compartment function and its gradient. It has an `initial` attribute that creates initial estimates of the parameters `lKe`, `lKa`, and `lCl`.

Usage

```
SSfol(Dose, input, lKe, lKa, lCl)
```

Arguments

`Dose`	a numeric value representing the initial dose.
`input`	a numeric vector at which to evaluate the model.
`lKe`	a numeric parameter representing the natural logarithm of the elimination rate constant.
`lKa`	a numeric parameter representing the natural logarithm of the absorption rate constant.
`lCl`	a numeric parameter representing the natural logarithm of the clearance.

Value

a numeric vector of the same length as `input`, which is the value of the expression `Dose * exp(lKe+lKa-lCl) * (exp(-exp(lKe)*input)-exp(-exp(lKa)*input)) / (exp(lKa)-exp(lKe))`.

If all of the arguments `lKe`, `lKa`, and `lCl` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

nls, selfStart

Examples

```
Theoph.1 <- Theoph[ Theoph$Subject == 1, ]
SSfol( Theoph.1$Dose, Theoph.1$Time, -2.5, 0.5, -3 )  # response only
lKe <- -2.5; lKa <- 0.5; lCl <- -3
SSfol( Theoph.1$Dose, Theoph.1$Time, lKe, lKa, lCl ) # response and gradient
getInitial(conc ~ SSfol(Dose, Time, lKe, lKa, lCl), data = Theoph.1)
## Initial values are in fact the converged values
fm1 <- nls(conc ~ SSfol(Dose, Time, lKe, lKa, lCl), data = Theoph.1)
summary(fm1)
```

SSfpl *Four-parameter Logistic Model*

Description

This selfStart model evaluates the four-parameter logistic function and its gradient. It has an initial attribute that will evaluate initial estimates of the parameters A, B, xmid, and scal for a given set of data.

Usage

```
SSfpl(input, A, B, xmid, scal)
```

Arguments

input	a numeric vector of values at which to evaluate the model.
A	a numeric parameter representing the horizontal asymptote on the left side (very small values of input).
B	a numeric parameter representing the horizontal asymptote on the right side (very large values of input).
xmid	a numeric parameter representing the input value at the inflection point of the curve. The value of SSfpl will be midway between A and B at xmid.
scal	a numeric scale parameter on the input axis.

Value

a numeric vector of the same length as input. It is the value of the expression A+(B-A)/(1+exp((xmid-input)/scal)). If all of the arguments A, B, xmid, and scal are names of objects, the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

nls, selfStart

Examples

```
Chick.1 <- ChickWeight[ChickWeight$Chick == 1, ]
SSfpl( Chick.1$Time, 13, 368, 14, 6 )  # response only
A <- 13; B <- 368; xmid <- 14; scal <- 6
SSfpl( Chick.1$Time, A, B, xmid, scal ) # response and gradient
getInitial(weight ~ SSfpl(Time, A, B, xmid, scal), data = Chick.1)
## Initial values are in fact the converged values
fm1 <- nls(weight ~ SSfpl(Time, A, B, xmid, scal), data = Chick.1)
summary(fm1)
```

SSgompertz *Gompertz Growth Model*

Description

This selfStart model evaluates the Gompertz growth model and its gradient. It has an initial attribute that creates initial estimates of the parameters Asym, b2, and b3.

Usage

```
SSgompertz(x, Asym, b2, b3)
```

Arguments

x	a numeric vector of values at which to evaluate the model.
Asym	a numeric parameter representing the asymptote.
b2	a numeric parameter related to the value of the function at x = 0
b3	a numeric parameter related to the scale the x axis.

Value

a numeric vector of the same length as input. It is the value of the expression Asym*exp(-b2*b3^x). If all of the arguments Asym, b2, and b3 are names of objects the gradient matrix with respect to these names is attached as an attribute named gradient.

Author(s)

Douglas Bates

See Also

nls, selfStart

Examples

```
DNase.1 <- subset(DNase, Run == 1)
SSlogis(log(DNase.1$conc), 4.5, 2.3, 0.7)  # response only
Asym <- 4.5; b2 <- 2.3; b3 <- 0.7
SSgompertz(log(DNase.1$conc), Asym, b2, b3 ) # response and gradient
getInitial(density ~ SSgompertz(log(conc), Asym, b2, b3),
           data = DNase.1)
## Initial values are in fact the converged values
fm1 <- nls(density ~ SSgompertz(log(conc), Asym, b2, b3),
           data = DNase.1)
summary(fm1)
```

SSlogis *Logistic Model*

Description

This `selfStart` model evaluates the logistic function and its gradient. It has an `initial` attribute that creates initial estimates of the parameters `Asym`, `xmid`, and `scal`.

Usage

```
SSlogis(input, Asym, xmid, scal)
```

Arguments

`input`	a numeric vector of values at which to evaluate the model.
`Asym`	a numeric parameter representing the asymptote.
`xmid`	a numeric parameter representing the `x` value at the inflection point of the curve. The value of `SSlogis` will be `Asym/2` at `xmid`.
`scal`	a numeric scale parameter on the `input` axis.

Value

a numeric vector of the same length as `input`. It is the value of the expression `Asym/(1+exp((xmid-input)/scal))`. If all of the arguments `Asym`, `xmid`, and `scal` are names of objects the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`nls`, `selfStart`

Examples

```
Chick.1 <- ChickWeight[ChickWeight$Chick == 1, ]
SSlogis( Chick.1$Time, 368, 14, 6 )  # response only
Asym <- 368; xmid <- 14; scal <- 6
SSlogis( Chick.1$Time, Asym, xmid, scal ) # response and gradient
getInitial(weight ~ SSlogis(Time, Asym, xmid, scal), data = Chick.1)
## Initial values are in fact the converged values
fm1 <- nls(weight ~ SSlogis(Time, Asym, xmid, scal), data = Chick.1)
summary(fm1)
```

SSmicmen *Michaelis-Menten Model*

Description

This `selfStart` model evaluates the Michaelis-Menten model and its gradient. It has an `initial` attribute that will evaluate initial estimates of the parameters `Vm` and `K`

Usage

```
SSmicmen(input, Vm, K)
```

Arguments

`input`	a numeric vector of values at which to evaluate the model.
`Vm`	a numeric parameter representing the maximum value of the response.
`K`	a numeric parameter representing the `input` value at which half the maximum response is attained. In the field of enzyme kinetics this is called the Michaelis parameter.

Value

a numeric vector of the same length as `input`. It is the value of the expression `Vm*input/(K+input)`. If both the arguments `Vm` and `K` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Jose Pinheiro and Douglas Bates

See Also

`nls`, `selfStart`

Examples

```
PurTrt <- Puromycin[ Puromycin$state == "treated", ]
SSmicmen( PurTrt$conc, 200, 0.05 )  # response only
Vm <- 200; K <- 0.05
SSmicmen( PurTrt$conc, Vm, K ) # response and gradient
getInitial(rate ~ SSmicmen(conc, Vm, K), data = PurTrt)
## Initial values are in fact the converged values
fm1 <- nls(rate ~ SSmicmen(conc, Vm, K), data = PurTrt)
summary( fm1 )
## Alternative call using the subset argument
fm2 <- nls(rate ~ SSmicmen(conc, Vm, K), data = Puromycin,
           subset = state == "treated")
summary(fm2)
```

SSweibull *Weibull growth curve model*

Description

This `selfStart` model evaluates the Weibull model for growth curve data and its gradient. It has an `initial` attribute that will evaluate initial estimates of the parameters `Asym`, `Drop`, `lrc`, and `pwr` for a given set of data.

Usage

```
SSweibull(x, Asym, Drop, lrc, pwr)
```

Arguments

`x`	a numeric vector of values at which to evaluate the model.
`Asym`	a numeric parameter representing the horizontal asymptote on the right side (very small values of `x`).
`Drop`	a numeric parameter representing the change from `Asym` to the `y` intercept.
`lrc`	a numeric parameter representing the natural logarithm of the rate constant.
`pwr`	a numeric parameter representing the power to which `x` is raised.

Details

This model is a generalization of the `SSasymp` model in that it reduces to `SSasymp` when `pwr` is unity.

Value

a numeric vector of the same length as `x`. It is the value of the expression `Asym-Drop*exp(-exp(lrc)*x^pwr)`. If all of the arguments `Asym`, `Drop`, `lrc`, and `pwr` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

Author(s)

Douglas Bates

References

Ratkowsky, David A. (1983), *Nonlinear Regression Modeling*, Dekker. (section 4.4.5)

See Also

`nls`, `selfStart`, `SSasymp`

Examples

```
Chick.6 <- subset(ChickWeight, (Chick == 6) & (Time > 0))
SSweibull(Chick.6$Time, 160, 115, -5.5, 2.5 )  # response only
Asym <- 160; Drop <- 115; lrc <- -5.5; pwr <- 2.5
SSweibull(Chick.6$Time, Asym, Drop, lrc, pwr)  # response and gradient
getInitial(weight ~ SSweibull(Time, Asym, Drop, lrc, pwr), data = Chick.6)
## Initial values are in fact the converged values
fm1 <- nls(weight ~ SSweibull(Time, Asym, Drop, lrc, pwr), data = Chick.6)
summary(fm1)
```

`start` *Encode the Terminal Times of Time Series*

Description

Extract and encode the times the first and last observations were taken. Provided only for compatibility with S version 2.

Usage

```
start(x, ...)
end(x, ...)
```

Arguments

`x`	a univariate or multivariate time-series, or a vector or matrix.
`...`	extra arguments for future methods.

Details

These are generic functions, which will use the `tsp` attribute of `x` if it exists. Their default methods decode the start time from the original time units, so that for a monthly series `1995.5` is represented as `c(1995, 7)`. For a series of frequency `f`, time `n+i/f` is presented as `c(n, i+1)` (even for `i = 0` and `f = 1`).

Warning

The representation used by `start` and `end` has no meaning unless the frequency is supplied.

See Also

`ts`, `time`, `tsp`.

stat.anova *GLM Anova Statistics*

Description

This is a utility function, used in lm and glm methods for anova(..., test != NULL) and should not be used by the average user.

Usage

```
stat.anova(table, test = c("Chisq", "F", "Cp"), scale, df.scale, n)
```

Arguments

table	numeric matrix as results from anova.glm(..., test=NULL).
test	a character string, matching one of "Chisq", "F" or "Cp".
scale	a residual mean square or other scale estimate to be used as the denominator in an F test.
df.scale	degrees of freedom corresponding to scale.
n	number of observations.

Value

A matrix which is the original table, augmented by a column of test statistics, depending on the test argument.

References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

anova.lm, anova.glm.

Examples

```
##-- Continued from '?glm':

print(ag <- anova(glm.D93))
stat.anova(ag$table, test = "Cp",
           scale = sum(resid(glm.D93, "pearson")^2)/4,
           df.scale = 4, n = 9)
```

stats-deprecated *Deprecated Functions in Stats package*

Description

These functions are provided for compatibility with older versions of R only, and may be defunct as soon as the next release.

Details

There are currently no deprecated functions in this package.

See Also

Deprecated

step *Choose a model by AIC in a Stepwise Algorithm*

Description

Select a formula-based model by AIC.

Usage

```
step(object, scope, scale = 0,
     direction = c("both", "backward", "forward"),
     trace = 1, keep = NULL, steps = 1000, k = 2, ...)
```

Arguments

object	an object representing a model of an appropriate class (mainly "lm" and "glm"). This is used as the initial model in the stepwise search.
scope	defines the range of models examined in the stepwise search. This should be either a single formula, or a list containing components upper and lower, both formulae. See the details for how to specify the formulae and how they are used.
scale	used in the definition of the AIC statistic for selecting the models, currently only for lm, aov and glm models. The default value, 0, indicates the scale should be estimated: see extractAIC.
direction	the mode of stepwise search, can be one of "both", "backward", or "forward", with a default of "both". If the scope argument is missing the default for direction is "backward".
trace	if positive, information is printed during the running of step. Larger values may give more detailed information.
keep	a filter function whose input is a fitted model object and the associated AIC statistic, and whose output is arbitrary. Typically keep will select a subset of the components of the object and return them. The default is not to keep anything.

`steps`	the maximum number of steps to be considered. The default is 1000 (essentially as many as required). It is typically used to stop the process early.
`k`	the multiple of the number of degrees of freedom used for the penalty. Only `k = 2` gives the genuine AIC: `k = log(n)` is sometimes referred to as BIC or SBC.
`...`	any additional arguments to `extractAIC`.

Details

`step` uses `add1` and `drop1` repeatedly; it will work for any method for which they work, and that is determined by having a valid method for `extractAIC`. When the additive constant can be chosen so that AIC is equal to Mallows' C_p, this is done and the tables are labelled appropriately.

The set of models searched is determined by the `scope` argument. The right-hand-side of its `lower` component is always included in the model, and right-hand-side of the model is included in the `upper` component. If `scope` is a single formula, it specifies the `upper` component, and the `lower` model is empty. If `scope` is missing, the initial model is used as the `upper` model.

Models specified by `scope` can be templates to update `object` as used by `update.formula`. So using `.` in a `scope` formula means 'what is already there', with `.^2` indicating all interactions of existing terms.

There is a potential problem in using `glm` fits with a variable `scale`, as in that case the deviance is not simply related to the maximized log-likelihood. The `"glm"` method for function `extractAIC` makes the appropriate adjustment for a `gaussian` family, but may need to be amended for other cases. (The `binomial` and `poisson` families have fixed `scale` by default and do not correspond to a particular maximum-likelihood problem for variable `scale`.)

Value

the stepwise-selected model is returned, with up to two additional components. There is an `"anova"` component corresponding to the steps taken in the search, as well as a `"keep"` component if the `keep=` argument was supplied in the call. The `"Resid. Dev"` column of the analysis of deviance table refers to a constant minus twice the maximized log likelihood: it will be a deviance only in cases where a saturated model is well-defined (thus excluding `lm`, `aov` and `survreg` fits, for example).

Warning

The model fitting must apply the models to the same dataset. This may be a problem if there are missing values and R's default of `na.action = na.omit` is used. We suggest you remove the missing values first.

Note

This function differs considerably from the function in S, which uses a number of approximations and does not in general compute the correct AIC.

This is a minimal implementation. Use `stepAIC` in package **MASS** for a wider range of object classes.

Author(s)

B. D. Ripley: `step` is a slightly simplified version of `stepAIC` in package **MASS** (Venables & Ripley, 2002 and earlier editions).

The idea of a `step` function follows that described in Hastie & Pregibon (1992); but the implementation in R is more general.

References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* New York: Springer (4th ed).

See Also

`stepAIC` in **MASS**, `add1`, `drop1`

Examples

```
utils::example(lm)
step(lm.D9)

summary(lm1 <- lm(Fertility ~ ., data = swiss))
slm1 <- step(lm1)
summary(slm1)
slm1$anova
```

`stepfun` *Step Function Class*

Description

Given the vectors $(x_1, \ldots, x_n)$ and $(y_0, y_1, \ldots, y_n)$ (one value more!), `stepfun(x,y,...)` returns an interpolating 'step' function, say `fn`. I.e., $fn(t) = c_i$ (constant) for $t \in (x_i, x_{i+1})$ and at the abscissa values, if (by default) `right = FALSE`, $fn(x_i) = y_i$ and for `right = TRUE`, $fn(x_i) = y_{i-1}$, for $i = 1, \ldots, n$.

The value of the constant c_i above depends on the 'continuity' parameter `f`. For the default, `right = FALSE, f = 0`, `fn` is a *cadlag* function, i.e., continuous at right, limit ('the point') at left. In general, c_i is interpolated in between the neighbouring y values, $c_i = (1-f)y_i + f \cdot y_{i+1}$. Therefore, for non-0 values of `f`, `fn` may no longer be a proper step function, since it can be discontinuous from both sides, unless `right = TRUE, f = 1` which is right-continuous.

Usage

```
stepfun(x, y, f = as.numeric(right), ties = "ordered",
        right = FALSE)

is.stepfun(x)
knots(Fn, ...)
as.stepfun(x, ...)

## S3 method for class 'stepfun':
print(x, digits = getOption("digits") - 2, ...)

## S3 method for class 'stepfun':
summary(object, ...)
```

Arguments

x	numeric vector giving the knots or jump locations of the step function for stepfun(). For the other functions, x is as object below.
y	numeric vector one longer than x, giving the heights of the function values *between* the x values.
f	a number between 0 and 1, indicating how interpolation outside the given x values should happen. See approxfun.
ties	Handling of tied x values. Either a function or the string "ordered". See approxfun.
right	logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.
Fn, object	an R object inheriting from "stepfun".
digits	number of significant digits to use, see print.
...	potentially further arguments (required by the generic).

Value

A function of class "stepfun", say fn. There are methods available for summarizing ("summary(.)"), representing ("print(.)") and plotting ("plot(.)", see plot.stepfun) "stepfun" objects.

The environment of fn contains all the information needed;

"x","y"	the original arguments
"n"	number of knots (x values)
"f"	continuity parameter
"yleft", "yright"	the function values *outside* the knots
"method"	(always == "constant", from approxfun(.)).

The knots are also available via knots(fn).

Author(s)

Martin Maechler, ⟨maechler@stat.math.ethz.ch⟩ with some basic code from Thomas Lumley.

See Also

ecdf for empirical distribution functions as special step functions and plot.stepfun for *plotting* step functions.

approxfun and splinefun.

Examples

```
y0 <- c(1,2,4,3)
sfun0  <- stepfun(1:3, y0, f = 0)
sfun.2 <- stepfun(1:3, y0, f = .2)
sfun1  <- stepfun(1:3, y0, f = 1)
sfun1c <- stepfun(1:3, y0, right=TRUE)# hence f=1
sfun0
summary(sfun0)
```

```
summary(sfun.2)

## look at the internal structure:
unclass(sfun0)
ls(envir = environment(sfun0))

x0 <- seq(0.5,3.5, by = 0.25)
rbind(x=x0, f.f0 = sfun0(x0), f.f02= sfun.2(x0),
      f.f1 = sfun1(x0), f.f1c = sfun1c(x0))
## Identities :
stopifnot(identical(y0[-1], sfun0 (1:3)),# right = FALSE
          identical(y0[-4], sfun1c(1:3)))# right = TRUE
```

`stl` *Seasonal Decomposition of Time Series by Loess*

Description

Decompose a time series into seasonal, trend and irregular components using `loess`, acronym STL.

Usage

```
stl(x, s.window, s.degree = 0,
    t.window = NULL, t.degree = 1,
    l.window = nextodd(period), l.degree = t.degree,
    s.jump = ceiling(s.window/10),
    t.jump = ceiling(t.window/10),
    l.jump = ceiling(l.window/10),
    robust = FALSE,
    inner = if(robust)  1 else 2,
    outer = if(robust) 15 else 0,
    na.action = na.fail)
```

Arguments

`x`	univariate time series to be decomposed. This should be an object of class `"ts"` with a frequency greater than one.
`s.window`	either the character string `"periodic"` or the span (in lags) of the loess window for seasonal extraction, which should be odd. This has no default.
`s.degree`	degree of locally-fitted polynomial in seasonal extraction. Should be zero or one.
`t.window`	the span (in lags) of the loess window for trend extraction, which should be odd. If `NULL`, the default, `nextodd(ceiling((1.5*period) / (1-(1.5/s.window))))`, is taken.
`t.degree`	degree of locally-fitted polynomial in trend extraction. Should be zero or one.
`l.window`	the span (in lags) of the loess window of the low-pass filter used for each subseries. Defaults to the smallest odd integer greater than or equal to `frequency(x)` which is recommended since it prevents competition between the trend and seasonal components. If not an odd integer its given value is increased to the next odd one.

`l.degree` degree of locally-fitted polynomial for the subseries low-pass filter. Must be 0 or 1.

`s.jump, t.jump, l.jump`
integers at least one to increase speed of the respective smoother. Linear interpolation happens between every `*.jump`th value.

`robust` logical indicating if robust fitting be used in the `loess` procedure.

`inner` integer; the number of 'inner' (backfitting) iterations; usually very few (2) iterations suffice.

`outer` integer; the number of 'outer' robustness iterations.

`na.action` action on missing values.

Details

The seasonal component is found by *loess* smoothing the seasonal sub-series (the series of all January values, ...); if `s.window = "periodic"` smoothing is effectively replaced by taking the mean. The seasonal values are removed, and the remainder smoothed to find the trend. The overall level is removed from the seasonal component and added to the trend component. This process is iterated a few times. The `remainder` component is the residuals from the seasonal plus trend fit.

Several methods for the resulting class `"stl"` objects, see, `plot.stl`.

Value

`stl` returns an object of class `"stl"` with components

`time.series` a multiple time series with columns `seasonal`, `trend` and `remainder`.

`weights` the final robust weights (all one if fitting is not done robustly).

`call` the matched call.

`win` integer (length 3 vector) with the spans used for the `"s"`, `"t"`, and `"l"` smoothers.

`deg` integer (length 3) vector with the polynomial degrees for these smoothers.

`jump` integer (length 3) vector with the 'jumps' (skips) used for these smoothers.

`ni` number of **i**nner iterations

`no` number of **o**uter robustness iterations

Note

This is similar to but not identical to the `stl` function in S-PLUS. The `remainder` component given by S-PLUS is the sum of the `trend` and `remainder` series from this function.

Author(s)

B.D. Ripley; Fortran code by Cleveland *et al.* (1990) from 'netlib'.

References

R. B. Cleveland, W. S. Cleveland, J.E. McRae, and I. Terpenning (1990) STL: A Seasonal-Trend Decomposition Procedure Based on Loess. *Journal of Official Statistics*, **6**, 3–73.

See Also

`plot.stl` for `stl` methods; `loess` in package **stats** (which is not actually used in `stl`).

`StructTS` for different kind of decomposition.

Examples

```
require(graphics)

plot(stl(nottem, "per"))
plot(stl(nottem, s.window = 4, t.window = 50, t.jump = 1))

plot(stllc <- stl(log(co2), s.window=21))
summary(stllc)
## linear trend, strict period.
plot(stl(log(co2), s.window="per", t.window=1000))

## Two STL plotted side by side :
        stmd <- stl(mdeaths, s.window = "per") # non-robust
summary(stmR <- stl(mdeaths, s.window = "per", robust = TRUE))
op <- par(mar = c(0, 4, 0, 3), oma = c(5, 0, 4, 0), mfcol = c(4, 2))
plot(stmd, set.pars=NULL, labels = NULL,
     main = "stl(mdeaths, s.w = \"per\",  robust = FALSE / TRUE )")
plot(stmR, set.pars=NULL)
# mark the 'outliers' :
(iO <- which(stmR $ weights  < 1e-8)) # 10 were considered outliers
sts <- stmR$time.series
points(time(sts)[iO], 0.8* sts[,"remainder"][iO], pch = 4, col = "red")
par(op)# reset
```

---

stlmethods *Methods for STL Objects*

Description

Methods for objects of class `stl`, typically the result of `stl`. The `plot` method does a multiple figure plot with some flexibility.

There are also (non-visible) `print` and `summary` methods.

Usage

```
## S3 method for class 'stl':
plot(x, labels = colnames(X),
     set.pars = list(mar = c(0, 6, 0, 6), oma = c(6, 0, 4, 0),
                     tck = -0.01, mfrow = c(nplot, 1)),
     main = NULL, range.bars = TRUE, ..., col.range = "light gray")
```

Arguments

`x`	`stl` object.
`labels`	character of length 4 giving the names of the component time-series.
`set.pars`	settings for `par(.)` when setting up the plot.

main	plot main title.
range.bars	logical indicating if each plot should have a bar at its right side which are of equal heights in user coordinates.
...	further arguments passed to or from other methods.
col.range	colour to be used for the range bars, if plotted. Note this appears after ... and so cannot be abbreviated.

See Also

plot.ts and stl, particularly for examples.

StructTS *Fit Structural Time Series*

Description

Fit a structural model for a time series by maximum likelihood.

Usage

```
StructTS(x, type = c("level", "trend", "BSM"), init = NULL,
         fixed = NULL, optim.control = NULL)
```

Arguments

x	a univariate numeric time series. Missing values are allowed.
type	the class of structural model. If omitted, a BSM is used for a time series with frequency(x) > 1, and a local trend model otherwise.
init	initial values of the variance parameters.
fixed	optional numeric vector of the same length as the total number of parameters. If supplied, only NA entries in fixed will be varied. Probably most useful for setting variances to zero.
optim.control	List of control parameters for optim. Method "L-BFGS-B" is used.

Details

Structural time series models are (linear Gaussian) state-space models for (univariate) time series based on a decomposition of the series into a number of components. They are specified by a set of error variances, some of which may be zero.

The simplest model is the *local level* model specified by type = "level". This has an underlying level μ_t which evolves by

$$\mu_{t+1} = \mu_t + \xi_t, \qquad \xi_t \sim N(0, \sigma_\xi^2)$$

The observations are

$$x_t = \mu_t + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$

There are two parameters, σ_ξ^2 and σ_ϵ^2. It is an ARIMA(0,1,1) model, but with restrictions on the parameter set.

The *local linear trend model*, `type = "trend"`, has the same measurement equation, but with a time-varying slope in the dynamics for μ_t, given by

$$\mu_{t+1} = \mu_t + \nu_t + \xi_t, \qquad \xi_t \sim N(0, \sigma_\xi^2)$$

$$\nu_{t+1} = \nu_t + \zeta_t, \qquad \zeta_t \sim N(0, \sigma_\zeta^2)$$

with three variance parameters. It is not uncommon to find $\sigma_\zeta^2 = 0$ (which reduces to the local level model) or $\sigma_\xi^2 = 0$, which ensures a smooth trend. This is a restricted ARIMA(0,2,2) model.

The *basic structural model*, `type = "BSM"`, is a local trend model with an additional seasonal component. Thus the measurement equation is

$$x_t = \mu_t + \gamma_t + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$

where γ_t is a seasonal component with dynamics

$$\gamma_{t+1} = -\gamma_t + \cdots + \gamma_{t-s+2} + \omega_t, \qquad \omega_t \sim N(0, \sigma_\omega^2)$$

The boundary case $\sigma_\omega^2 = 0$ corresponds to a deterministic (but arbitrary) seasonal pattern. (This is sometimes known as the 'dummy variable' version of the BSM.)

Value

A list of class `"StructTS"` with components:

`coef`	the estimated variances of the components.
`loglik`	the maximized log-likelihood. Note that as all these models are non-stationary this includes a diffuse prior for some observations and hence is not comparable with `arima` nor different types of structural models.
`data`	the time series `x`.
`residuals`	the standardized residuals.
`fitted`	a multiple time series with one component for the level, slope and seasonal components, estimated contemporaneously (that is at time t and not at the end of the series).
`call`	the matched call.
`series`	the name of the series `x`.
`code`	the `convergence` code returned by `optim`.
`model, model0`	Lists representing the Kalman Filter used in the fitting. See `KalmanLike`. `model0` is the initial state of the filter, `model` its final state.
`xtsp`	the `tsp` attributes of `x`.

Note

Optimization of structural models is a lot harder than many of the references admit. For example, the `AirPassengers` data are considered in Brockwell & Davis (1996): their solution appears to be a local maximum, but nowhere near as good a fit as that produced by `StructTS`. It is quite common to find fits with one or more variances zero, and this can include σ_ϵ^2.

References

Brockwell, P. J. & Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. Springer, New York. Sections 8.2 and 8.5.

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press.

Harvey, A. C. (1989) *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press.

Harvey, A. C. (1993) *Time Series Models*. 2nd Edition, Harvester Wheatsheaf.

See Also

`KalmanLike`, `tsSmooth`; `stl` for different kind of (seasonal) decomposition.

Examples

```
## see also JohnsonJohnson, Nile and AirPassengers
require(graphics)

trees <- window(treering, start=0)
(fit <- StructTS(trees, type = "level"))
plot(trees)
lines(fitted(fit), col = "green")
tsdiag(fit)

(fit <- StructTS(log10(UKgas), type = "BSM"))
par(mfrow = c(4, 1))
plot(log10(UKgas))
plot(cbind(fitted(fit), resids=resid(fit)), main = "UK gas consumption")

## keep some parameters fixed; trace optimizer:
StructTS(log10(UKgas), type = "BSM", fixed = c(0.1,0.001,NA,NA),
         optim.control = list(trace=TRUE))
```

---

summary.aov *Summarize an Analysis of Variance Model*

Description

Summarize an analysis of variance model.

Usage

```
## S3 method for class 'aov':
summary(object, intercept = FALSE, split,
        expand.split = TRUE, keep.zero.df = TRUE, ...)

## S3 method for class 'aovlist':
summary(object, ...)
```

Arguments

object	An object of class "aov" or "aovlist".
intercept	logical: should intercept terms be included?
split	an optional named list, with names corresponding to terms in the model. Each component is itself a list with integer components giving contrasts whose contributions are to be summed.
expand.split	logical: should the split apply also to interactions involving the factor?
keep.zero.df	logical: should terms with no degrees of freedom be included?
...	Arguments to be passed to or from other methods, for summary.aovlist including those for summary.aov.

Value

An object of class c("summary.aov", "listof") or "summary.aovlist" respectively.

For a fits with a single stratum the result will be a list of ANOVA tables, one for each response (even if there is only one response): the tables are of class "anova" inheriting from class "data.frame". They have columns "Df", "Sum Sq", "Mean Sq", as well as "F value" and "Pr(>F)" if there are non-zero residual degrees of freedom. There is a row for each term in the model, plus one for "Residuals" if there are any.

For multistratum fits the return value is a list of such summaries, one for each stratum.

Note

The use of expand.split = TRUE is little tested: it is always possible to set it to FALSE and specify exactly all the splits required.

See Also

aov, summary, model.tables, TukeyHSD

Examples

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,55.0,
           62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)
npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)

( npk.aov <- aov(yield ~ block + N*P*K, npk) )
summary(npk.aov)
coefficients(npk.aov)

# Cochran and Cox (1957, p.164)
# 3x3 factorial with ordered factors, each is average of 12.
CC <- data.frame(
    y = c(449, 413, 326, 409, 358, 291, 341, 278, 312)/12,
    P = ordered(gl(3, 3)), N = ordered(gl(3, 1, 9))
)
CC.aov <- aov(y ~ N * P, data = CC , weights = rep(12, 9))
```

```
summary(CC.aov)

# Split both main effects into linear and quadratic parts.
summary(CC.aov, split = list(N = list(L = 1, Q = 2),
                             P = list(L = 1, Q = 2)))

# Split only the interaction
summary(CC.aov, split = list("N:P" = list(L.L = 1, Q = 2:4)))

# split on just one var
summary(CC.aov, split = list(P = list(lin = 1, quad = 2)))
summary(CC.aov, split = list(P = list(lin = 1, quad = 2)),
        expand.split=FALSE)
```

`summary.glm` *Summarizing Generalized Linear Model Fits*

Description

These functions are all `methods` for class `glm` or `summary.glm` objects.

Usage

```
## S3 method for class 'glm':
summary(object, dispersion = NULL, correlation = FALSE,
        symbolic.cor = FALSE, ...)

## S3 method for class 'summary.glm':
print(x, digits = max(3, getOption("digits") - 3),
      symbolic.cor = x$symbolic.cor,
      signif.stars = getOption("show.signif.stars"), ...)
```

Arguments

`object`	an object of class `"glm"`, usually, a result of a call to `glm`.
`x`	an object of class `"summary.glm"`, usually, a result of a call to `summary.glm`.
`dispersion`	the dispersion parameter for the family used. Either a single numerical value or `NULL` (the default), when it is inferred from `object` (see 'Details').
`correlation`	logical; if `TRUE`, the correlation matrix of the estimated parameters is returned and printed.
`digits`	the number of significant digits to use when printing.
`symbolic.cor`	logical. If `TRUE`, print the correlations in a symbolic form (see `symnum`) rather than as numbers.
`signif.stars`	logical. If `TRUE`, 'significance stars' are printed for each coefficient.
`...`	further arguments passed to or from other methods.

Details

print.summary.glm tries to be smart about formatting the coefficients, standard errors, etc. and additionally gives 'significance stars' if signif.stars is TRUE. The coefficients component of the result gives the estimated coefficients and their estimated standard errors, together with their ratio. This third column is labelled t ratio if the dispersion is estimated, and z ratio if the dispersion is known (or fixed by the family). A fourth column gives the two-tailed p-value corresponding to the t or z ratio based on a Student t or Normal reference distribution. (It is possible that the dispersion is not known and there are no residual degrees of freedom from which to estimate it. In that case the estimate is NaN.)

Aliased coefficients are omitted in the returned object but restored by the print method.

Correlations are printed to two decimal places (or symbolically): to see the actual correlations print summary(object)$correlation directly.

The dispersion of a GLM is not used in the fitting process, but it is needed to find standard errors. If dispersion is not supplied or NULL, the dispersion is taken as 1 for the binomial and Poisson families, and otherwise estimated by the residual Chisquared statistic (calculated from cases with non-zero weights) divided by the residual degrees of freedom.

summary can be used with Gaussian glm fits to handle the case of a linear regression with known error variance, something not handled by summary.lm.

Value

summary.glm returns an object of class "summary.glm", a list with components

call	the component from object.
family	the component from object.
deviance	the component from object.
contrasts	the component from object.
df.residual	the component from object.
null.deviance	the component from object.
df.null	the component from object.
deviance.resid	the deviance residuals: see residuals.glm.
coefficients	the matrix of coefficients, standard errors, z-values and p-values. Aliased coefficients are omitted.
aliased	named logical vector showing if the original coefficients are aliased.
dispersion	either the supplied argument or the inferred/estimated dispersion if the latter is NULL.
df	a 3-vector of the rank of the model and the number of residual degrees of freedom, plus number of non-aliased coefficients.
cov.unscaled	the unscaled (dispersion = 1) estimated covariance matrix of the estimated coefficients.
cov.scaled	ditto, scaled by dispersion.
correlation	(only if correlation is true.) The estimated correlations of the estimated coefficients.
symbolic.cor	(only if correlation is true.) The value of the argument symbolic.cor.

See Also

glm, summary.

Examples

```
## --- Continuing the Example from  '?glm':

summary(glm.D93)
```

---

summary.lm *Summarizing Linear Model Fits*

Description

summary method for class "lm".

Usage

```
## S3 method for class 'lm':
summary(object, correlation = FALSE, symbolic.cor = FALSE, ...)

## S3 method for class 'summary.lm':
print(x, digits = max(3, getOption("digits") - 3),
      symbolic.cor = x$symbolic.cor,
      signif.stars = getOption("show.signif.stars"), ...)
```

Arguments

object	an object of class "lm", usually, a result of a call to lm.
x	an object of class "summary.lm", usually, a result of a call to summary.lm.
correlation	logical; if TRUE, the correlation matrix of the estimated parameters is returned and printed.
digits	the number of significant digits to use when printing.
symbolic.cor	logical. If TRUE, print the correlations in a symbolic form (see symnum) rather than as numbers.
signif.stars	logical. If TRUE, 'significance stars' are printed for each coefficient.
...	further arguments passed to or from other methods.

Details

print.summary.lm tries to be smart about formatting the coefficients, standard errors, etc. and additionally gives 'significance stars' if signif.stars is TRUE.

Correlations are printed to two decimal places (or symbolically): to see the actual correlations print summary(object)$correlation directly.

Value

The function `summary.lm` computes and returns a list of summary statistics of the fitted linear model given in `object`, using the components (list elements) `"call"` and `"terms"` from its argument, plus

`residuals` the *weighted* residuals, the usual residuals rescaled by the square root of the weights specified in the call to `lm`.

`coefficients` a $p \times 4$ matrix with columns for the estimated coefficient, its standard error, t-statistic and corresponding (two-sided) p-value. Aliased coefficients are omitted.

`aliased` named logical vector showing if the original coefficients are aliased.

`sigma` the square root of the estimated variance of the random error

$$\hat{\sigma}^2 = \frac{1}{n-p} \sum_i w_i R_i^2,$$

where R_i is the i-th residual, `residuals[i]`.

`df` degrees of freedom, a 3-vector $(p, n-p, p*)$, the last being the number of non-aliased coefficients.

`fstatistic` (for models including non-intercept terms) a 3-vector with the value of the F-statistic with its numerator and denominator degrees of freedom.

`r.squared` R^2, the 'fraction of variance explained by the model',

$$R^2 = 1 - \frac{\sum_i R_i^2}{\sum_i (y_i - y^*)^2},$$

where y^* is the mean of y_i if there is an intercept and zero otherwise.

`adj.r.squared` the above R^2 statistic '*adjusted*', penalizing for higher p.

`cov.unscaled` a $p \times p$ matrix of (unscaled) covariances of the $\hat{\beta}_j$, $j = 1, \ldots, p$.

`correlation` the correlation matrix corresponding to the above `cov.unscaled`, if `correlation = TRUE` is specified.

`symbolic.cor` (only if `correlation` is true.) The value of the argument `symbolic.cor`.

`na.action` from `object`, if present there.

See Also

The model fitting function `lm`, `summary`.

Function `coef` will extract the matrix of coefficients with standard errors, t-statistics and p-values.

Examples

```
##-- Continuing the  lm(.) example:
coef(lm.D90)# the bare coefficients
sld90 <- summary(lm.D90 <- lm(weight ~ group -1))# omitting intercept
sld90
coef(sld90)# much more
```

summary.manova *Summary Method for Multivariate Analysis of Variance*

Description

A summary method for class "manova".

Usage

```
## S3 method for class 'manova':
summary(object,
        test = c("Pillai", "Wilks", "Hotelling-Lawley", "Roy"),
        intercept = FALSE, tol = 1e-7, ...)
```

Arguments

object	An object of class "manova" or an aov object with multiple responses.
test	The name of the test statistic to be used. Partial matching is used so the name can be abbreviated.
intercept	logical. If TRUE, the intercept term is included in the table.
tol	tolerance to be used in deciding if the residuals are rank-deficient: see qr.
...	further arguments passed to or from other methods.

Details

The summary.manova method uses a multivariate test statistic for the summary table. Wilks' statistic is most popular in the literature, but the default Pillai–Bartlett statistic is recommended by Hand and Taylor (1987).

The table gives a transformation of the test statistic which has approximately an F distribution. The approximations used follow S-PLUS and SAS (the latter apart from some cases of the Hotelling–Lawley statistic), but many other distributional approximations exist: see Anderson (1984) and Krzanowski and Marriott (1994) for further references. All four approximate F statistics are the same when the term being tested has one degree of freedom, but in other cases that for the Roy statistic is an upper bound.

The tolerance tol is applied to the QR decomposition of the residual correlation matrix (unless some response has essentially zero residuals, when it is unscaled). Thus the default value guards against very highly correlated responses: it can be reduced but doing so will allow rather inaccurate results and it will normally be better to transform the responses to remove the high correlation.

Value

An object of class "summary.manova". If there is a positive residual degrees of freedom, this is a list with components

row.names	The names of the terms, the row names of the stats table if present.
SS	A named list of sums of squares and product matrices.
Eigenvalues	A matrix of eigenvalues.
stats	A matrix of the statistics, approximate F value, degrees of freedom and P value.

otherwise components row.names, SS and Df (degrees of freedom) for the terms (and not the residuals).

References

Anderson, T. W. (1994) *An Introduction to Multivariate Statistical Analysis.* Wiley.

Hand, D. J. and Taylor, C. C. (1987) *Multivariate Analysis of Variance and Repeated Measures.* Chapman and Hall.

Krzanowski, W. J. (1988) *Principles of Multivariate Analysis. A User's Perspective.* Oxford.

Krzanowski, W. J. and Marriott, F. H. C. (1994) *Multivariate Analysis. Part I: Distributions, Ordination and Inference.* Edward Arnold.

See Also

`manova`, `aov`

Examples

```
## Example on producing plastic film from Krzanowski (1998, p. 381)
tear <- c(6.5, 6.2, 5.8, 6.5, 6.5, 6.9, 7.2, 6.9, 6.1, 6.3,
          6.7, 6.6, 7.2, 7.1, 6.8, 7.1, 7.0, 7.2, 7.5, 7.6)
gloss <- c(9.5, 9.9, 9.6, 9.6, 9.2, 9.1, 10.0, 9.9, 9.5, 9.4,
           9.1, 9.3, 8.3, 8.4, 8.5, 9.2, 8.8, 9.7, 10.1, 9.2)
opacity <- c(4.4, 6.4, 3.0, 4.1, 0.8, 5.7, 2.0, 3.9, 1.9, 5.7,
             2.8, 4.1, 3.8, 1.6, 3.4, 8.4, 5.2, 6.9, 2.7, 1.9)
Y <- cbind(tear, gloss, opacity)
rate <- factor(gl(2,10), labels=c("Low", "High"))
additive <- factor(gl(2, 5, length=20), labels=c("Low", "High"))

fit <- manova(Y ~ rate * additive)
summary.aov(fit)           # univariate ANOVA tables
summary(fit, test="Wilks") # ANOVA table of Wilks' lambda
summary(fit)               # same F statistics as single-df terms
```

---

summary.nls *Summarizing Non-Linear Least-Squares Model Fits*

Description

`summary` method for class `"nls"`.

Usage

```
## S3 method for class 'nls':
summary(object, correlation = FALSE, symbolic.cor = FALSE, ...)

## S3 method for class 'summary.nls':
print(x, digits = max(3, getOption("digits") - 3),
      symbolic.cor = x$symbolic.cor,
      signif.stars = getOption("show.signif.stars"), ...)
```

Arguments

object	an object of class "nls".
x	an object of class "summary.nls", usually the result of a call to summary.nls.
correlation	logical; if TRUE, the correlation matrix of the estimated parameters is returned and printed.
digits	the number of significant digits to use when printing.
symbolic.cor	logical. If TRUE, print the correlations in a symbolic form (see symnum) rather than as numbers.
signif.stars	logical. If TRUE, 'significance stars' are printed for each coefficient.
...	further arguments passed to or from other methods.

Details

The distribution theory used to find the distribution of the standard errors and of the residual standard error (for t ratios) is based on linearization and is approximate, maybe very approximate.

print.summary.nls tries to be smart about formatting the coefficients, standard errors, etc. and additionally gives 'significance stars' if signif.stars is TRUE.

Correlations are printed to two decimal places (or symbolically): to see the actual correlations print summary(object)$correlation directly.

Value

The function summary.nls computes and returns a list of summary statistics of the fitted model given in object, using the component "formula" from its argument, plus

residuals	the *weighted* residuals, the usual residuals rescaled by the square root of the weights specified in the call to nls.
coefficients	a $p \times 4$ matrix with columns for the estimated coefficient, its standard error, t-statistic and corresponding (two-sided) p-value.
sigma	the square root of the estimated variance of the random error $$\hat{\sigma}^2 = \frac{1}{n-p}\sum_i R_i^2,$$ where R_i is the i-th weighted residual.
df	degrees of freedom, a 2-vector $(p, n-p)$. (Here and elsewhere n omits observations with zero weights.)
cov.unscaled	a $p \times p$ matrix of (unscaled) covariances of the parameter estimates.
correlation	the correlation matrix corresponding to the above cov.unscaled, if correlation = TRUE is specified and there are a non-zero number of residual degrees of freedom.
symbolic.cor	(only if correlation is true.) The value of the argument symbolic.cor.

See Also

The model fitting function nls, summary.

Function coef will extract the matrix of coefficients with standard errors, t-statistics and p-values.

summary.princomp *Summary method for Principal Components Analysis*

Description

The summary method for class "princomp".

Usage

```
## S3 method for class 'princomp':
summary(object, loadings = FALSE, cutoff = 0.1, ...)

## S3 method for class 'summary.princomp':
print(x, digits = 3, loadings = x$print.loadings,
      cutoff = x$cutoff, ...)
```

Arguments

object	an object of class "princomp", as from princomp().
loadings	logical. Should loadings be included?
cutoff	numeric. Loadings below this cutoff in absolute value are shown as blank in the output.
x	an object of class "summary.princomp".
digits	the number of significant digits to be used in listing loadings.
...	arguments to be passed to or from other methods.

Value

object with additional components cutoff and print.loadings.

See Also

princomp

Examples

```
summary(pc.cr <- princomp(USArrests, cor=TRUE))
print(summary(princomp(USArrests, cor=TRUE),
              loadings = TRUE, cutoff = 0.2), digits = 2)
```

`supsmu` *Friedman's SuperSmoother*

Description

Smooth the (x, y) values by Friedman's 'super smoother'.

Usage

```
supsmu(x, y, wt, span = "cv", periodic = FALSE, bass = 0)
```

Arguments

`x`	x values for smoothing
`y`	y values for smoothing
`wt`	case weights, by default all equal
`span`	the fraction of the observations in the span of the running lines smoother, or `"cv"` to choose this by leave-one-out cross-validation.
`periodic`	if `TRUE`, the x values are assumed to be in `[0, 1]` and of period 1.
`bass`	controls the smoothness of the fitted curve. Values of up to 10 indicate increasing smoothness.

Details

`supsmu` is a running lines smoother which chooses between three spans for the lines. The running lines smoothers are symmetric, with `k/2` data points each side of the predicted point, and values of `k` as `0.5 * n`, `0.2 * n` and `0.05 * n`, where `n` is the number of data points. If `span` is specified, a single smoother with span `span * n` is used.

The best of the three smoothers is chosen by cross-validation for each prediction. The best spans are then smoothed by a running lines smoother and the final prediction chosen by linear interpolation.

The FORTRAN code says: "For small samples (`n < 40`) or if there are substantial serial correlations between observations close in x-value, then a pre-specified fixed span smoother (`span > 0`) should be used. Reasonable span values are 0.2 to 0.4."

Cases with non-finite values of `x`, `y` or `wt` are dropped, with a warning.

Value

A list with components

`x`	the input values in increasing order with duplicates removed.
`y`	the corresponding y values on the fitted curve.

References

Friedman, J. H. (1984) SMART User's Guide. Laboratory for Computational Statistics, Stanford University Technical Report No. 1.

Friedman, J. H. (1984) A variable span scatterplot smoother. Laboratory for Computational Statistics, Stanford University Technical Report No. 5.

See Also

ppr

Examples

```
require(graphics)

with(cars, {
    plot(speed, dist)
    lines(supsmu(speed, dist))
    lines(supsmu(speed, dist, bass = 7), lty = 2)
    })
```

---

symnum *Symbolic Number Coding*

Description

Symbolically encode a given numeric or logical vector or array. Particularly useful for visualization of structured matrices, e.g., correlation, sparse, or logical ones.

Usage

```
symnum(x, cutpoints = c(0.3, 0.6, 0.8, 0.9, 0.95),
       symbols = if(numeric.x) c(" ", ".", ",", "+", "*", "B")
                 else c(".", "|"),
       legend = length(symbols) >= 3,
       na = "?", eps = 1e-5, numeric.x = is.numeric(x),
       corr = missing(cutpoints) && numeric.x,
       show.max = if(corr) "1", show.min = NULL,
       abbr.colnames = has.colnames,
       lower.triangular = corr && is.numeric(x) && is.matrix(x),
       diag.lower.tri   = corr && !is.null(show.max))
```

Arguments

x	numeric or logical vector or array.
cutpoints	numeric vector whose values cutpoints[j] $= c_j$ (*after* augmentation, see corr below) are used for intervals.
symbols	character vector, one shorter than (the *augmented*, see corr below) cutpoints. symbols[j]$= s_j$ are used as 'code' for the (half open) interval $(c_j, c_{j+1}]$. When numeric.x is FALSE, i.e., by default when argument x is logical, the default is c(".","|") (graphical 0 / 1 s).
legend	logical indicating if a "legend" attribute is desired.
na	character or logical. How NAs are coded. If na == FALSE, NAs are coded invisibly, *including* the "legend" attribute below, which otherwise mentions NA coding.
eps	absolute precision to be used at left and right boundary.

`numeric.x` logical indicating if `x` should be treated as numbers, otherwise as logical.

`corr` logical. If `TRUE`, `x` contains correlations. The cutpoints are augmented by `0` and `1` and `abs(x)` is coded.

`show.max` if `TRUE`, or of mode `character`, the maximal cutpoint is coded especially.

`show.min` if `TRUE`, or of mode `character`, the minimal cutpoint is coded especially.

`abbr.colnames` logical, integer or `NULL` indicating how column names should be abbreviated (if they are); if `NULL` (or `FALSE` and `x` has no column names), the column names will all be empty, i.e., `""`; otherwise if `abbr.colnames` is false, they are left unchanged. If `TRUE` or integer, existing column names will be abbreviated to `abbreviate(*, minlength = abbr.colnames)`.

`lower.triangular` logical. If `TRUE` and `x` is a matrix, only the *lower triangular* part of the matrix is coded as non-blank.

`diag.lower.tri` logical. If `lower.triangular` *and* this are `TRUE`, the *diagonal* part of the matrix is shown.

Value

An atomic character object of class `noquote` and the same dimensions as `x`.

If `legend` is `TRUE` (as by default when there are more than two classes), the result has an attribute `"legend"` containing a legend of the returned character codes, in the form

$$c_1 s_1 c_2 s_2 \ldots s_n c_{n+1}$$

where c_j `= cutpoints[j]` and s_j `= symbols[j]`.

Note

The optional (mostly logical) arguments all try to use smart defaults. Specifying them explicitly may lead to considerably improved output in many cases.

Author(s)

Martin Maechler ⟨maechler@stat.math.ethz.ch⟩

See Also

`as.character`; `image`

Examples

```
ii <- 0:8; names(ii) <- ii
symnum(ii, cut= 2*(0:4), sym = c(".", "-", "+", "$"))
symnum(ii, cut= 2*(0:4), sym = c(".", "-", "+", "$"), show.max=TRUE)

symnum(1:12 %% 3 == 0)# --> "|" = TRUE, "." = FALSE  for logical

## Pascal's Triangle modulo 2 -- odd and even numbers:
N <- 38
pascal <- t(sapply(0:N, function(n) round(choose(n, 0:N - (N-n)%/%2))))
rownames(pascal) <- rep("", 1+N) # <-- to improve "graphic"
```

```
symnum(pascal %% 2, symbols = c(" ", "A"), numeric = FALSE)

##-- Symbolic correlation matrices:
symnum(cor(attitude), diag = FALSE)
symnum(cor(attitude), abbr.= NULL)
symnum(cor(attitude), abbr.= FALSE)
symnum(cor(attitude), abbr.= 2)

symnum(cor(rbind(1, rnorm(25), rnorm(25)^2)))
symnum(cor(matrix(rexp(30, 1), 5, 18))) # <<-- PATTERN ! --
symnum(cm1 <- cor(matrix(rnorm(90) ,  5, 18))) # < White Noise SMALL n
symnum(cm1, diag=FALSE)
symnum(cm2 <- cor(matrix(rnorm(900), 50, 18))) # < White Noise "BIG" n
symnum(cm2, lower=FALSE)

## NA's:
Cm <- cor(matrix(rnorm(60),  10, 6)); Cm[c(3,6), 2] <- NA
symnum(Cm, show.max=NULL)

## Graphical P-values (aka "significance stars"):
pval <- rev(sort(c(outer(1:6, 10^-(1:3)))))
symp <- symnum(pval, corr=FALSE,
               cutpoints = c(0,  .001,.01,.05, .1, 1),
               symbols = c("***","**","*",".","  "))
noquote(cbind(P.val = format(pval), Signif= symp))
```

`t.test` *Student's t-Test*

Description

Performs one and two sample t-tests on vectors of data.

Usage

```
t.test(x, ...)

## Default S3 method:
t.test(x, y = NULL,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)

## S3 method for class 'formula':
t.test(formula, data, subset, na.action, ...)
```

Arguments

`x`	a (non-empty) numeric vector of data values.
`y`	an optional (non-empty) numeric vector of data values.
`alternative`	a character string specifying the alternative hypothesis, must be one of `"two.sided"` (default), `"greater"` or `"less"`. You can specify just the initial letter.

mu	a number indicating the true value of the mean (or difference in means if you are performing a two sample test).
paired	a logical indicating whether you want a paired t-test.
var.equal	a logical variable indicating whether to treat the two variances as being equal. If TRUE then the pooled variance is used to estimate the variance otherwise the Welch (or Satterthwaite) approximation to the degrees of freedom is used.
conf.level	confidence level of the interval.
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs a factor with two levels giving the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

The formula interface is only applicable for the 2-sample tests.

alternative = "greater" is the alternative that x has a larger mean than y.

If paired is TRUE then both x and y must be specified and they must be the same length. Missing values are removed (in pairs if paired is TRUE). If var.equal is TRUE then the pooled estimate of the variance is used. By default, if var.equal is FALSE then the variance is estimated separately for both groups and the Welch modification to the degrees of freedom is used.

If the input data are effectively constant (compared to the larger of the two means) an error is generated.

Value

A list with class "htest" containing the following components:

statistic	the value of the t-statistic.
parameter	the degrees of freedom for the t-statistic.
p.value	the p-value for the test.
conf.int	a confidence interval for the mean appropriate to the specified alternative hypothesis.
estimate	the estimated mean or difference in means depending on whether it was a one-sample test or a two-sample test.
null.value	the specified hypothesized value of the mean or mean difference depending on whether it was a one-sample test or a two-sample test.
alternative	a character string describing the alternative hypothesis.
method	a character string indicating what type of t-test was performed.
data.name	a character string giving the name(s) of the data.

See Also

prop.test

Examples

```
require(graphics)

t.test(1:10,y=c(7:20))      # P = .00001855
t.test(1:10,y=c(7:20, 200)) # P = .1245    -- NOT significant anymore

## Classical example: Student's sleep data
plot(extra ~ group, data = sleep)
## Traditional interface
with(sleep, t.test(extra[group == 1], extra[group == 2]))
## Formula interface
t.test(extra ~ group, data = sleep)
```

TDist *The Student t Distribution*

Description

Density, distribution function, quantile function and random generation for the t distribution with `df` degrees of freedom (and optional non-centrality parameter `ncp`).

Usage

```
dt(x, df, ncp, log = FALSE)
pt(q, df, ncp, lower.tail = TRUE, log.p = FALSE)
qt(p, df, ncp, lower.tail = TRUE, log.p = FALSE)
rt(n, df, ncp)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`df`	degrees of freedom (> 0, maybe non-integer). `df = Inf` is allowed.
`ncp`	non-centrality parameter δ; currently except for `rt()`, only for `abs(ncp) <= 37.62`. If omitted, use the central t distribution.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

The t distribution with `df` $= \nu$ degrees of freedom has density

$$f(x) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\Gamma(\nu/2)}(1 + x^2/\nu)^{-(\nu+1)/2}$$

for all real x. It has mean 0 (for $\nu > 1$) and variance $\frac{\nu}{\nu-2}$ (for $\nu > 2$).

The general *non-central* t with parameters (ν, δ) = `(df, ncp)` is defined as the distribution of $T_\nu(\delta) := (U + \delta)/\sqrt{V/\nu}$ where U and V are independent random variables, $U \sim \mathcal{N}(0, 1)$ and $V \sim \chi^2_\nu$ (see Chisquare).

The most used applications are power calculations for t-tests:
Let $T = \frac{\bar{X}-\mu_0}{S/\sqrt{n}}$ where $\bar{X}$ is the `mean` and S the sample standard deviation (`sd`) of $X_1, X_2, \ldots, X_n$ which are i.i.d. $\mathcal{N}(\mu, \sigma^2)$ Then T is distributed as non-central t with `df` $= n-1$ degrees of freedom and **n**on-**c**entrality **p**arameter `ncp` $= (\mu - \mu_0)\sqrt{n}/\sigma$.

Value

`dt` gives the density, `pt` gives the distribution function, `qt` gives the quantile function, and `rt` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Note

Setting `ncp = 0` is *not* equivalent to omitting `ncp`. R uses the non-centrality functionality whenever `ncp` is specified which provides continuous behavior at $ncp = 0$.

Source

The central `dt` is computed via an accurate formula provided by Catherine Loader (see the reference in `dbinom`).

For the non-central case of `dt`, contributed by Claus Ekstrøm based on the relationship (for $x \neq 0$) to the cumulative distribution.

For the central case of `pt`, a normal approximation in the tails, otherwise via `pbeta`.

For the non-central case of `pt` based on a C translation of

Lenth, R. V. (1989). *Algorithm AS 243* — Cumulative distribution function of the non-central t distribution, *Applied Statistics* **38**, 185–189.

For central `qt`, a C translation of

Hill, G. W. (1970) Algorithm 396: Student's t-quantiles. *Communications of the ACM*, **13(10)**, 619–620.

altered to take account of

Hill, G. W. (1981) Remark on Algorithm 396, *ACM Transactions on Mathematical Software*, **7**, 250–1.

The non-central case is done by inversion.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (Except non-central versions.)

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 2, chapters 28 and 31. Wiley, New York.

See Also

`df` for the F distribution.

Examples

```
require(graphics)

1 - pt(1:5, df = 1)
qt(.975, df = c(1:10,20,50,100,1000))

tt <- seq(0,10, len=21)
ncp <- seq(0,6, len=31)
ptn <- outer(tt,ncp, function(t,d) pt(t, df = 3, ncp=d))
t.tit <- "Non-central t - Probabilities"
image(tt,ncp,ptn, zlim=c(0,1), main = t.tit)
persp(tt,ncp,ptn, zlim=0:1, r=2, phi=20, theta=200, main=t.tit,
      xlab = "t", ylab = "non-centrality parameter",
      zlab = "Pr(T <= t)")

plot(function(x) dt(x, df = 3, ncp = 2), -3, 11, ylim = c(0, 0.32),
     main="Non-central t - Density", yaxs="i")
```

---

termplot *Plot regression terms*

Description

Plots regression terms against their predictors, optionally with standard errors and partial residuals added.

Usage

```
termplot(model, data = NULL, envir = environment(formula(model)),
         partial.resid = FALSE, rug = FALSE,
         terms = NULL, se = FALSE,
         xlabs = NULL, ylabs = NULL, main = NULL,
         col.term = 2, lwd.term = 1.5,
         col.se = "orange", lty.se = 2, lwd.se = 1,
         col.res = "gray", cex = 1, pch = par("pch"),
         col.smth = "darkred", lty.smth = 2, span.smth = 2/3,
         ask = dev.interactive() && nb.fig < n.tms,
         use.factor.levels = TRUE, smooth = NULL, ylim = "common",
         ...)
```

Arguments

`model`	fitted model object
`data`	data frame in which variables in `model` can be found
`envir`	environment in which variables in `model` can be found
`partial.resid`	logical; should partial residuals be plotted?
`rug`	add rugplots (jittered 1-d histograms) to the axes?
`terms`	which terms to plot (default `NULL` means all terms)
`se`	plot pointwise standard errors?

`xlabs`	vector of labels for the x axes
`ylabs`	vector of labels for the y axes
`main`	logical, or vector of main titles; if `TRUE`, the model's call is taken as main title, `NULL` or `FALSE` mean no titles.
`col.term, lwd.term`	color and line width for the 'term curve', see `lines`.
`col.se, lty.se, lwd.se`	color, line type and line width for the 'twice-standard-error curve' when `se = TRUE`.
`col.res, cex, pch`	color, plotting character expansion and type for partial residuals, when `partial.resid = TRUE`, see `points`.
`ask`	logical; if `TRUE`, the user is *ask*ed before each plot, see `par(ask=.)`.
`use.factor.levels`	Should x-axis ticks use factor levels or numbers for factor terms?
`smooth`	`NULL` or a function with the same arguments as `panel.smooth` to draw a smooth through the partial residuals for non-factor terms
`lty.smth, col.smth, span.smth`	Passed to `smooth`
`ylim`	an optional range for the y axis, or `"common"` when a range sufficient for all the plot will be computed, or `"free"` when limits are computed for each plot.
`...`	other graphical parameters.

Details

The model object must have a `predict` method that accepts `type=terms`, eg `glm` in the **base** package, `coxph` and `survreg` in the **survival** package.

For the `partial.resid=TRUE` option it must have a `residuals` method that accepts `type="partial"`, which `lm` and `glm` do.

The `data` argument should rarely be needed, but in some cases `termplot` may be unable to reconstruct the original data frame. Using `na.action=na.exclude` makes these problems less likely.

Nothing sensible happens for interaction terms.

See Also

For (generalized) linear models, `plot.lm` and `predict.glm`.

Examples

```
require(graphics)

had.splines <- "package:splines" %in% search()
if(!had.splines) rs <- require(splines)
x <- 1:100
z <- factor(rep(LETTERS[1:4],25))
y <- rnorm(100, sin(x/10)+as.numeric(z))
model <- glm(y ~ ns(x,6) + z)

par(mfrow=c(2,2)) ## 2 x 2 plots for same model :
termplot(model, main = paste("termplot( ", deparse(model$call)," ...)"))
```

```
termplot(model, rug=TRUE)
termplot(model, partial.resid=TRUE, se = TRUE, main = TRUE)
termplot(model, partial.resid=TRUE, smooth=panel.smooth, span.smth=1/4)
if(!had.splines && rs) detach("package:splines")
```

`terms` *Model Terms*

Description

The function `terms` is a generic function which can be used to extract *terms* objects from various kinds of R data objects.

Usage

```
terms(x, ...)
```

Arguments

`x`	object used to select a method to dispatch.
`...`	further arguments passed to or from other methods.

Details

There are methods for classes `"aovlist"`, and `"terms" "formula"` (see `terms.formula`): the default method just extracts the `terms` component of the object (if any).

There are `print` and `labels` methods for class `"terms"`: the latter prints the term labels (see `terms.object`).

Value

An object of class `c("terms", "formula")` which contains the *terms* representation of a symbolic model. See `terms.object` for its structure.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models.* Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

`terms.object`, `terms.formula`, `lm`, `glm`, `formula`.

`terms.formula` *Construct a terms Object from a Formula*

Description

This function takes a formula and some optional arguments and constructs a terms object. The terms object can then be used to construct a `model.matrix`.

Usage

```
## S3 method for class 'formula':
terms(x, specials = NULL, abb = NULL, data = NULL, neg.out = TRUE,
      keep.order = FALSE, simplify = FALSE, ...,
      allowDotAsName = FALSE)
```

Arguments

`x`	a formula.
`specials`	which functions in the formula should be marked as special in the `terms` object.
`abb`	Not implemented in R.
`data`	a data frame from which the meaning of the special symbol `.` can be inferred. It is unused if there is no `.` in the formula.
`neg.out`	Not implemented in R.
`keep.order`	a logical value indicating whether the terms should keep their positions. If `FALSE` the terms are reordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on. Effects of a given order are kept in the order specified.
`simplify`	should the formula be expanded and simplified, the pre-1.7.0 behaviour?
`...`	further arguments passed to or from other methods.
`allowDotAsName`	normally `.` in a formula refers to the remaining variables contained in `data`. Exceptionally, `.` can be treated as a name for non-standard uses of formulae.

Details

Not all of the options work in the same way that they do in S and not all are implemented.

Value

A `terms.object` object is returned. The object itself is the re-ordered (unless `keep.order = TRUE`) formula. In all cases variables within an interaction term in the formula are re-ordered by the ordering of the `"variables"` attribute, which is the order in which the variables occur in the formula.

See Also

`terms`, `terms.object`

terms.object *Description of Terms Objects*

Description

An object of class terms holds information about a model. Usually the model was specified in terms of a formula and that formula was used to determine the terms object.

Value

The object itself is simply the formula supplied to the call of terms.formula. The object has a number of attributes and they are used to construct the model frame:

factors	A matrix of variables by terms showing which variables appear in which terms. The entries are 0 if the variable does not occur in the term, 1 if it does occur and should be coded by contrasts, and 2 if it occurs and should be coded via dummy variables for all levels (as when an intercept or lower-order term is missing). If there are no terms other than an intercept and offsets, this is numeric(0).
term.labels	A character vector containing the labels for each of the terms in the model, except for offsets. Non-syntactic names will be quoted by backticks. Note that these are after possible re-ordering (unless argument keep.order was false).
variables	A call to list of the variables in the model.
intercept	Either 0, indicating no intercept is to be fit, or 1 indicating that an intercept is to be fit.
order	A vector of the same length as term.labels indicating the order of interaction for each term.
response	The index of the variable (in variables) of the response (the left hand side of the formula). Zero, if there is no response.
offset	If the model contains offset terms there is an offset attribute indicating which variable(s) are offsets
specials	If a specials argument was given to terms.formula there is a specials attribute, a list of vectors (one for each specified special function) giving numeric indices of the arguments of the list returned as the variables attribute which contain these special functions.
dataClasses	optional. A named character vector giving the classes (as given by .MFclass) of the variables used in a fit.

The object has class c("terms", "formula").

Note

These objects are different from those found in S. In particular there is no formula attribute, instead the object is itself a formula. Thus, the mode of a terms object is different as well.

Examples of the specials argument can be seen in the aov and coxph functions.

See Also

terms, formula.

Examples

```
## use of specials (as used for gam() in packages mgcv and gam)
(tf <- terms(y ~ x + x:z + s(x), specials = "s"))
## Note that the "factors" attribute has variables as row names
## and term labels as column names, both as character vectors.
attr(tf, "specials")    # index 's' variable(s)
rownames(attr(tf, "factors"))[attr(tf, "specials")$s]

## we can keep the order by
terms(y ~ x + x:z + s(x), specials = "s", keep.order = TRUE)
```

---

time *Sampling Times of Time Series*

Description

`time` creates the vector of times at which a time series was sampled.

`cycle` gives the positions in the cycle of each observation.

`frequency` returns the number of samples per unit time and `deltat` the time interval between observations (see `ts`).

Usage

```
time(x, ...)
## Default S3 method:
time(x, offset=0, ...)

cycle(x, ...)
frequency(x, ...)
deltat(x, ...)
```

Arguments

`x`	a univariate or multivariate time-series, or a vector or matrix.
`offset`	can be used to indicate when sampling took place in the time unit. `0` (the default) indicates the start of the unit, `0.5` the middle and `1` the end of the interval.
`...`	extra arguments for future methods.

Details

These are all generic functions, which will use the `tsp` attribute of `x` if it exists. `time` and `cycle` have methods for class `ts` that coerce the result to that class.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`ts`, `start`, `tsp`, `window`.

`date` for clock time, `system.time` for CPU usage.

Examples

```
require(graphics)

cycle(presidents)
# a simple series plot
plot(as.vector(time(presidents)), as.vector(presidents), type="l")
```

`toeplitz` *Form Symmetric Toeplitz Matrix*

Description

Forms a symmetric Toeplitz matrix given its first row.

Usage

```
toeplitz(x)
```

Arguments

`x`	the first row to form the Toeplitz matrix.

Value

The Toeplitz matrix.

Author(s)

A. Trapletti

Examples

```
x <- 1:5
toeplitz (x)
```

ts *Time-Series Objects*

Description

The function ts is used to create time-series objects.

as.ts and is.ts coerce an object to a time-series and test whether an object is a time series.

Usage

```
ts(data = NA, start = 1, end = numeric(0), frequency = 1,
   deltat = 1, ts.eps = getOption("ts.eps"), class = , names = )
as.ts(x, ...)
is.ts(x)
```

Arguments

data	a numeric vector or matrix of the observed time-series values. A data frame will be coerced to a numeric matrix via data.matrix.
start	the time of the first observation. Either a single number or a vector of two integers, which specify a natural time unit and a (1-based) number of samples into the time unit. See the examples for the use of the second form.
end	the time of the last observation, specified in the same way as start.
frequency	the number of observations per unit of time.
deltat	the fraction of the sampling period between successive observations; e.g., 1/12 for monthly data. Only one of frequency or deltat should be provided.
ts.eps	time series comparison tolerance. Frequencies are considered equal if their absolute difference is less than ts.eps.
class	class to be given to the result, or none if NULL or "none". The default is "ts" for a single series, c("mts", "ts") for multiple series.
names	a character vector of names for the series in a multiple series: defaults to the colnames of data, or Series 1, Series 2, ...
x	an arbitrary R object.
...	arguments passed to methods (unused for the default method).

Details

The function ts is used to create time-series objects. These are vector or matrices with class of "ts" (and additional attributes) which represent data which has been sampled at equispaced points in time. In the matrix case, each column of the matrix data is assumed to contain a single (univariate) time series. Time series must have at least one observation, and although they need not be numeric there is very limited support for non-numeric series.

Class "ts" has a number of methods. In particular arithmetic will attempt to align time axes, and subsetting to extract subsets of series can be used (e.g., EuStockMarkets[, "DAX"]). However, subsetting the first (or only) dimension will return a matrix or vector, as will matrix subsetting. Subassignment can be used to replace values but not to extend a series (see window). There is a method for t that transposes the series as a matrix (a one-column matrix if a vector) and hence returns a result that does not inherit from class "ts".

The value of argument `frequency` is used when the series is sampled an integral number of times in each unit time interval. For example, one could use a value of `7` for `frequency` when the data are sampled daily, and the natural time period is a week, or `12` when the data are sampled monthly and the natural time period is a year. Values of `4` and `12` are assumed in (e.g.) `print` methods to imply a quarterly and monthly series respectively.

`as.ts` is generic. Its default method will use the `tsp` attribute of the object if it has one to set the start and end times and frequency.

`is.ts` tests if an object is a time series. It is generic: you can write methods to handle specific classes of objects, see InternalMethods.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`tsp`, `frequency`, `start`, `end`, `time`, `window`; `print.ts`, the print method for time series objects; `plot.ts`, the plot method for time series objects.

Examples

```
require(graphics)

ts(1:10, frequency = 4, start = c(1959, 2)) # 2nd Quarter of 1959
print( ts(1:10, frequency = 7, start = c(12, 2)), calendar = TRUE)
# print.ts(.)
## Using July 1954 as start date:
gnp <- ts(cumsum(1 + round(rnorm(100), 2)),
          start = c(1954, 7), frequency = 12)
plot(gnp) # using 'plot.ts' for time-series plot

## Multivariate
z <- ts(matrix(rnorm(300), 100, 3), start=c(1961, 1), frequency=12)
class(z)
plot(z)
plot(z, plot.type="single", lty=1:3)

## A phase plot:
plot(nhtemp, c(nhtemp[-1], NA), cex = .8, col="blue",
     main = "Lag plot of New Haven temperatures")
## a clearer way to do this would be
## Not run:
plot(nhtemp, lag(nhtemp, 1), cex = .8, col="blue",
     main = "Lag plot of New Haven temperatures")
## End(Not run)
```

---

ts-methods *Methods for Time Series Objects*

Description

Methods for objects of class `"ts"`, typically the result of `ts`.

Usage

```
## S3 method for class 'ts':
diff(x, lag = 1, differences = 1, ...)

## S3 method for class 'ts':
na.omit(object, ...)
```

Arguments

x	an object of class "ts" containing the values to be differenced.
lag	an integer indicating which lag to use.
differences	an integer indicating the order of the difference.
object	a univariate or multivariate time series.
...	further arguments to be passed to or from methods.

Details

The na.omit method omits initial and final segments with missing values in one or more of the series. 'Internal' missing values will lead to failure.

Value

For the na.omit method, a time series without missing values. The class of object will be preserved.

See Also

diff; na.omit, na.fail, na.contiguous.

---

ts.plot *Plot Multiple Time Series*

Description

Plot several time series on a common plot. Unlike plot.ts the series can have a different time bases, but they should have the same frequency.

Usage

```
ts.plot(..., gpars = list())
```

Arguments

...	one or more univariate or multivariate time series.
gpars	list of named graphics parameters to be passed to the plotting functions. Those commonly used can be supplied directly in ....

Value

None.

Note

Although this can be used for a single time series, `plot` is easier to use and is preferred.

See Also

`plot.ts`

Examples

```
require(graphics)

ts.plot(ldeaths, mdeaths, fdeaths,
        gpars=list(xlab="year", ylab="deaths", lty=c(1:3)))
```

---

ts.union *Bind Two or More Time Series*

Description

Bind time series which have a common frequency. `ts.union` pads with `NA`s to the total time coverage, `ts.intersect` restricts to the time covered by all the series.

Usage

```
ts.intersect(..., dframe = FALSE)
ts.union(..., dframe = FALSE)
```

Arguments

`...`	two or more univariate or multivariate time series, or objects which can coerced to time series.
`dframe`	logical; if `TRUE` return the result as a data frame.

Details

As a special case, `...` can contain vectors or matrices of the same length as the combined time series of the time series present, as well as those of a single row.

Value

A time series object if `dframe` is `FALSE`, otherwise a data frame.

See Also

`cbind`.

Examples

```
ts.union(mdeaths, fdeaths)
cbind(mdeaths, fdeaths) # same as the previous line
ts.intersect(window(mdeaths, 1976), window(fdeaths, 1974, 1978))

sales1 <- ts.union(BJsales, lead = BJsales.lead)
ts.intersect(sales1, lead3 = lag(BJsales.lead, -3))
```

`tsdiag` *Diagnostic Plots for Time-Series Fits*

Description

A generic function to plot time-series diagnostics.

Usage

```
tsdiag(object, gof.lag, ...)
```

Arguments

`object`	a fitted time-series model
`gof.lag`	the maximum number of lags for a Portmanteau goodness-of-fit test
`...`	further arguments to be passed to particular methods

Details

This is a generic function. It will generally plot the residuals, often standardized, the autocorrelation function of the residuals, and the p-values of a Portmanteau test for all lags up to `gof.lag`.

The methods for `arima` and `StructTS` objects plots residuals scaled by the estimate of their (individual) variance, and use the Ljung–Box version of the portmanteau test.

Value

None. Diagnostics are plotted.

See Also

`arima`, `StructTS`, `Box.test`

Examples

```
## Not run: require(graphics)

fit <- arima(lh, c(1,0,0))
tsdiag(fit)

## see also examples(arima)

(fit <- StructTS(log10(JohnsonJohnson), type="BSM"))
tsdiag(fit)
## End(Not run)
```

tsp *Tsp Attribute of Time-Series-like Objects*

Description

`tsp` returns the `tsp` attribute (or `NULL`). It is included for compatibility with S version 2. `tsp<-` sets the `tsp` attribute. `hasTsp` ensures `x` has a `tsp` attribute, by adding one if needed.

Usage

```
tsp(x)
tsp(x) <- value
hasTsp(x)
```

Arguments

`x`	a vector or matrix or univariate or multivariate time-series.
`value`	a numeric vector of length 3 or `NULL`.

Details

The `tsp` attribute was previously described here as `c(start(x), end(x), frequency(x))`, but this is incorrect. It gives the start time *in time units*, the end time and the frequency.

Assignments are checked for consistency.

Assigning `NULL` which removes the `tsp` attribute *and* any `"ts"` (or `"mts"`) class of `x`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`ts`, `time`, `start`.

tsSmooth *Use Fixed-Interval Smoothing on Time Series*

Description

Performs fixed-interval smoothing on a univariate time series via a state-space model. Fixed-interval smoothing gives the best estimate of the state at each time point based on the whole observed series.

Usage

```
tsSmooth(object, ...)
```

Arguments

`object`	a time-series fit. Currently only class "`StructTS`" is supported
`...`	possible arguments for future methods.

Value

A time series, with as many dimensions as the state space and results at each time point of the original series. (For seasonal models, only the current seasonal component is returned.)

Author(s)

B. D. Ripley

References

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods.* Oxford University Press.

See Also

`KalmanSmooth`, `StructTS`.

For examples consult `AirPassengers`, `JohnsonJohnson` and `Nile`.

`Tukey` *The Studentized Range Distribution*

Description

Functions of the distribution of the studentized range, R/s, where R is the range of a standard normal sample and $df \times s^2$ is independently distributed as chi-squared with df degrees of freedom, see `pchisq`.

Usage

```
ptukey(q, nmeans, df, nranges = 1, lower.tail = TRUE, log.p = FALSE)
qtukey(p, nmeans, df, nranges = 1, lower.tail = TRUE, log.p = FALSE)
```

Arguments

`q`	vector of quantiles.
`p`	vector of probabilities.
`nmeans`	sample size for range (same for each group).
`df`	degrees of freedom for s (see below).
`nranges`	number of *groups* whose **maximum** range is considered.
`log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \le x]$, otherwise, $P[X > x]$.

Details

If n_g =nranges is greater than one, R is the *maximum* of n_g groups of nmeans observations each.

Value

ptukey gives the distribution function and qtukey its inverse, the quantile function.

Note

A Legendre 16-point formula is used for the integral of ptukey. The computations are relatively expensive, especially for qtukey which uses a simple secant method for finding the inverse of ptukey. qtukey will be accurate to the 4th decimal place.

References

Copenhaver, Margaret Diponzio and Holland, Burt S. (1988) Multiple comparisons of simple effects in the two-way analysis of variance with fixed effects. *Journal of Statistical Computation and Simulation*, **30**, 1–15.

See Also

pnorm and qnorm for the corresponding functions for the normal distribution.

Examples

```
if(interactive())
  curve(ptukey(x, nm=6, df=5), from=-1, to=8, n=101)
(ptt <- ptukey(0:10, 2, df= 5))
(qtt <- qtukey(.95, 2, df= 2:11))
## The precision may be not much more than about 8 digits:
summary(abs(.95 - ptukey(qtt,2, df = 2:11)))
```

TukeyHSD *Compute Tukey Honest Significant Differences*

Description

Create a set of confidence intervals on the differences between the means of the levels of a factor with the specified family-wise probability of coverage. The intervals are based on the Studentized range statistic, Tukey's 'Honest Significant Difference' method. There is a plot method.

Usage

```
TukeyHSD(x, which, ordered = FALSE, conf.level = 0.95, ...)
```

Arguments

x	A fitted model object, usually an aov fit.
which	A character vector listing terms in the fitted model for which the intervals should be calculated. Defaults to all the terms.
ordered	A logical value indicating if the levels of the factor should be ordered according to increasing average in the sample before taking differences. If ordered is true then the calculated differences in the means will all be positive. The significant differences will be those for which the lwr end point is positive.
conf.level	A numeric value between zero and one giving the family-wise confidence level to use.
...	Optional additional arguments. None are used at present.

Details

When comparing the means for the levels of a factor in an analysis of variance, a simple comparison using t-tests will inflate the probability of declaring a significant difference when it is not in fact present. This because the intervals are calculated with a given coverage probability for each interval but the interpretation of the coverage is usually with respect to the entire family of intervals.

John Tukey introduced intervals based on the range of the sample means rather than the individual differences. The intervals returned by this function are based on this Studentized range statistics.

Technically the intervals constructed in this way would only apply to balanced designs where there are the same number of observations made at each level of the factor. This function incorporates an adjustment for sample size that produces sensible intervals for mildly unbalanced designs.

If which specifies non-factor terms these will be dropped with a warning: if no terms are left this is a an error.

Value

A list with one component for each term requested in which. Each component is a matrix with columns diff giving the difference in the observed means, lwr giving the lower end point of the interval, upr giving the upper end point and p adj giving the p-value after adjustment for the multiple comparisons.

Author(s)

Douglas Bates

References

Miller, R. G. (1981) *Simultaneous Statistical Inference*. Springer.

Yandell, B. S. (1997) *Practical Data Analysis for Designed Experiments*. Chapman & Hall.

See Also

aov, qtukey, model.tables, simint

Examples

```
require(graphics)

summary(fm1 <- aov(breaks ~ wool + tension, data = warpbreaks))
TukeyHSD(fm1, "tension", ordered = TRUE)
plot(TukeyHSD(fm1, "tension"))
```

Uniform *The Uniform Distribution*

Description

These functions provide information about the uniform distribution on the interval from `min` to `max`. `dunif` gives the density, `punif` gives the distribution function `qunif` gives the quantile function and `runif` generates random deviates.

Usage

```
dunif(x, min=0, max=1, log = FALSE)
punif(q, min=0, max=1, lower.tail = TRUE, log.p = FALSE)
qunif(p, min=0, max=1, lower.tail = TRUE, log.p = FALSE)
runif(n, min=0, max=1)
```

Arguments

`x,q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`min,max`	lower and upper limits of the distribution. Must be finite.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

If `min` or `max` are not specified they assume the default values of `0` and `1` respectively.

The uniform distribution has density

$$f(x) = \frac{1}{max - min}$$

for $min \leq x \leq max$.

For the case of $u := min == max$, the limit case of $X \equiv u$ is assumed, although there is no density in that case and `dunif` will return `NaN` (the error condition).

`runif` will not generate either of the extreme values unless `max = min` or `max-min` is small compared to `min`, and in particular not for the default arguments.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`.Random.seed` about random number generation, `rnorm`, etc for other distributions.

Examples

```
u <- runif(20)

## The following relations always hold :
punif(u) == u
dunif(u) == 1

var(runif(10000))#- ~ = 1/12 = .08333
```

`uniroot` *One Dimensional Root (Zero) Finding*

Description

The function `uniroot` searches the interval from `lower` to `upper` for a root (i.e., zero) of the function `f` with respect to its first argument.

Usage

```
uniroot(f, interval, ...,
        lower = min(interval), upper = max(interval),
        f.lower = f(lower, ...), f.upper = f(upper, ...),
        tol = .Machine$double.eps^0.25, maxiter = 1000)
```

Arguments

`f`	the function for which the root is sought.
`interval`	a vector containing the end-points of the interval to be searched for the root.
`...`	additional named or unnamed arguments to be passed to `f`
`lower, upper`	the lower and upper end points of the interval to be searched.
`f.lower, f.upper`	the same as `f(upper)` and `f(lower)`, respectively. Passing these values from the caller where they are often known is more economical as soon as `f()` contains non-trivial computations.
`tol`	the desired accuracy (convergence tolerance).
`maxiter`	the maximum number of iterations.

Details

Note that arguments after `...` must be matched exactly.

Either `interval` or both `lower` and `upper` must be specified: the upper endpoint must be strictly larger than the lower endpoint. The function values at the endpoints must be of opposite signs (or zero).

The function uses Fortran subroutine '"zeroin"' (from Netlib) based on algorithms given in the reference below. They assume a continuous function (which then is known to have at least one root in the interval).

Convergence is declared either if `f(x) == 0` or the change in `x` for one step of the algorithm is less than `tol` (plus an allowance for representation error in `x`).

If the algorithm does not converge in `maxiter` steps, a warning is printed and the current approximation is returned.

`f` will be called as `f(x, ...)` for a numeric value of *x*.

Value

A list with four components: `root` and `f.root` give the location of the root and the value of the function evaluated at that point. `iter` and `estim.prec` give the number of iterations used and an approximate estimated precision for `root`. (If the root occurs at one of the endpoints, the estimated precision is `NA`.)

Source

Based on 'zeroin.c' in http://www.netlib.org/c/brent.shar.

References

Brent, R. (1973) *Algorithms for Minimization without Derivatives.* Englewood Cliffs, NJ: Prentice-Hall.

See Also

`polyroot` for all complex roots of a polynomial; `optimize`, `nlm`.

Examples

```
require(utils) # for str

f <- function (x,a) x - a
str(xmin <- uniroot(f, c(0, 1), tol = 0.0001, a = 1/3))
str(uniroot(function(x) x*(x^2-1) + .5, lower = -2, upper = 2,
            tol = 0.0001), dig = 10)
str(uniroot(function(x) x*(x^2-1) + .5, lower = -2, upper = 2,
            tol = 1e-10 ), dig = 10)

## Find the smallest value x for which exp(x) > 0 (numerically):
r <- uniroot(function(x) 1e80*exp(x)-1e-300, c(-1000,0), tol = 1e-15)
str(r, digits= 15) ##> around -745, depending on the platform.

exp(r$root)     # = 0, but not for r$root * 0.999...
minexp <- r$root * (1 - 10*.Machine$double.eps)
exp(minexp)     # typically denormalized
```

---

update *Update and Re-fit a Model Call*

Description

`update` will update and (by default) re-fit a model. It does this by extracting the call stored in the object, updating the call and (by default) evaluating that call. Sometimes it is useful to call `update` with only one argument, for example if the data frame has been corrected.

Usage

```
update(object, ...)

## Default S3 method:
update(object, formula., ..., evaluate = TRUE)
```

Arguments

object	An existing fit from a model function such as lm, glm and many others.
formula.	Changes to the formula – see update.formula for details.
...	Additional arguments to the call, or arguments with changed values. Use name=NULL to remove the argument name.
evaluate	If true evaluate the new call else return the call.

Value

If evaluate = TRUE the fitted object, otherwise the updated call.

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

update.formula

Examples

```
oldcon <- options(contrasts = c("contr.treatment", "contr.poly"))
## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2, 10, 20, labels = c("Ctl", "Trt"))
weight <- c(ctl, trt)
lm.D9 <- lm(weight ~ group)
lm.D9
summary(lm.D90 <- update(lm.D9, . ~ . - 1))
options(contrasts = c("contr.helmert", "contr.poly"))
update(lm.D9)
options(oldcon)
```

---

update.formula *Model Updating*

Description

update.formula is used to update model formulae. This typically involves adding or dropping terms, but updates can be more general.

Usage

```
## S3 method for class 'formula':
update(old, new, ...)
```

Arguments

old	a model formula to be updated.
new	a formula giving a template which specifies how to update.
...	further arguments passed to or from other methods.

Details

Either or both of old and new can be objects such as length-one character vectors which can be coerced to a formula via as.formula.

The function works by first identifying the *left-hand side* and *right-hand side* of the old formula. It then examines the new formula and substitutes the *lhs* of the old formula for any occurrence of '.' on the left of new, and substitutes the *rhs* of the old formula for any occurrence of '.' on the right of new. The result is then simplified *via* terms.formula(simplify = TRUE).

Value

The updated formula is returned. The environment of the result will that of old.

See Also

terms, model.matrix.

Examples

```
update(y ~ x,    ~ . + x2) #> y ~ x + x2
update(y ~ x, log(.) ~ . ) #> log(y) ~ x
```

---

var.test *F Test to Compare Two Variances*

Description

Performs an F test to compare the variances of two samples from normal populations.

Usage

```
var.test(x, ...)

## Default S3 method:
var.test(x, y, ratio = 1,
         alternative = c("two.sided", "less", "greater"),
         conf.level = 0.95, ...)

## S3 method for class 'formula':
var.test(formula, data, subset, na.action, ...)
```

Arguments

x, y	numeric vectors of data values, or fitted linear model objects (inheriting from class "lm").
ratio	the hypothesized ratio of the population variances of x and y.
alternative	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.
conf.level	confidence level for the returned confidence interval.
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs a factor with two levels giving the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

The null hypothesis is that the ratio of the variances of the populations from which x and y were drawn, or in the data to which the linear models x and y were fitted, is equal to ratio.

Value

A list with class "htest" containing the following components:

statistic	the value of the F test statistic.
parameter	the degrees of the freedom of the F distribution of the test statistic.
p.value	the p-value of the test.
conf.int	a confidence interval for the ratio of the population variances.
estimate	the ratio of the sample variances of x and y.
null.value	the ratio of population variances under the null.
alternative	a character string describing the alternative hypothesis.
method	the character string "F test to compare two variances".
data.name	a character string giving the names of the data.

See Also

bartlett.test for testing homogeneity of variances in more than two samples from normal distributions; ansari.test and mood.test for two rank based (nonparametric) two-sample tests for difference in scale.

Examples

```
x <- rnorm(50, mean = 0, sd = 2)
y <- rnorm(30, mean = 1, sd = 1)
var.test(x, y)                   # Do x and y have the same variance?
var.test(lm(x ~ 1), lm(y ~ 1))   # The same.
```

`varimax` *Rotation Methods for Factor Analysis*

Description

These functions 'rotate' loading matrices in factor analysis.

Usage

```
varimax(x, normalize = TRUE, eps = 1e-5)
promax(x, m = 4)
```

Arguments

`x`	A loadings matrix, with p rows and $k < p$ columns
`m`	The power used the target for `promax`. Values of 2 to 4 are recommended.
`normalize`	logical. Should Kaiser normalization be performed? If so the rows of `x` are re-scaled to unit length before rotation, and scaled back afterwards.
`eps`	The tolerance for stopping: the relative change in the sum of singular values.

Details

These seek a 'rotation' of the factors `x %*% T` that aims to clarify the structure of the loadings matrix. The matrix `T` is a rotation (possibly with reflection) for `varimax`, but a general linear transformation for `promax`, with the variance of the factors being preserved.

Value

A list with components

`loadings`	The 'rotated' loadings matrix, `x %*% rotmat`, of class `"loadings"`.
`rotmat`	The 'rotation' matrix.

References

Hendrickson, A. E. and White, P. O. (1964) Promax: a quick method for rotation to orthogonal oblique structure. *British Journal of Statistical Psychology*, **17**, 65–70.

Horst, P. (1965) *Factor Analysis of Data Matrices.* Holt, Rinehart and Winston. Chapter 10.

Kaiser, H. F. (1958) The varimax criterion for analytic rotation in factor analysis. *Psychometrika* **23**, 187–200.

Lawley, D. N. and Maxwell, A. E. (1971) *Factor Analysis as a Statistical Method*. Second edition. Butterworths.

See Also

`factanal`, `Harman74.cor`.

Examples

```
## varimax with normalize = TRUE is the default
fa <- factanal( ~., 2, data = swiss)
varimax(loadings(fa), normalize = FALSE)
promax(loadings(fa))
```

---

vcov *Calculate Variance-Covariance Matrix for a Fitted Model Object*

Description

Returns the variance-covariance matrix of the main parameters of a fitted model object.

Usage

```
vcov(object, ...)
```

Arguments

`object`	a fitted model object.
`...`	additional arguments for method functions. For the `glm` method this can be used to pass a `dispersion` parameter.

Details

This is a generic function. Functions with names beginning in `vcov.` will be methods for this function. Classes with methods for this function include: `lm`, `mlm`, `glm`, `nls`, `negbin`, `polr`, `rlm` (in package **MASS**), `multinom` (in package `nnet`) `gls`, `lme` (in package **nlme**, `coxph` and `survreg` (in package **survival**).

Value

A matrix of the estimated covariances between the parameter estimates in the linear or non-linear predictor of the model.

Weibull *The Weibull Distribution*

Description

Density, distribution function, quantile function and random generation for the Weibull distribution with parameters `shape` and `scale`.

Usage

```
dweibull(x, shape, scale = 1, log = FALSE)
pweibull(q, shape, scale = 1, lower.tail = TRUE, log.p = FALSE)
qweibull(p, shape, scale = 1, lower.tail = TRUE, log.p = FALSE)
rweibull(n, shape, scale = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`shape, scale`	shape and scale parameters, the latter defaulting to 1.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

The Weibull distribution with `shape` parameter a and `scale` parameter σ has density given by

$$f(x) = (a/\sigma)(x/\sigma)^{a-1} \exp(-(x/\sigma)^a)$$

for $x \geq 0$. The cumulative distribution function is $F(x) = 1 - \exp(-(x/\sigma)^a)$ on $x \geq 0$, the mean is $E(X) = \sigma\Gamma(1 + 1/a)$, and the $Var(X) = \sigma^2(\Gamma(1 + 2/a) - (\Gamma(1 + 1/a))^2)$.

Value

`dweibull` gives the density, `pweibull` gives the distribution function, `qweibull` gives the quantile function, and `rweibull` generates random deviates.

Invalid arguments will result in return value `NaN`, with a warning.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-pweibull(t, a, b, lower = FALSE, log = TRUE)` which is just $H(t) = (t/b)^a$.

Source

`[dpq]weibull` are calculated directly from the definitions. `rweibull` uses inversion.

References

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 21. Wiley, New York.

See Also

The Exponential is a special case of the Weibull distribution.

Examples

```
x <- c(0,rlnorm(50))
all.equal(dweibull(x, shape = 1), dexp(x))
all.equal(pweibull(x, shape = 1, scale = pi), pexp(x, rate = 1/pi))
## Cumulative hazard H():
all.equal(pweibull(x, 2.5, pi, lower.tail=FALSE, log.p=TRUE), -(x/pi)^2.5,
          tol = 1e-15)
all.equal(qweibull(x/11, shape = 1, scale = pi), qexp(x/11, rate = 1/pi))
```

weighted.mean *Weighted Arithmetic Mean*

Description

Compute a weighted mean of a numeric vector.

Usage

```
weighted.mean(x, w, na.rm = FALSE)
```

Arguments

x	a numeric vector containing the values whose mean is to be computed.
w	a vector of weights the same length as x giving the weights to use for each element of x.
na.rm	a logical value indicating whether NA values in x should be stripped before the computation proceeds.

Details

If w is missing then all elements of x are given the same weight.

Missing values in w are not handled.

See Also

mean

Examples

```
## GPA from Siegel 1994
wt <- c(5,  5,  4,  1)/15
x <- c(3.7,3.3,3.5,2.8)
xm <- weighted.mean(x,wt)
```

weighted.residuals *Compute Weighted Residuals*

Description

Computed weighted residuals from a linear model fit.

Usage

```
weighted.residuals(obj, drop0 = TRUE)
```

Arguments

obj	R object, typically of class lm or glm.
drop0	logical. If TRUE, drop all cases with weights == 0.

Details

Weighted residuals are based on the deviance residuals, which for a `lm` fit are the raw residuals R_i multiplied by $\sqrt{w_i}$, where w_i are the `weights` as specified in `lm`'s call.

Dropping cases with weights zero is compatible with `influence` and related functions.

Value

Numeric vector of length n', where n' is the number of of non-0 weights (`drop0 = TRUE`) or the number of observations, otherwise.

See Also

`residuals`, `lm.influence`, etc.

Examples

```
utils::example("lm")
all.equal(weighted.residuals(lm.D9),
          residuals(lm.D9))
x <- 1:10
w <- 0:9
y <- rnorm(x)
weighted.residuals(lmxy <- lm(y ~ x, weights = w))
weighted.residuals(lmxy, drop0 = FALSE)
```

`wilcox.test` *Wilcoxon Rank Sum and Signed Rank Tests*

Description

Performs one and two sample Wilcoxon tests on vectors of data; the latter is also known as 'Mann-Whitney' test.

Usage

```
wilcox.test(x, ...)

## Default S3 method:
wilcox.test(x, y = NULL,
            alternative = c("two.sided", "less", "greater"),
            mu = 0, paired = FALSE, exact = NULL, correct = TRUE,
            conf.int = FALSE, conf.level = 0.95, ...)

## S3 method for class 'formula':
wilcox.test(formula, data, subset, na.action, ...)
```

Arguments

x	numeric vector of data values. Non-finite (e.g. infinite or missing) values will be omitted.
y	an optional numeric vector of data values.
alternative	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.
mu	a number specifying an optional parameter used to form the null hypothesis. See 'Details'.
paired	a logical indicating whether you want a paired test.
exact	a logical indicating whether an exact p-value should be computed.
correct	a logical indicating whether to apply continuity correction in the normal approximation for the p-value.
conf.int	a logical indicating whether a confidence interval should be computed.
conf.level	confidence level of the interval.
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs a factor with two levels giving the corresponding groups.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to getOption("na.action").
...	further arguments to be passed to or from methods.

Details

The formula interface is only applicable for the 2-sample tests.

If only x is given, or if both x and y are given and paired is TRUE, a Wilcoxon signed rank test of the null that the distribution of x (in the one sample case) or of x − y (in the paired two sample case) is symmetric about mu is performed.

Otherwise, if both x and y are given and paired is FALSE, a Wilcoxon rank sum test (equivalent to the Mann-Whitney test: see the Note) is carried out. In this case, the null hypothesis is that the distributions of x and y differ by a location shift of mu and the alternative is that they differ by some other location shift (and the one-sided alternative "greater" is that x is shifted to the right of y).

By default (if exact is not specified), an exact p-value is computed if the samples contain less than 50 finite values and there are no ties. Otherwise, a normal approximation is used.

Optionally (if argument conf.int is true), a nonparametric confidence interval and an estimator for the pseudomedian (one-sample case) or for the difference of the location parameters x−y is computed. (The pseudomedian of a distribution F is the median of the distribution of $(u + v)/2$, where u and v are independent, each with distribution F. If F is symmetric, then the pseudomedian and median coincide. See Hollander & Wolfe (1973), page 34.) If exact p-values are available, an exact confidence interval is obtained by the algorithm described in Bauer (1972), and the Hodges-Lehmann estimator is employed. Otherwise, the returned confidence interval and point estimate are based on normal approximations.

With small samples it may not be possible to achieve very high confidence interval coverages. If this happens a warning will be given and an interval with lower coverage will be substituted.

Value

A list with class `"htest"` containing the following components:

`statistic`	the value of the test statistic with a name describing it.
`parameter`	the parameter(s) for the exact distribution of the test statistic.
`p.value`	the p-value for the test.
`null.value`	the location parameter `mu`.
`alternative`	a character string describing the alternative hypothesis.
`method`	the type of test applied.
`data.name`	a character string giving the names of the data.
`conf.int`	a confidence interval for the location parameter. (Only present if argument `conf.int = TRUE`.)
`estimate`	an estimate of the location parameter. (Only present if argument `conf.int = TRUE`.)

Warning

This function can use large amounts of memory and stack (and even crash R if the stack limit is exceeded) if `exact = TRUE` and one sample is large (several thousands or more).

Note

The literature is not unanimous about the definitions of the Wilcoxon rank sum and Mann-Whitney tests. The two most common definitions correspond to the sum of the ranks of the first sample with the minimum value subtracted or not: R subtracts and S-PLUS does not, giving a value which is larger by $m(m+1)/2$ for a first sample of size m. (It seems Wilcoxon's original paper used the unadjusted sum of the ranks but subsequent tables subtracted the minimum.)

R's value can also be computed as the number of all pairs `(x[i], y[j])` for which `y[j]` is not greater than `x[i]`, the most common definition of the Mann-Whitney test.

References

David F. Bauer (1972), Constructing confidence sets using rank statistics. *Journal of the American Statistical Association* **67**, 687–690.

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric Statistical Methods.* New York: John Wiley & Sons. Pages 27–33 (one-sample), 68–75 (two-sample).
Or second edition (1999).

See Also

`psignrank`, `pwilcox`.

`wilcox.exact` in **exactRankTests** covers much of the same ground, but also produces exact p-values in the presence of ties.

`wilcox_test` in package **coin** for exact and approximate *conditional* p-values for the Wilcoxon tests.

`kruskal.test` for testing homogeneity in location parameters in the case of two or more samples; `t.test` for an alternative under normality assumptions [or large samples]

Examples

```
require(graphics)
## One-sample test.
## Hollander & Wolfe (1973), 29f.
## Hamilton depression scale factor measurements in 9 patients with
##  mixed anxiety and depression, taken at the first (x) and second
##  (y) visit after initiation of a therapy (administration of a
##  tranquilizer).
x <- c(1.83,  0.50,  1.62,  2.48, 1.68, 1.88, 1.55, 3.06, 1.30)
y <- c(0.878, 0.647, 0.598, 2.05, 1.06, 1.29, 1.06, 3.14, 1.29)
wilcox.test(x, y, paired = TRUE, alternative = "greater")
wilcox.test(y - x, alternative = "less")    # The same.
wilcox.test(y - x, alternative = "less",
            exact = FALSE, correct = FALSE) # H&W large sample
                                            # approximation

## Two-sample test.
## Hollander & Wolfe (1973), 69f.
## Permeability constants of the human chorioamnion (a placental
##  membrane) at term (x) and between 12 to 26 weeks gestational
##  age (y).  The alternative of interest is greater permeability
##  of the human chorioamnion for the term pregnancy.
x <- c(0.80, 0.83, 1.89, 1.04, 1.45, 1.38, 1.91, 1.64, 0.73, 1.46)
y <- c(1.15, 0.88, 0.90, 0.74, 1.21)
wilcox.test(x, y, alternative = "g")        # greater
wilcox.test(x, y, alternative = "greater",
            exact = FALSE, correct = FALSE) # H&W large sample
                                            # approximation

wilcox.test(rnorm(10), rnorm(10, 2), conf.int = TRUE)

## Formula interface.
boxplot(Ozone ~ Month, data = airquality)
wilcox.test(Ozone ~ Month, data = airquality,
            subset = Month %in% c(5, 8))
```

Wilcoxon *Distribution of the Wilcoxon Rank Sum Statistic*

Description

Density, distribution function, quantile function and random generation for the distribution of the Wilcoxon rank sum statistic obtained from samples with size `m` and `n`, respectively.

Usage

```
dwilcox(x, m, n, log = FALSE)
pwilcox(q, m, n, lower.tail = TRUE, log.p = FALSE)
qwilcox(p, m, n, lower.tail = TRUE, log.p = FALSE)
rwilcox(nn, m, n)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`nn`	number of observations. If `length(nn) > 1`, the length is taken to be the number required.
`m, n`	numbers of observations in the first and second sample, respectively. Can be vectors of positive integers.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

Details

This distribution is obtained as follows. Let `x` and `y` be two random, independent samples of size `m` and `n`. Then the Wilcoxon rank sum statistic is the number of all pairs `(x[i], y[j])` for which `y[j]` is not greater than `x[i]`. This statistic takes values between `0` and `m * n`, and its mean and variance are `m * n / 2` and `m * n * (m + n + 1) / 12`, respectively.

If any of the first three arguments are vectors, the recycling rule is used to do the calculations for all combinations of the three up to the length of the longest vector.

Value

`dwilcox` gives the density, `pwilcox` gives the distribution function, `qwilcox` gives the quantile function, and `rwilcox` generates random deviates.

Warning

These functions can use large amounts of memory and stack (and even crash R if the stack limit is exceeded and stack-checking is not in place) if one sample is large (several thousands or more).

Note

S-PLUS uses a different (but equivalent) definition of the Wilcoxon statistic: see `wilcox.test` for details.

Author(s)

Kurt Hornik

Source

These are calculated via recursion, based on `cwilcox(k, m, n)`, the number of choices with statistic `k` from samples of size `m` and `n`, which is itself calculated recursively and the results cached. Then `dwilcox` and `pwilcox` sum appropriate values of `cwilcox`, and `qwilcox` is based on inversion.

`rwilcox` generates a random permutation of ranks and evaluates the statistic.

See Also

`wilcox.test` to calculate the statistic from data, find p values and so on.

`dsignrank` etc, for the distribution of the *one-sample* Wilcoxon signed rank statistic.

Examples

```
require(graphics)

x <- -1:(4*6 + 1)
fx <- dwilcox(x, 4, 6)
Fx <- pwilcox(x, 4, 6)

layout(rbind(1,2), widths=1, heights=c(3,2))
plot(x, fx,type='h', col="violet",
     main= "Probabilities (density) of Wilcoxon-Statist.(n=6,m=4)")
plot(x, Fx,type="s", col="blue",
     main= "Distribution of Wilcoxon-Statist.(n=6,m=4)")
abline(h=0:1, col="gray20",lty=2)
layout(1)# set back

N <- 200
hist(U <- rwilcox(N, m=4,n=6), breaks=0:25 - 1/2,
     border="red", col="pink", sub = paste("N =",N))
mtext("N * f(x),  f() = true \"density\"", side=3, col="blue")
 lines(x, N*fx, type='h', col='blue', lwd=2)
points(x, N*fx, cex=2)

## Better is a Quantile-Quantile Plot
qqplot(U, qw <- qwilcox((1:N - 1/2)/N, m=4,n=6),
       main = paste("Q-Q-Plot of empirical and theoretical quantiles",
                    "Wilcoxon Statistic,  (m=4, n=6)",sep="\n"))
n <- as.numeric(names(print(tU <- table(U))))
text(n+.2, n+.5, labels=tU, col="red")
```

---

window *Time Windows*

Description

`window` is a generic function which extracts the subset of the object `x` observed between the times `start` and `end`. If a frequency is specified, the series is then re-sampled at the new frequency.

Usage

```
window(x, ...)
## S3 method for class 'ts':
window(x, ...)
## Default S3 method:
window(x, start = NULL, end = NULL,
       frequency = NULL, deltat = NULL, extend = FALSE, ...)

window(x, ...) <- value
## S3 replacement method for class 'ts':
window(x, start, end, frequency, deltat, ...) <- value
```

Arguments

x	a time-series (or other object if not replacing values).
start	the start time of the period of interest.
end	the end time of the period of interest.
frequency, deltat	the new frequency can be specified by either (or both if they are consistent).
extend	logical. If true, the start and end values are allowed to extend the series. If false, attempts to extend the series give a warning and are ignored.
...	further arguments passed to or from other methods.
value	replacement values.

Details

The start and end times can be specified as for ts. If there is no observation at the new start or end, the immediately following (start) or preceding (end) observation time is used.

The replacement function has a method for ts objects, and is allowed to extend the series (with a warning). There is no default method.

Value

The value depends on the method. window.default will return a vector or matrix with an appropriate tsp attribute.

window.ts differs from window.default only in ensuring the result is a ts object.

If extend = TRUE the series will be padded with NAs if needed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

time, ts.

Examples

```
window(presidents, 1960, c(1969,4)) # values in the 1960's
window(presidents, deltat=1)  # All Qtr1s
window(presidents, start=c(1945,3), deltat=1)  # All Qtr3s
window(presidents, 1944, c(1979,2), extend=TRUE)

pres <- window(presidents, 1945, c(1949,4)) # values in the 1940's
window(pres, 1945.25, 1945.50) <- c(60, 70)
window(pres, 1944, 1944.75) <- 0 # will generate a warning
window(pres, c(1945,4), c(1949,4), frequency=1) <- 85:89
pres
```

xtabs *Cross Tabulation*

Description

Create a contingency table from cross-classifying factors, usually contained in a data frame, using a formula interface.

Usage

```
xtabs(formula = ~., data = parent.frame(), subset, na.action,
      exclude = c(NA, NaN), drop.unused.levels = FALSE)
```

Arguments

formula	a formula object with the cross-classifying variables (separated by +) on the right hand side (or an object which can be coerced to a formula). Interactions are not allowed. On the left hand side, one may optionally give a vector or a matrix of counts; in the latter case, the columns are interpreted as corresponding to the levels of a variable. This is useful if the data have already been tabulated, see the examples below.
data	an optional matrix or data frame (or similar: see model.frame) containing the variables in the formula formula. By default the variables are taken from environment(formula).
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs.
exclude	a vector of values to be excluded when forming the set of levels of the classifying factors.
drop.unused.levels	a logical indicating whether to drop unused levels in the classifying factors. If this is FALSE and there are unused levels, the table will contain zero marginals, and a subsequent chi-squared test for independence of the factors will not work.

Details

There is a summary method for contingency table objects created by table or xtabs, which gives basic information and performs a chi-squared test for independence of factors (note that the function chisq.test currently only handles 2-d tables).

If a left hand side is given in formula, its entries are simply summed over the cells corresponding to the right hand side; this also works if the lhs does not give counts.

Value

A contingency table in array representation of class c("xtabs", "table"), with a "call" attribute storing the matched call.

See Also

table for traditional cross-tabulation, and as.data.frame.table which is the inverse operation of xtabs (see the DF example below).

Examples

```
## 'esoph' has the frequencies of cases and controls for all levels of
## the variables 'agegp', 'alcgp', and 'tobgp'.
xtabs(cbind(ncases, ncontrols) ~ ., data = esoph)
## Output is not really helpful ... flat tables are better:
ftable(xtabs(cbind(ncases, ncontrols) ~ ., data = esoph))
## In particular if we have fewer factors ...
ftable(xtabs(cbind(ncases, ncontrols) ~ agegp, data = esoph))

## This is already a contingency table in array form.
DF <- as.data.frame(UCBAdmissions)
## Now 'DF' is a data frame with a grid of the factors and the counts
## in variable 'Freq'.
DF
## Nice for taking margins ...
xtabs(Freq ~ Gender + Admit, DF)
## And for testing independence ...
summary(xtabs(Freq ~ ., DF))

## Create a nice display for the warp break data.
warpbreaks$replicate <- rep(1:9, len = 54)
ftable(xtabs(breaks ~ wool + tension + replicate, data = warpbreaks))
```

Chapter 8

The `tools` package

`tools-package` *Tools for Package Development*

Description

Tools for package development, administration and documentation.

Details

This package contains tools for manipulating R packages and their documentation.

For a complete list of functions, use `library(help="tools")`.

Author(s)

Kurt Hornik and Friedrich Leisch

Maintainer: R Core Team ⟨R-core@r-project.org⟩

`buildVignettes` *List and Build Package Vignettes*

Description

Run `Sweave` and `texi2dvi` on all vignettes of a package.

Usage

```
buildVignettes(package, dir, lib.loc = NULL, quiet = TRUE, clean = TRUE)
pkgVignettes(package, dir, lib.loc = NULL)
```

Arguments

package	a character string naming an installed package. If given, Sweave files are searched in subdirectory 'doc'.
dir	a character string specifying the path to a package's root source directory. This subdirectory 'inst/doc' is searched for Sweave files.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for package.
quiet	logical. Run Sweave and texi2dvi in quiet mode.
clean	Remove all files generated by the build, even if there were copies there before.

Value

buildVignettes is called for its side effect of creating the PDF versions of all vignettes.

pkgVignettes returns an object of class "pkgVignettes".

---

charsets *Conversion Tables between Character Sets*

Description

charset_to_Unicode is a matrix of Unicode points with columns for the common 8-bit encodings.

Adobe_glyphs is a dataframe which gives Adobe glyph names for Unicode points. It has two character columns, "adobe" and "unicode" (a 4-digit hex representation).

Usage

```
charset_to_Unicode

Adobe_glyphs
```

Details

charset_to_Unicode is an integer matrix of class c("noquote", "noquote") so prints in hexadecimal. The mappings are those used by libiconv: there are differences in the way quotes and minus/hyphen are mapped between sources (and the postscript encoding files use a different mapping).

Adobe_glyphs include all the Adobe glyph names which correspond to single Unicode characters. It is sorted by Unicode point and within a point alphabetically on the glyph(there can be more than one name for a Unicode point). The data are in the file '*R_HOME*/share/encodings/Adobe_glyphlist'.

Source

http://partners.adobe.com/public/developer/en/opentype/glyphlist.txt

Examples

```
## find Adobe names for ISOLatin2 chars.
latin2 <- charset_to_Unicode[, "ISOLatin2"]
aUnicode <- as.numeric(paste("0x", Adobe_glyphs$unicode, sep=""))
keep <- aUnicode %in% latin2
aUnicode <- aUnicode[keep]
aAdobe <- Adobe_glyphs[keep, 1]
## first match
aLatin2 <- aAdobe[match(latin2, aUnicode)]
## all matches
bLatin2 <- lapply(1:256, function(x) aAdobe[aUnicode == latin2[x]])
format(bLatin2, justify="none")
```

---

checkFF *Check Foreign Function Calls*

Description

Performs checks on calls to compiled code from R code. Currently only checks whether the interface functions such as `.C` and `.Fortran` are called with a "`NativeSymbolInfo`" first argument or with argument `PACKAGE` specified, which is highly recommended to avoid name clashes in foreign function calls.

Usage

```
checkFF(package, dir, file, lib.loc = NULL,
        verbose = getOption("verbose"))
```

Arguments

`package`	a character string naming an installed package. If given, the installed R code of the package is checked.
`dir`	a character string specifying the path to a package's root source directory. This should contain the subdirectory 'R' (for R code). Only used if `package` is not given.
`file`	the name of a file containing R code to be checked. Used if neither `package` nor `dir` are given.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. The specified library trees are used to search for `package`.
`verbose`	a logical. If `TRUE`, additional diagnostics are printed (and the result is returned invisibly).

Details

Note that we can only check if the `name` argument is a symbol or a character string, not what class of object the symbol resolves to at run-time.

If the package has a name space and *if* that contains a `useDynLib` directive, calls in top-level functions in the package are not reported as their symbols will be preferentially looked up in the DLL named in the first `useDynLib` directive.

Value

An object of class "checkFF", which currently is a list of the (parsed) foreign function calls with a character first argument and no PACKAGE argument.

There is a print method to display the information contained in such objects.

Warning

This function is still experimental. Both name and interface might change in future versions.

See Also

.C, .Fortran; Foreign.

Examples

```
checkFF(package = "stats", verbose = TRUE)
```

---

checkMD5sums *Check and Create MD5 Checksum Files*

Description

checkMD5sums checks the files against a file 'MD5'.

Usage

```
checkMD5sums(package, dir)
```

Arguments

package	the name of an installed package
dir	the path to the top-level directory of an installed package.

Details

The file 'MD5' which is created is in a format which can be checked by md5sum -c MD5 if a suitable command-line version of md5sum is available. (One is supplied in the bundle at http://www.murdoch-sutherland.com/Rtools/tools.zip.)

If dir is missing, an installed package of name package is searched for.

The private function tools:::.installMD5sums is used to create MD5 files in the Windows build.

Value

checkMD5sums returns a logical, NA if there is no 'MD5' file to be checked.

See Also

md5sum

checkTnF *Check R Packages or Code for T/F*

Description

Checks the specified R package or code file for occurrences of `T` or `F`, and gathers the expression containing these. This is useful as in R `T` and `F` are just variables which are set to the logicals `TRUE` and `FALSE` by default, but are not reserved words and hence can be overwritten by the user. Hence, one should always use `TRUE` and `FALSE` for the logicals.

Usage

```
checkTnF(package, dir, file, lib.loc = NULL)
```

Arguments

`package`	a character string naming an installed package. If given, the installed R code and the examples in the documentation files of the package are checked. R code installed as an image file cannot be checked.
`dir`	a character string specifying the path to a package's root source directory. This must contain the subdirectory 'R' (for R code), and should also contain 'man' (for documentation). Only used if `package` is not given. If used, the R code files and the examples in the documentation files are checked.
`file`	the name of a file containing R code to be checked. Used if neither `package` nor `dir` are given.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. The specified library trees are used to search for `package`.

Value

An object of class `"checkTnF"` which is a list containing, for each file where occurrences of `T` or `F` were found, a list with the expressions containing these occurrences. The names of the list are the corresponding file names.

There is a `print` method for nicely displaying the information contained in such objects.

Warning

This function is still experimental. Both name and interface might change in future versions.

`checkVignettes` *Check Package Vignettes*

Description

Check all `Sweave` files of a package by running `Sweave` and/or `Stangle` on them. All R source code files found after the tangling step are `source`ed to check whether all code can be executed without errors.

Usage

```
checkVignettes(package, dir, lib.loc = NULL,
               tangle = TRUE, weave = TRUE, latex = FALSE,
               workdir = c("tmp", "src", "cur"),
               keepfiles = FALSE)
```

Arguments

`package`	a character string naming an installed package. If given, Sweave files are searched in subdirectory 'doc'.
`dir`	a character string specifying the path to a package's root source directory. This subdirectory 'inst/doc' is searched for Sweave files.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. The specified library trees are used to search for `package`.
`tangle`	Perform a tangle and `source` the extracted code?
`weave`	Perform a weave?
`latex`	logical: if `tangle`, `weave` and `latex` are `TRUE` and there is no 'Makefile' in the vignettes directory, run the tangled files through `pdflatex`.
`workdir`	Directory used as working directory while checking the vignettes. If `"tmp"` then a temporary directory is created, this is the default. If `"src"` then the directory containing the vignettes itself is used, if `"cur"` then the current working directory of R is used.
`keepfiles`	Delete file in temporary directory? This option is ignored when `workdir != "tmp"`.

Value

An object of class `"checkVignettes"` which is a list with the error messages found during the tangle and weave steps. There is a `print` method for nicely displaying the information contained in such objects.

codoc *Check Code/Documentation Consistency*

Description

Find inconsistencies between actual and documented 'structure' of R objects in a package. `codoc` compares names and optionally also corresponding positions and default values of the arguments of functions. `codocClasses` and `codocData` compare slot names of S4 classes and variable names of data sets, respectively.

Usage

```
codoc(package, dir, lib.loc = NULL,
      use.values = NULL, verbose = getOption("verbose"))
codocClasses(package, lib.loc = NULL)
codocData(package, lib.loc = NULL)
```

Arguments

`package`	a character string naming an installed package.
`dir`	a character string specifying the path to a package's root source directory. This must contain the subdirectories 'man' with R documentation sources (in Rd format) and 'R' with R code. Only used if `package` is not given.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. The specified library trees are used to search for `package`.
`use.values`	if `FALSE`, do not use function default values when comparing code and docs. Otherwise, compare *all* default values if `TRUE`, and only the ones documented in the usage otherwise (default).
`verbose`	a logical. If `TRUE`, additional diagnostics are printed.

Details

The purpose of `codoc` is to check whether the documented usage of function objects agrees with their formal arguments as defined in the R code. This is not always straightforward, in particular as the usage information for methods to generic functions often employs the name of the generic rather than the method.

The following algorithm is used. If an installed package is used, it is loaded (unless it is the **base** package), after possibly detaching an already loaded version of the package. Otherwise, if the sources are used, the R code files of the package are collected and sourced in a new environment. Then, the usage sections of the Rd files are extracted and parsed 'as much as possible' to give the formals documented. For interpreted functions in the code environment, the formals are compared between code and documentation according to the values of the argument `use.values`. Synopsis sections are used if present; their occurrence is reported if `verbose` is true.

If a package has a name space both exported and unexported objects are checked, as well as registered S3 methods. (In the unlikely event of differences the order is exported objects in the package, registered S3 methods and finally objects in the name space and only the first found is checked.)

Currently, the R documentation format has no high-level markup for the basic 'structure' of classes and data sets (similar to the usage sections for function synopses). Variable names for data frames in

documentation objects obtained by suitably editing 'templates' created by `prompt` are recognized by `codocData` and used provided that the documentation object is for a single data frame (i.e., only has one alias). `codocClasses` analogously handles slot names for classes in documentation objects obtained by editing shells created by `promptClass`.

Help files named '*pkgname*-defunct.Rd' for the appropriate *pkgname* are checked more loosely, as they may have undocumented arguments.

Value

`codoc` returns an object of class `"codoc"`. Currently, this is a list which, for each Rd object in the package where an inconsistency was found, contains an element with a list of the mismatches (which in turn are lists with elements `code` and `docs`, giving the corresponding arguments obtained from the function's code and documented usage).

`codocClasses` and `codocData` return objects of class `"codocClasses"` and `"codocData"`, respectively, with a structure similar to class `"codoc"`.

There are `print` methods for nicely displaying the information contained in such objects.

Warning

Both `codocClasses` and `codocData` are still experimental. Names, interfaces and values might change in future versions.

Note

The default for `use.values` has been changed from `FALSE` to `NULL`, for R versions 1.9.0 and later.

See Also

`undoc`, `QC`

---

delimMatch *Delimited Pattern Matching*

Description

Match delimited substrings in a character vector, with proper nesting.

Usage

```
delimMatch(x, delim = c("{", "}"), syntax = "Rd")
```

Arguments

`x`	a character vector.
`delim`	a character vector of length 2 giving the start and end delimiters. Future versions might allow for arbitrary regular expressions.
`syntax`	currently, always the string `"Rd"` indicating Rd syntax (i.e., '`%`' starts a comment extending till the end of the line, and '`\`' escapes). Future versions might know about other syntax, perhaps via 'syntax tables' allowing to flexibly specify comment, escape, and quote characters.

Value

An integer vector of the same length as `x` giving the starting position (in characters) of the first match, or -1 if there is none, with attribute `"match.length"` giving the length (in characters) of the matched text (or -1 for no match).

See Also

`regexpr` for 'simple' pattern matching.

Examples

```
x <- c("\\value{foo}", "function(bar)")
delimMatch(x)
delimMatch(x, c("(", ")"))
```

---

encoded_text_to_latex

Translate non-ASCII Text to LaTeX Escapes

Description

Translate non-ASCII characters in text to LaTeX escape sequences.

Usage

```
encoded_text_to_latex(x,
                      encoding = c("latin1", "latin2", "latin9",
                                   "UTF-8", "utf8"))
```

Arguments

`x`	a character vector.
`encoding`	the encoding to be assumed. `"latin9"` is officially ISO-8859-15 or Latin-9, but known as latin9 to LaTeX's `inputenc` package.

Details

Non-ASCII characters in `x` are replaced by an appropriate LaTeX escape sequence, or '`?`' if there is no appropriate sequence.

Even if there is an appropriate sequence, it may not be supported by the font in use. Hyphen is mapped to '`\-`'.

Value

A character vector of the same length as `x`.

See Also

`iconv`

Examples

```
x <- "fa\xE7ile"
encoded_text_to_latex(x, "latin1")
## Not run:
## create a tex file to show the upper half of 8-bit charsets
x <- rawToChar(as.raw(160:255), multiple=TRUE)
(x <- matrix(x, ncol=16, byrow=TRUE))
xx <- x
xx[] <- encoded_text_to_latex(x, "latin1") # or latin2 or latin9
xx <- apply(xx, 1, paste, collapse="&")
con <- file("test-encoding.tex", "w")
header <- c(
"\\documentclass{article}",
"\\usepackage[T1]{fontenc}",
"\\usepackage{Rd}",
"\\begin{document}",
"\\HeaderA{test}{}{test}",
"\\begin{Details}\relax",
"\\Tabular{cccccccccccccccc}{")
trailer <- c("}", "\\end{Details}", "\\end{document}")
writeLines(header, con)
writeLines(paste(xx, "\\", sep=""), con)
writeLines(trailer, con)
close(con)
## and some UTF_8 chars
x <- intToUtf8(as.integer(
    c(160:383,0x0192,0x02C6,0x02C7,0x02CA,0x02D8,
      0x02D9, 0x02DD, 0x200C, 0x2018, 0x2019, 0x201C,
      0x201D, 0x2020, 0x2022, 0x2026, 0x20AC)),
               multiple=TRUE)
x <- matrix(x, ncol=16, byrow=TRUE)
xx <- x
xx[] <- encoded_text_to_latex(x, "UTF-8")
xx <- apply(xx, 1, paste, collapse="&")
con <- file("test-utf8.tex", "w")
writeLines(header, con)
writeLines(paste(xx, "\\", sep=""), con)
writeLines(trailer, con)
close(con)
## End(Not run)
```

---

fileutils *File Utilities*

Description

Utilities for listing files, and manipulating file paths.

Usage

```
file_path_as_absolute(x)
file_path_sans_ext(x)

list_files_with_exts(dir, exts, all.files = FALSE,
```

```
                        full.names = TRUE)
list_files_with_type(dir, type, all.files = FALSE,
                        full.names = TRUE, OS_subdirs = .OStype())
```

Arguments

x	character vector giving file paths.
dir	a character string with the path name to a directory.
exts	a character vector of possible file extensions.
all.files	a logical. If FALSE (default), only visible files are considered; if TRUE, all files are used.
full.names	a logical indicating whether the full paths of the files found are returned (default), or just the file names.
type	a character string giving the 'type' of the files to be listed, as characterized by their extensions. Currently, possible values are "code" (R code), "data" (data sets), "demo" (demos), "docs" (R documentation), and "vignette" (vignettes).
OS_subdirs	a character vector with the names of OS-specific subdirectories to possibly include in the listing of R code and documentation files. By default, the value of the environment variable R_OSTYPE, or if this is empty, the value of .Platform$OS.type, is used.

Details

file_path_as_absolute turns a possibly relative file path absolute, performing tilde expansion if necessary. Currently, only a single existing path can be given.

file_path_sans_ext returns the file paths without extensions. (Only purely alphanumeric extensions are recognized.)

list_files_with_exts returns the paths or names of the files in directory dir with extension matching one of the elements of exts. Note that by default, full paths are returned, and that only visible files are used.

list_files_with_type returns the paths of the files in dir of the given 'type', as determined by the extensions recognized by R. When listing R code and documentation files, files in OS-specific subdirectories are included if present according to the value of OS_subdirs. Note that by default, full paths are returned, and that only visible files are used.

See Also

file.path, file.info, list.files

Examples

```
dir <- file.path(R.home(), "library", "stats")
list_files_with_exts(file.path(dir, "demo"), "R")
list_files_with_type(file.path(dir, "demo"), "demo") # the same
file_path_sans_ext(list.files(file.path(R.home(), "modules")))
```

getDepList *Functions to Retrieve Dependency Information*

Description

Given a dependency matrix, will create a DependsList object for that package which will include the dependencies for that matrix, which ones are installed, which unresolved dependencies were found online, which unresolved dependencies were not found online, and any R dependencies.

Usage

```
getDepList(depMtrx, instPkgs, recursive = TRUE, local = TRUE,
           reduce = TRUE, lib.loc = NULL)

pkgDepends(pkg, recursive = TRUE, local = TRUE, reduce = TRUE,
           lib.loc = NULL)
```

Arguments

depMtrx	A dependency matrix as from package.dependencies
pkg	The name of the package
instPkgs	A matrix specifying all packages installed on the local system, as from installed.packages
recursive	Whether or not to include indirect dependencies
local	Whether or not to search only locally
reduce	Whether or not to collapse all sets of dependencies to a minimal value
lib.loc	What libraries to use when looking for installed packages. NULL indicates all library directories in the user's .libPaths().

Details

The function pkgDepends is a convenience function which wraps getDepList and takes as input a package name. It will then query installed.packages and also generate a dependency matrix, calling getDepList with this information and returning the result.

These functions will retrieve information about the dependencies of the matrix, resulting in a DependsList object. This is a list with four elements:

Depends A vector of the dependencies for this package.

Installed A vector of the dependencies which have been satisfied by the currently installed packages.

Found A list representing the dependencies which are not in Installed but were found online. This list has element names which are the URLs for the repositories in which packages were found and the elements themselves are vectors of package names which were found in the respective repositories. If local=TRUE, the Found element will always be empty.

R Any R version dependencies.

If `recursive` is `TRUE`, any package that is specified as a dependency will in turn have its dependencies included (and so on), these are known as indirect dependencies. If `recursive` is `FALSE`, only the dependencies directly stated by the package will be used.

If `local` is `TRUE`, the system will only look at the user's local install and not online to find unresolved dependencies.

If `reduce` is `TRUE`, the system will collapse the fields in the `DependsList` object such that a minimal set of dependencies are specified (for instance if there was ('foo', 'foo (>= 1.0.0)', 'foo (>= 1.3.0)'), it would only return 'foo (>= 1.3.0)').

Value

An object of class `DependsList`

Author(s)

Jeff Gentry

See Also

`installFoundDepends`

Examples

```
pkgDepends("tools", local = FALSE)
```

---

installFoundDepends

A function to install unresolved dependencies

Description

This function will take the `Found` element of a `pkgDependsList` object and attempt to install all of the listed packages from the specified repositories.

Usage

```
installFoundDepends(depPkgList, ...)
```

Arguments

`depPkgList`	A `Found` element from a `pkgDependsList` object
`...`	Arguments to pass on to `install.packages`

Details

This function takes as input the `Found` list from a `pkgDependsList` object. This list will have element names being URLs corresponding to repositories and the elements will be vectors of package names. For each element, `install.packages` is called for that URL to install all packages listed in the vector.

Author(s)

Jeff Gentry

See Also

pkgDepends, install.packages

Examples

```
## Set up a temporary directory to install packages to
tmp <- tempfile()
dir.create(tmp)

pDL <- pkgDepends("tools",local=FALSE)
installFoundDepends(pDL$Found, destdir=tmp)
```

---

makeLazyLoading *Lazy Loading of Packages*

Description

Tools for lazy loading of packages from a database.

Usage

```
makeLazyLoading(package, lib.loc = NULL, compress = TRUE,
                keep.source = getOption("keep.source.pkgs"))
```

Arguments

package	package name string
lib.loc	library trees, as in library
keep.source	logical; should sources be kept when saving from source
compress	logical; whether to compress entries on the database.

Details

A tool to set up packages for lazy loading from a database. For packages other than **base** you can use makeLazyLoading(package) to convert them to use lazy loading.

Author(s)

Luke Tierney and Brian Ripley

Examples

```
## set up package "splines" for lazy loading -- already done
## Not run:
tools:::makeLazyLoading("splines")
## End(Not run)
```

md5sum *Compute MD5 Checksums*

Description

Compute the 32-byte MD5 checksums of one or more files.

Usage

```
md5sum(files)
```

Arguments

files	character. The paths of file(s) to be check-summed.

Value

A character vector of the same length as `files`, with names equal to `files`. The elements will be `NA` for non-existent or unreadable files, otherwise a 32-character string of hexadecimal digits.

On Windows all files are read in binary mode (as the `md5sum` utilities there do): on other OSes the files are read in the default way.

See Also

`checkMD5sums`

Examples

```
md5sum(dir(R.home(), pattern="^COPY", full.names=TRUE))
```

package.dependencies *Check Package Dependencies*

Description

Parses and checks the dependencies of a package against the currently installed version of R (and other packages).

Usage

```
package.dependencies(x, check = FALSE,
                     depLevel = c("Depends", "Imports", "Suggests"))
```

Arguments

x	A matrix of package descriptions as returned by `available.packages`.
check	If `TRUE`, return logical vector of check results. If `FALSE`, return parsed list of dependencies.
depLevel	Whether to look for `Depends` or `Suggests` level dependencies.

Details

Currently we only check if the package conforms with the currently running version of R. In the future we might add checks for inter-package dependencies.

See Also

update.packages

QC *QC Checks for R Code and/or Documentation*

Description

Functions for performing various quality checks.

Usage

```
checkDocFiles(package, dir, lib.loc = NULL)
checkDocStyle(package, dir, lib.loc = NULL)
checkReplaceFuns(package, dir, lib.loc = NULL)
checkS3methods(package, dir, lib.loc = NULL)
```

Arguments

package	a character string naming an installed package.
dir	a character string specifying the path to a package's root source directory. This should contain the subdirectories 'R' (for R code) and 'man' with R documentation sources (in Rd format). Only used if package is not given.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for package.

Details

checkDocFiles checks, for all Rd files in a package, whether all arguments shown in the usage sections of the Rd file are documented in its arguments section. It also reports duplicated entries in the arguments section, and 'over-documented' arguments which are given in the arguments section but not in the usage. Note that the match is for the usage section and not a possibly existing synopsis section, as the usage is what gets displayed.

checkDocStyle investigates how (S3) methods are shown in the usages of the Rd files in a package. It reports the methods shown by their full name rather than using the Rd \method markup for indicating S3 methods. Earlier versions of R also reported about methods shown along with their generic, which typically caused problems for the documentation of the primary argument in the generic and its methods. With \method now being expanded in a way that class information is preserved, joint documentation is no longer necessarily a problem. (The corresponding information is still contained in the object returned by checkDocStyle.)

checkReplaceFuns checks whether replacement functions or S3/S4 replacement methods in the package R code have their final argument named value.

checkS3methods checks whether all S3 methods defined in the package R code have all arguments of the corresponding generic, with positional arguments of the generics in the same positions

for the method. As an exception, the first argument of a formula method *may* be called `formula` even if this is not the name used by the generic. The rules when `...` is involved are subtle: see the source code. Functions recognized as S3 generics are those with a call to `UseMethod` in their body, internal S3 generics (see InternalMethods), and S3 group generics (see `Math`). Possible dispatch under a different name is not taken into account. The generics are sought first in the given package, then in the **base** package and (currently) the packages **graphics**, **stats**, and **utils** added in R 1.9.0 by splitting the former **base**, and, if an installed package is tested, also in the loaded name spaces/packages listed in the package's 'DESCRIPTION' Depends field.

If using an installed package, the checks needing access to all R objects of the package will load the package (unless it is the **base** package), after possibly detaching an already loaded version of the package.

Value

The functions return objects of class the same as the respective function names containing the information about problems detected. There are `print` methods for nicely displaying the information contained in such objects.

Warning

These functions are still experimental. Names, interfaces and values might change in future versions.

`Rdindex` *Generate Index from Rd Files*

Description

Print a 2-column index table with names and titles from given R documentation files to a given output file or connection. The titles are nicely formatted between two column positions (typically 25 and 72, respectively).

Usage

```
Rdindex(RdFiles, outFile = "", type = NULL,
        width = 0.9 * getOption("width"), indent = NULL)
```

Arguments

`RdFiles`	a character vector specifying the Rd files to be used for creating the index, either by giving the paths to the files, or the path to a single directory with the sources of a package.
`outFile`	a connection, or a character string naming the output file to print to. `""` (the default) indicates output to the console.
`type`	a character string giving the documentation type of the Rd files to be included in the index, or `NULL` (the default). The type of an Rd file is typically specified via the `\docType` tag; if `type` is `"data"`, Rd files whose *only* keyword is `datasets` are included as well.
`width`	a positive integer giving the target column for wrapping lines in the output.
`indent`	a positive integer specifying the indentation of the second column. Must not be greater than `width/2`, and defaults to `width/3`.

Details

If a name is not a valid alias, the first alias (or the empty string if there is none) is used instead.

`Rdutils` *Rd Utilities*

Description

Utilities for computing on the information in Rd objects.

Usage

```
Rd_db(package, dir, lib.loc = NULL)
Rd_parse(file, text = NULL)
```

Arguments

`package`	a character string naming an installed package.
`dir`	a character string specifying the path to a package's root source directory. This should contain the subdirectory 'man' with R documentation sources (in Rd format). Only used if `package` is not given.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. The specified library trees are used to search for `package`.
`file`	a connection, or a character string giving the name of a file or a URL to read documentation in Rd format from.
`text`	character vector with documentation in Rd format. Elements are treated as if they were lines of a file.

Details

`Rd_db` builds a simple database of all Rd sources in a package, as a list of character vectors with the lines of the Rd files in the package. This is particularly useful for working on installed packages, where the individual Rd files in the sources are no longer available.

`Rd_parse` is a simple top-level Rd parser/analyzer. It returns a list with components

`meta` a list containing the Rd metadata (aliases, concepts, keywords, and documentation type);

`data` a data frame with the names (`tags`) and corresponding text (`vals`) of the top-level sections in the R documentation object;

`rest` top-level text not accounted for (currently, silently discarded by Rdconv, and hence usually the indication of a problem).

Note that at least for the time being, only the top-level structure is analyzed.

Warning

These functions are still experimental. Names, interfaces and values might change in future versions.

Examples

```
## Build the Rd db for the (installed) base package.
db <- Rd_db("base")
## Run Rd_parse on all entries in the Rd db.
db <- lapply(db, function(txt) Rd_parse(text = txt))
## Extract the metadata.
meta <- lapply(db, "[[", "meta")

## Keyword metadata per Rd file.
keywords <- lapply(meta, "[[", "keywords")
## Tabulate the keyword entries.
kw_table <- sort(table(unlist(keywords)))
## The 5 most frequent ones:
rev(kw_table)[1 : 5]
## The "most informative" ones:
kw_table[kw_table == 1]

## Concept metadata per Rd file.
concepts <- lapply(meta, "[[", "concepts")
## How many files already have \concept metadata?
sum(sapply(concepts, length) > 0)
## How many concept entries altogether?
length(unlist(concepts))
```

---

read.00Index *Read 00Index-style Files*

Description

Read item/description information from 00Index-style files. Such files are description lists rendered in tabular form, and currently used for the 'INDEX' and 'demo/00Index' files of add-on packages.

Usage

```
read.00Index(file)
```

Arguments

`file`	the name of a file to read data values from. If the specified file is `""`, then input is taken from the keyboard (in this case input can be terminated by a blank line). Alternatively, `file` can be a `connection`, which will be opened if necessary, and if so closed at the end of the function call.

Value

A character matrix with 2 columns named `"Item"` and `"Description"` which hold the items and descriptions.

See Also

`formatDL` for the inverse operation of creating a 00Index-style file from items and their descriptions.

readNEWS *Read R's NEWS file or a similar one*

Description

Read R's NEWS file or a similarly formatted one. This is an experimental feature, new in R 2.4.0 and may change in several ways.

Usage

```
readNEWS(file = file.path(R.home(), "NEWS"), trace = FALSE,
         chop = c("first", "1", "par1", "keepAll"))
checkNEWS(file = file.path(R.home(), "NEWS"))
```

Arguments

file	the name of the file which the data are to be read from. Alternatively, file can be a connection, which will be opened if necessary, and can also be a complete URL. For more details, see the file argument of read.table.
trace	logical indicating if the recursive reading should be traced, i.e., print what it is doing.
chop	a character string specifying how the news entries should be *chopped*; chop = "keepAll" saves the full entries.

Details

readNEWS() reads a NEWS file; checkNEWS() checks for common errors in formatting. Currently it detects an incorrect number of spaces before the "o" item marker.

Value

readNEWS() returns an (S3) object of class "newsTree"; effectively a list of lists which is a tree of NEWS entries.

checkNEWS() returns TRUE if no suspected errors are found, or prints a message for each suspected error and returns FALSE.

Note that this is still experimental and may change in the future.

Examples

```
NEWStr  <- readNEWS(trace = TRUE)# chop = "first" ( = "first non-empty")
## keep the full NEWS entry text i.e. "no chopping":
NEWStrA <- readNEWS(chop = "keepAll")
object.size(NEWStr)
object.size(NEWStrA) ## (no chopping) ==> about double the size

str(NEWStr, max.level = 3)

str(NEWStr[[c("2.3", "2.3.1")]], max.level=2, vec.len=1)

NEWStr [[c("2.3", "2.3.1", "NEW FEATURES")]]
NEWStrA[[c("2.4", "2.4.0", "NEW FEATURES")]]
```

```
# Check the current NEWS file

stopifnot(checkNEWS())
```

`showNonASCII` *Highlight non-ASCII characters*

Description

This function prints elements of a character vector which contain non-ASCII bytes, printing such bytes as a escape like `<fc>`.

Usage

```
showNonASCII(x)
```

Arguments

`x`	a character vector.

Details

This was originally written to help detect non-portable text in files in packages.

It prints all lines of the files which contain non-ACII characters, preceded by the line number and with non-ASCII bytes highlighted *via* `iconv(sub = "byte")`.

Examples

```
out <- c(
"fa\xE7ile test of showNonASCII():",
"\\details{",
"   This is a good line",
"   This has an \xfcmlaut in it.",
"   OK again.",
"}")
cat(out, file = "my.Rd", sep = "\n")

showNonASCII(readLines("my.Rd"))
unlink("my.Rd")
```

texi2dvi *Compile LaTeX Files*

Description

Run `latex` and `bibtex` until all cross-references are resolved and create either a dvi or PDF file.

Usage

```
texi2dvi(file, pdf = FALSE, clean = FALSE, quiet = TRUE,
         texi2dvi = getOption("texi2dvi"), texinputs = NULL)
```

Arguments

`file`	character. Name of LaTeX source file.
`pdf`	logical. If `TRUE`, a PDF file is produced instead of the default dvi file (`texi2dvi` command line option '`--pdf`').
`clean`	logical. If `TRUE`, all auxiliary files are removed (`texi2dvi` command line option '`--clean`'). May not work on some platforms.
`quiet`	logical. No output unless an error occurs. Ignored if emulation (see the `texi2dvi` argument) is used.
`texi2dvi`	character (or `NULL`). Script or program used to compile a TeX file to dvi or PDF, respectively. The default (selected by `NULL`) is to look for an executable on the search path and otherwise emulate the script with `system` calls.
`texinputs`	`NULL` or a character vector of paths to add to the LaTeX and bibtex input search paths.

Details

Despite the name, this is used in R to compile LaTeX files, specifically those generated from vignettes. As from R 2.7.1 it ensures that the '*R_HOME*/share/texmf' directory is in the `TEXINPUTS` path, so R style files such as 'Sweave' and 'Rd' will be found. The search path used is first the existing `TEXINPUTS` setting (or the current directory if unset), then elements of `texinputs`, then '*R_HOME*/share/texmf' and finally the default path. Analogous changes are made to `BIBINPUTS` and `BSTINPUTS`settings.

Author(s)

Achim Zeileis and R-core

tools-deprecated *Deprecated Objects in Package tools*

Description

The functions or variables listed here are provided for compatibility with older versions of R only, and may be defunct as soon as of the next release.

See Also

`Deprecated`, `Defunct`

undoc *Find Undocumented Objects*

Description

Finds the objects in a package which are undocumented, in the sense that they are visible to the user (or data objects or S4 classes provided by the package), but no documentation entry exists.

Usage

```
undoc(package, dir, lib.loc = NULL)
```

Arguments

package	a character string naming an installed package.
dir	a character string specifying the path to a package's root source directory. This must contain the subdirectory 'man' with R documentation sources (in Rd format), and at least one of the 'R' or 'data' subdirectories with R code or data objects, respectively.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for package.

Details

This function is useful for package maintainers mostly. In principle, *all* user-level R objects should be documented.

The **base** package is special as it contains the primitives and these do not have definitions available at code level. We provide equivalent closures in environments .ArgsEnv and .GenericArgsEnv in the **base** package that are used for various purposes: undoc("base") checks that all the primitives that are not language constructs are prototyped in those environments and no others are.

Value

An object of class "undoc" which is a list of character vectors containing the names of the undocumented objects split according to documentation type. This representation is still experimental, and might change in future versions.

There is a print method for nicely displaying the information contained in such objects.

See Also

codoc, QC

Examples

```
undoc("tools")                  # Undocumented objects in 'tools'
```

`vignetteDepends` *Retrieve Dependency Information for a Vignette*

Description

Given a vignette name, will create a DependsList object that reports information about the packages the vignette depends on.

Usage

```
vignetteDepends(vignette, recursive = TRUE, reduce = TRUE,
                local = TRUE, lib.loc = NULL)
```

Arguments

`vignette`	The path to the vignette source
`recursive`	Whether or not to include indirect dependencies
`reduce`	Whether or not to collapse all sets of dependencies to a minimal value
`local`	Whether or not to search only locally
`lib.loc`	What libraries to search in locally

Details

If `recursive` is `TRUE`, any package that is specified as a dependency will in turn have its dependencies included (and so on), these are known as indirect dependencies. If recursive is `FALSE`, only the dependencies directly named by the vignette will be used.

If `local` is `TRUE`, the system will only look at the user's local machine and not online to find dependencies.

If `reduce` is `TRUE`, the system will collapse the fields in the `DependsList` to the minimal set of dependencies (for instance if the dependencies were ('foo', 'foo (>= 1.0.0)', 'foo (>= 1.3.0)'), the return value would be 'foo (>= 1.3.0)').

Value

An object of class `DependsList`

Author(s)

Jeff Gentry

See Also

`pkgDepends`

Examples

```
gridEx <- system.file("doc", "grid.Snw", package = "grid")
vignetteDepends(gridEx)
```

`write_PACKAGES` *Generate PACKAGES files*

Description

Generate 'PACKAGES' and 'PACKAGES.gz' files for a repository of source or Mac/Windows binary packages.

Usage

```
write_PACKAGES(dir, fields = NULL,
               type = c("source", "mac.binary", "win.binary"),
               verbose = FALSE, unpacked = FALSE, subdirs = FALSE)
```

Arguments

`dir`	Character vector describing the location of the repository (directory including source or binary packages) to generate the 'PACKAGES' and 'PACKAGES.gz' files from and write them to.
`fields`	a character vector giving the fields to be used in the 'PACKAGES' and 'PACKAGES.gz' files in addition to the default ones, or `NULL` (default). The default corresponds to the fields needed by `available.packages`: `"Package"`, `"Bundle"`, `"Priority"`, `"Version"`, `"Depends"`, `"Suggests"`, `"Imports"` and `"Contains"`.
`type`	Type of packages: currently source '.tar.gz' archives, and Mac or Windows binary ('.tgz' or '.zip', respectively) packages are supported. Defaults to `"win.binary"` on Windows and to `"source"` otherwise.
`verbose`	logical. Should packages be listed as they are processed?
`unpacked`	a logical indicating whether the package contents are available in unpacked form or not (default).
`subdirs`	either logical (to indicate if subdirectories should be included) or a character vector of subdirectories to include.

Details

`type = "win.binary"` uses `unz` connections to read all 'DESCRIPTION' files contained in the (zipped) binary packages for Windows in the given directory `dir`, and builds 'PACKAGES' and 'PACKAGES.gz' files from this information.

Value

Invisibly returns the number of packages described in the resulting 'PACKAGES' and 'PACKAGES.gz' files. If 0, no packages were found and no files were written.

Note

Processing '.tar.gz' archives to extract the 'DESCRIPTION' files is quite slow.

This function can be useful on other OSes to prepare a repository to be accessed by Windows machines, so `type = "win.binary"` should work on all OSes.

Author(s)

Uwe Ligges and R-core.

See Also

See `read.dcf` and `write.dcf` for reading 'DESCRIPTION' files and writing the 'PACKAGES' and 'PACKAGES.gz' files.

Examples

```
## Not run:
write_PACKAGES("c:/myFolder/myRepository")  # on Windows
write_PACKAGES("/pub/RWin/bin/windows/contrib/2.1",
               type="win.binary")  # on Linux
## End(Not run)
```

---

xgettext *Extract Translatable Messages from R Files in a Package*

Description

For each file in the 'R' directory (including system-specific subdirectories) of a package, extract the unique arguments passed to `stop`, `warning`, `message`, `gettext` and `gettextf`, or to `ngettext`.

Usage

```
xgettext(dir, verbose = FALSE, asCall = TRUE)

xngettext(dir, verbose = FALSE)

xgettext2pot(dir, potFile)
```

Arguments

`dir`	the directory of a source package.
`verbose`	logical: should each file be listed as it is processed?
`asCall`	logical: if `TRUE` each argument is returned whole, otherwise the strings within each argument are extracted.
`potFile`	name of `po` template file to be produced. Defaults to `"R-`*pkgname*`.pot"` where *pkgname* is the basename of `dir`.

Details

Leading and trailing white space (space, tab and linefeed) is removed for calls to `gettext`, `gettextf`, `stop`, `warning`, and `message`, as it is by the internal code that passes strings for translation.

We look to see if these functions were called with `domain = NA` and if so omit the call if `asCall = TRUE`: note that the call might contain a call to `gettext` which would be visible if `asCall = FALSE`.

`xgettext2pot` calls `xgettext` and then `xngettext`, and writes a PO template file for use with the **GNU Gettext** tools. This ensures that the strings for simple translation are unique in the file (as **GNU Gettext** requires), but does not do so for `ngettext` calls (and the rules are not stated in the Gettext manual).

If applied to the **base** package, this also looks in the '.R' files in '*R_HOME*/share/R'.

Value

For `xgettext`, a list of objects of class `"xgettext"` (which has a `print` method), one per source file that potentially contains translatable strings.

For `xngettext`, a list of objects of class `"xngettext"`, which are themselves lists of length-2 character strings.

Examples

```
## Not run:
## in a source-directory build of R:
xgettext(file.path(R.home(), "src", "library", "splines"))
## End(Not run)
```

Chapter 9

The `utils` package

`utils-package` *The R Utils Package*

Description

R utility functions

Details

This package contains a collection of utility functions.

For a complete list, use `library(help="utils")`.

Author(s)

R Development Core Team and contributors worldwide

Maintainer: R Core Team ⟨R-core@r-project.org⟩

`alarm` *Alert the user*

Description

Gives an audible or visual signal to the user.

Usage

```
alarm()
```

Details

`alarm()` works by sending a `"\a"` character to the console. On most platforms this will ring a bell, beep, or give some other signal to the user (unless standard output has been redirected).

Value

No useful value is returned.

Examples

```
alarm()
```

`apropos` *Find Objects by (Partial) Name*

Description

`apropos()` returns a character vector giving the names of all objects in the search list matching `what`.

`find()` is a different user interface to the same task.

Usage

```
apropos(what, where = FALSE, ignore.case = TRUE, mode = "any")

find(what, mode = "any", numeric = FALSE, simple.words = TRUE)
```

Arguments

`what`	character string with name of an object, or more generally a regular expression to match against.
`where, numeric`	a logical indicating whether positions in the search list should also be returned
`ignore.case`	logical indicating if the search should be case-insensitive, `TRUE` by default. Note that in R versions prior to 2.5.0, the default was implicitly `ignore.case = FALSE`.
`mode`	character; if not `"any"`, only objects whose `mode` equals `mode` are searched.
`simple.words`	logical; if `TRUE`, the `what` argument is only searched as whole word.

Details

If `mode != "any"` only those objects which are of mode `mode` are considered. If `where` is `TRUE`, the positions in the search list are returned as the names attribute.

`find` is a different user interface for the same task as `apropos`. However, by default (`simple.words == TRUE`), only full words are searched with `grep(fixed = TRUE)`.

Note that in R versions prior to 2.5.0, `what` was allowed to be non-character, such that `find(cor)` worked as it does in S. This possibility has been dropped in line with the aim of minimizing all use of non-standard evaluation in R.

Value

For `apropos` character vector, sorted by name, possibly with names giving the (numerical) positions on the search path.

For `find`, either a character vector of environment names, or for `numeric = TRUE`, a numerical vector of positions on the search path, with names giving the names of the corresponding environments.

Author(s)

Kurt Hornik and Martin Maechler (May 1997).

See Also

`glob2rx` to convert wildcard patterns to regular expressions.

`objects` for listing objects from one place, `help.search` for searching the help system, `search` for the search path.

Examples

```
require(stats)

## Not run: apropos("lm")
apropos("GLM")                        # more than a dozen
## that may include internal objects starting '.__C__' if
## methods is attached
apropos("GLM", ignore.case = FALSE) # not one
apropos("lq")

cor <- 1:pi
find("cor")         #> ".GlobalEnv"   "package:stats"
find("cor", numeric=TRUE) # numbers with these names
find("cor", numeric=TRUE, mode="function")# only the second one
rm(cor)

## Not run: apropos(".", mode="list") # a long list

# need a DOUBLE backslash '\\' (in case you don't see it anymore)
apropos("\\[")

## Not run: # everything
length(apropos("."))

# those starting with 'pr'
apropos("^pr")

# the 1-letter things
apropos("^.$")
# the 1-2-letter things
apropos("^..?$")
# the 2-to-4 letter things
apropos("^.{2,4}$")

# the 8-and-more letter things
apropos("^.{8,}$")
table(nchar(apropos("^.{8,}$")))
## End(Not run)
```

BATCH *Batch Execution of R*

Description

Run R non-interactively with input from `infile` and send output (stdout/stderr) to another file.

Usage

```
R CMD BATCH [options] infile [outfile]
```

Arguments

`infile`	the name of a file with R code to be executed.
`options`	a list of R command line options, e.g., for setting the amount of memory available and controlling the load/save process. If `infile` starts with a '`-`', use '`--`' as the final option. The default options are '`--restore --save --no-readline`'.
`outfile`	the name of a file to which to write output. If not given, the name used is that of `infile`, with a possible '.R' extension stripped, and '.Rout' appended.

Details

Use `R CMD BATCH --help` to be reminded of the usage.

By default, the input commands are printed along with the output. To suppress this behavior, add `options(echo = FALSE)` at the beginning of `infile`, or use option '`--slave`'.

The `infile` can have end of line marked by LF or CRLF (but not just CR), and files with an incomplete last line (missing end of line (EOL) mark) are processed correctly.

A final expression '`proc.time()`' will be executed after the input script unless the latter calls `q(runLast=FALSE)` or is aborted.

Additional options can be set by the environment variable `R_BATCH_OPTIONS`: these come after '`--restore --save --no-readline`' and before any options given on the command line.

Note

Unlike `Splus BATCH`, this does not run the R process in the background. In most shells, `R CMD BATCH [options] infile [outfile] &` will do so.

Report bugs to ⟨r-bugs@r-project.org⟩.

`browseEnv` *Browse Objects in Environment*

Description

The `browseEnv` function opens a browser with list of objects currently in `sys.frame()` environment.

Usage

```
browseEnv(envir = .GlobalEnv, pattern,
          excludepatt = "^last\\.warning",
          html = .Platform$OS.type != "mac",
          expanded = TRUE, properties = NULL,
          main = NULL, debugMe = FALSE)
```

Arguments

`envir`	an `environment` the objects of which are to be browsed.
`pattern`	a regular expression for object subselection is passed to the internal `ls()` call.
`excludepatt`	a regular expression for *dropping* objects with matching names.
`html`	is used on non Macintosh machines to display the workspace on a HTML page in your favorite browser.
`expanded`	whether to show one level of recursion. It can be useful to switch it to `FALSE` if your workspace is large. This option is ignored if `html` is set to `FALSE`.
`properties`	a named list of global properties (of the objects chosen) to be showed in the browser; when `NULL` (as per default), user, date, and machine information is used.
`main`	a title string to be used in the browser; when `NULL` (as per default) a title is constructed.
`debugMe`	logical switch; if true, some diagnostic output is produced.

Details

Very experimental code. Only allows one level of recursion into object structures. The HTML version is not dynamic.

It can be generalized. See sources ('..../library/base/R/databrowser.R') for details.

`wsbrowser()` is currently just an internally used function; its argument list will certainly change.

Most probably, this should rather work through using the 'tkWidget' package (from `www.Bioconductor.org`).

See Also

`str`, `ls`.

Examples

```
if(interactive()) {
   ## create some interesting objects :
   ofa <- ordered(4:1)
   ex1 <- expression(1+ 0:9)
   ex3 <- expression(u,v, 1+ 0:9)
   example(factor, echo = FALSE)
   example(table, echo = FALSE)
   example(ftable, echo = FALSE)
   example(lm, echo = FALSE, ask = FALSE)
   example(str, echo = FALSE)

   ## and browse them:
   browseEnv()

   ## a (simple) function's environment:
   af12 <- approxfun(1:2, 1:2, method = "const")
   browseEnv(envir = environment(af12))
 }
```

`browseURL` *Load URL into a WWW Browser*

Description

Load a given URL into a WWW browser.

Usage

```
browseURL(url, browser = getOption("browser"), encodeIfNeeded = FALSE)
```

Arguments

`url`	a non-empty character string giving the URL to be loaded.
`browser`	a non-empty character string giving the name of the program to be used as hypertext browser. It should be in the PATH, or a full path specified. Alternatively, an R function to be called to invoke the browser.
`encodeIfNeeded`	Should the URL be encoded by `URLencode` before passing to the browser? This is not needed (and might be harmful) if the `browser` program/function itself does encoding, and can be harmful for `file://` URLs on some systems and for `http://` URLs passed to some CGI applications. Fortunately, most URLs do not need encoding.

Details

The default browser is set by option `"browser"`, in turn set by the environment variable `R_BROWSER` which is by default set in file '*R_HOME*/etc/Renviron' to a choice made manually or automatically when R was configured. (See `Startup` for where to override that default value.)

If `browser` supports remote control and R knows how to perform it, the URL is opened in any already running browser or a new one if necessary. This mechanism currently is available for browsers which support the `"-remote openURL(...)"` interface (which includes Opera 5/6, Mozilla >= 0.9.5 and Mozilla Firefox), Galeon, KDE konqueror (via kfmclient) and the GNOME interface to Mozilla. Netscape 7.0 and Opera 7 behave slightly differently, and you will need to open them first. Note that the type of browser is determined from its name, so this mechanism will only be used if the browser is installed under its canonical name.

Because `"-remote"` will use any browser displaying on the X server (whatever machine it is running on), the remote control mechanism is only used if `DISPLAY` points to the local host. This may not allow displaying more than one URL at a time from a remote host.

It is the caller's responsibility to encode `url` if necessary (see `URLencode`). This can be tricky for file URLs, where the format accepted can depend on the browser and OS.

Examples

```
## Not run:
## for KDE users who want to open files in a new tab
option(browser="kfmclient newTab")
browseURL("http://www.r-project.org")
## End(Not run)
```

`browseVignettes` *List Vignettes in an HTML Browser*

Description

List available vignettes in an HTML browser with links to PDF, LaTeX/noweb source, and (tangled) R code (if available).

Usage

```
browseVignettes(package = NULL, lib.loc = NULL, all = TRUE)

## S3 method for class 'browseVignettes':
print(x, ...)
```

Arguments

`package`	a character vector with the names of packages to search through, or `NULL` in which "all" packages (as defined by argument `all`) are searched.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known.
`all`	logical; if `TRUE` search all available packages in the library trees specified by `lib.loc`, and if `FALSE`, search only attached packages.
`x`	Object of class `browseVignettes`.
`...`	Further arguments, ignored by the `print` method.

Details

Function `browseVignettes` returns an object of the same class; the print method displays it as an HTML page in a browser (using `browseURL`).

See Also

`browseURL`, `vignette`

Examples

```
## Not run:
## List vignettes from all *attached* packages
browseVignettes(all = FALSE)

## List vignettes from a specific package
browseVignettes("grid")
## End(Not run)
```

`bug.report` *Send a Bug Report*

Description

Invokes an editor to write a bug report and optionally mail it to the automated r-bugs repository at ⟨r-bugs@r-project.org⟩. Some standard information on the current version and configuration of R are included automatically.

Usage

```
bug.report(subject = "",
           ccaddress = Sys.getenv("USER"),
           method = getOption("mailer"),
           address = "r-bugs@r-project.org",
           file = "R.bug.report")
```

Arguments

`subject`	Subject of the email. Please do not use single quotes (') in the subject! File separate bug reports for multiple bugs
`ccaddress`	Optional email address for copies (default is current user). Use `ccaddress = FALSE` for no copies.
`method`	Submission method, one of `"mailx"`, `"gnudoit"`, `"none"`, or `"ess"`.
`address`	Recipient's email address.
`file`	File to use for setting up the email (or storing it when method is `"none"` or sending mail fails).

Details

Currently direct submission of bug reports works only on Unix systems. If the submission method is `"mailx"`, then the default editor is used to write the bug report. Which editor is used can be controlled using `options`, type `getOption("editor")` to see what editor is currently defined. Please use the help pages of the respective editor for details of usage. After saving the bug report (in the temporary file opened) and exiting the editor the report is mailed using a Unix command line mail utility such as `mailx`. A copy of the mail is sent to the current user.

If method is `"gnudoit"`, then an emacs mail buffer is opened and used for sending the email.

If method is `"none"` or `NULL` (and in every case on Windows systems), then only an editor is opened to help writing the bug report. The report can then be copied to your favorite email program and be sent to the r-bugs list.

If method is `"ess"` the body of the mail is simply sent to stdout.

Value

Nothing useful.

When is there a bug?

If R executes an illegal instruction, or dies with an operating system error message that indicates a problem in the program (as opposed to something like "disk full"), then it is certainly a bug.

Taking forever to complete a command can be a bug, but you must make certain that it was really R's fault. Some commands simply take a long time. If the input was such that you KNOW it should have been processed quickly, report a bug. If you don't know whether the command should take a long time, find out by looking in the manual or by asking for assistance.

If a command you are familiar with causes an R error message in a case where its usual definition ought to be reasonable, it is probably a bug. If a command does the wrong thing, that is a bug. But be sure you know for certain what it ought to have done. If you aren't familiar with the command, or don't know for certain how the command is supposed to work, then it might actually be working right. Rather than jumping to conclusions, show the problem to someone who knows for certain.

Finally, a command's intended definition may not be best for statistical analysis. This is a very important sort of problem, but it is also a matter of judgement. Also, it is easy to come to such a conclusion out of ignorance of some of the existing features. It is probably best not to complain about such a problem until you have checked the documentation in the usual ways, feel confident that you understand it, and know for certain that what you want is not available. The mailing list `r-devel@r-project.org` is a better place for discussions of this sort than the bug list.

If you are not sure what the command is supposed to do after a careful reading of the manual this indicates a bug in the manual. The manual's job is to make everything clear. It is just as important to report documentation bugs as program bugs.

If the online argument list of a function disagrees with the manual, one of them must be wrong, so report the bug.

How to report a bug

When you decide that there is a bug, it is important to report it and to report it in a way which is useful. What is most useful is an exact description of what commands you type, from when you start R until the problem happens. Always include the version of R, machine, and operating system that you are using; type *`version`* in R to print this. To help us keep track of which bugs have been fixed and which are still open please send a separate report for each bug.

The most important principle in reporting a bug is to report FACTS, not hypotheses or categorizations. It is always easier to report the facts, but people seem to prefer to strain to posit explanations and report them instead. If the explanations are based on guesses about how R is implemented, they will be useless; we will have to try to figure out what the facts must have been to lead to such speculations. Sometimes this is impossible. But in any case, it is unnecessary work for us.

For example, suppose that on a data set which you know to be quite large the command `data.frame(x, y, z, monday, tuesday)` never returns. Do not report that `data.frame()` fails for large data sets. Perhaps it fails when a variable name is a day of the week. If this is so then when we got your report we would try out the `data.frame()` command on a large data set, probably with no day of the week variable name, and not see any problem. There is no way in the world that we could guess that we should try a day of the week variable name.

Or perhaps the command fails because the last command you used was a `[` method that had a bug causing R's internal data structures to be corrupted and making the `data.frame()` command fail from then on. This is why we need to know what other commands you have typed (or read from your startup file).

It is very useful to try and find simple examples that produce apparently the same bug, and somewhat useful to find simple examples that might be expected to produce the bug but actually do not. If you want to debug the problem and find exactly what caused it, that is wonderful. You should still report the facts as well as any explanations or solutions.

Invoking R with the '--vanilla' option may help in isolating a bug. This ensures that the site profile and saved data files are not read.

A bug report can be generated using the bug.report() function. This automatically includes the version information and sends the bug to the correct address. Alternatively the bug report can be emailed to ⟨r-bugs@r-project.org⟩ or submitted to the Web page at http://bugs.r-project.org.

Bug reports on **contributed packages** should be sent to the package maintainer rather than to r-bugs.

Author(s)

This help page is adapted from the Emacs manual and the R FAQ

See Also

R FAQ, also sessionInfo() from which you may add to the bug report.

---

capture.output *Send output to a character string or file*

Description

Evaluates its arguments with the output being returned as a character string or sent to a file. Related to sink in the same way that with is related to attach.

Usage

```
capture.output(..., file = NULL, append = FALSE)
```

Arguments

...	Expressions to be evaluated.
file	A file name or a connection, or NULL to return the output as a character vector. If the connection is not open, it will be opened initially and closed on exit.
append	logical. If file a file name or unopened connection, append or overwrite?

Details

An attempt is made to write output as far as possible to file if there is an error in evaluating the expressions, but for file = NULL all output will be lost.

Value

A character string (if file=NULL), or invisible NULL.

See Also

sink, textConnection

Examples

```
require(stats)
glmout <- capture.output(example(glm))
glmout[1:5]
capture.output(1+1, 2+2)
capture.output({1+1; 2+2})
## Not run:
## on Unix with enscript available
ps <- pipe("enscript -o tempout.ps","w")
capture.output(example(glm), file=ps)
close(ps)
## End(Not run)
```

---

chooseCRANmirror *Select a CRAN Mirror*

Description

Interact with the user to choose a CRAN mirror.

Usage

```
chooseCRANmirror(graphics = getOption("menu.graphics"))

getCRANmirrors(all = FALSE, local.only = FALSE)
```

Arguments

graphics	Logical. If true and **tcltk** and an X server are available, use a Tk widget, or if under the AQUA interface use a MacOS X widget, otherwise use menu.
all	Logical, get all known mirrors or only the ones flagged as OK.
local.only	Logical, try to get most recent list from CRAN or use file on local disk only.

Details

A list of mirrors is stored in file '*R_HOME*/doc/CRAN_mirrors.csv', but first an on-line list of current mirrors is consulted, and the file copy used only if the on-line list is inaccessible.

This function was originally written to support a Windows GUI menu item, but is also called by contrib.url if it finds the initial dummy value of options("repos").

Value

None for chooseCRANmirror(), this function is invoked for its side effect of updating options("repos").

getCRANmirrors() returns a data frame with mirror information.

See Also

setRepositories, contrib.url.

citation *Citing R and R Packages in Publications*

Description

How to cite R and R packages in publications.

Usage

```
citation(package = "base", lib.loc = NULL)
## S3 method for class 'citation':
toBibtex(object, ...)
## S3 method for class 'citationList':
toBibtex(object, ...)
```

Arguments

package	a character string with the name of a single package. An error occurs if more than one package name is given.
lib.loc	a character vector with path names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
object	return object of citation.
...	currently not used.

Details

The R core development team and the very active community of package authors have invested a lot of time and effort in creating R as it is today. Please give credit where credit is due and cite R and R packages when you use them for data analysis.

Execute function citation() for information on how to cite the base R system in publications. If the name of a non-base package is given, the function either returns the information contained in the CITATION file of the package or auto-generates citation information. In the latter case the package 'DESCRIPTION' file is parsed, the resulting citation object may be arbitrarily bad, but is quite useful (at least as a starting point) in most cases.

If only one reference is given, the print method shows both a text version and a BibTeX entry for it, if a package has more than one reference then only the text versions are shown. The BibTeX versions can be obtained using function toBibtex (see the examples below).

Value

An object of class "citationList".

See Also

citEntry

Examples

```
## the basic R reference
citation()

## references for a package -- might not have these installed
if(nchar(system.file(package="lattice"))) citation("lattice")
if(nchar(system.file(package="foreign"))) citation("foreign")

## extract the bibtex entry from the return value
x <- citation()
toBibtex(x)
```

---

citEntry *Writing Package CITATION Files*

Description

The 'CITATION' file of R packages contains an annotated list of references that should be used for citing the packages.

Usage

```
citEntry(entry, textVersion, header = NULL, footer = NULL, ...)
citHeader(...)
citFooter(...)
readCitationFile(file, meta = NULL)
```

Arguments

`entry`	a character string with a BibTeX entry type
`textVersion`	a character string with a text representation of the reference
`header`	a character string with optional header text
`footer`	a character string with optional footer text
`file`	a file name
`...`	see details below
`meta`	a list of package metadata as obtained by `packageDescription`, or `NULL` (default).

Details

The 'CITATION' file of an R package should be placed in the 'inst' subdirectory of the package source. The file is an R source file and may contain arbitrary R commands including conditionals and computations. The file is `source()`ed by the R parser in a temporary environment and all resulting objects of class `"citation"` (the return value of `citEntry`) are collected.

Typically the file will contain zero or more calls to `citHeader`, then one or more calls to `citEntry`, and finally zero or more calls to `citFooter`. `citHeader` and `citFooter` are simply wrappers to `paste`, and their `...` argument is passed on to `paste` as is.

Value

`citEntry` returns an object of class `"citation"`, `readCitationFile` returns an object of class `"citationList"`.

Entry Types

`citEntry` creates `"citation"` objects, which are modeled after BibTeX entries. The entry should be a valid BibTeX entry type, e.g.,

article: An article from a journal or magazine.

book: A book with an explicit publisher.

inbook: A part of a book, which may be a chapter (or section or whatever) and/or a range of pages.

incollection: A part of a book having its own title.

inproceedings: An article in a conference proceedings.

manual: Technical documentation like a software manual.

mastersthesis: A Master's thesis.

misc: Use this type when nothing else fits.

phdthesis: A PhD thesis.

proceedings: The proceedings of a conference.

techreport: A report published by a school or other institution, usually numbered within a series.

unpublished: A document having an author and title, but not formally published.

Entry Fields

The `...` argument of `citEntry` can be any number of BibTeX fields, including

address: The address of the publisher or other type of institution.

author: The name(s) of the author(s), either as a character string in the format described in the LaTeX book, or a `personList` object.

booktitle: Title of a book, part of which is being cited.

chapter: A chapter (or section or whatever) number.

editor: Name(s) of editor(s), same format as `author`.

institution: The publishing institution of a technical report.

journal: A journal name.

note: Any additional information that can help the reader. The first word should be capitalized.

number: The number of a journal, magazine, technical report, or of a work in a series.

pages: One or more page numbers or range of numbers.

publisher: The publisher's name.

school: The name of the school where a thesis was written.

series: The name of a series or set of books.

title: The work's title.

volume: The volume of a journal or multi-volume book.

year: The year of publication.

Examples

```
basecit <- system.file("CITATION", package="base")
source(basecit, echo=TRUE)
readCitationFile(basecit)
```

close.socket *Close a Socket*

Description

Closes the socket and frees the space in the file descriptor table. The port may not be freed immediately.

Usage

```
close.socket(socket, ...)
```

Arguments

socket	A socket object
...	further arguments passed to or from other methods.

Value

logical indicating success or failure

Author(s)

Thomas Lumley

See Also

make.socket, read.socket

combn *Generate All Combinations of n Elements, Taken m at a Time*

Description

Generate all combinations of the elements of x taken m at a time. If x is a positive integer, returns all combinations of the elements of seq(x) taken m at a time. If argument FUN is not NULL, applies a function given by the argument to each point. If simplify is FALSE, returns a list; otherwise returns an array, typically a matrix. ... are passed unchanged to the FUN function, if specified.

Usage

```
combn(x, m, FUN = NULL, simplify = TRUE, ...)
```

Arguments

x	vector source for combinations, or integer n for x <- seq(n).
m	number of elements to choose.
FUN	function to be applied to each combination; default NULL means the identity, i.e., to return the combination (vector of length m).
simplify	logical indicating if the result should be simplified to an array (typically a matrix); if FALSE, the function returns a list. Note that when simplify = TRUE as by default, the dimension of the result is simply determined from FUN(\emph{<1st combination>}), for efficiency reasons. This will badly fail if FUN(u) is not of constant length.
...	optionally, further arguments to FUN.

Value

a list or array (in nondegenerate cases), see the simplify argument above.

Author(s)

Scott Chasalow wrote the original in 1994 for S; R package **combinat** and documentation by Vince Carey ⟨stvjc@channing.harvard.edu⟩; small changes by the R core team, notably to return an array in all cases of simplify = TRUE, e.g., for combn(5,5).

References

Nijenhuis, A. and Wilf, H.S. (1978) *Combinatorial Algorithms for Computers and Calculators*; Academic Press, NY.

See Also

choose for fast computation of the *number* of combinations. expand.grid for creating a data frame from all combinations of factors or vectors.

Examples

```
combn(letters[1:4], 2)
(m <- combn(10, 5, min))   # minimum value in each combination
mm <- combn(15, 6, function(x) matrix(x, 2,3))
stopifnot(round(choose(10,5)) == length(m),
          c(2,3, round(choose(15,6))) == dim(mm))

## Different way of encoding points:
combn(c(1,1,1,1,2,2,2,3,3,4), 3, tabulate, nbins = 4)

## Compute support points and (scaled) probabilities for a
## Multivariate-Hypergeometric(n = 3, N = c(4,3,2,1)) p.f.:
# table.mat(t(combn(c(1,1,1,1,2,2,2,3,3,4), 3, tabulate,nbins=4)))
```

compareVersion *Compare Two Package Version Numbers*

Description

Compare two package version numbers to see which is later.

Usage

```
compareVersion(a, b)
```

Arguments

a, b	Character strings representing package version numbers.

Details

R package version numbers are of the form x.y-z for integers x, y and z, with components after x optionally missing (in which case the version number is older than those with the components present).

Value

0 if the numbers are equal, -1 if b is later and 1 if a is later (analogous to the C function strcmp).

See Also

package_version, library, packageStatus.

Examples

```
compareVersion("1.0", "1.0-1")
compareVersion("7.2-0","7.1-12")
```

COMPILE *Compile Files for Use with R*

Description

Compile given source files so that they can subsequently be collected into a shared library using R CMD SHLIB and be loaded into R using dyn.load().

Usage

```
R CMD COMPILE [options] srcfiles
```

Arguments

srcfiles	A list of the names of source files to be compiled. Currently, C, C++, Objective C, Objective C++ and FORTRAN are supported; the corresponding files should have the extensions '.c', '.cc' (or '.cpp' or '.C'), '.m', '.mm' (or '.M') and '.f', respectively.
options	A list of compile-relevant settings, such as special values for CFLAGS or FFLAGS, or for obtaining information about usage and version of the utility.

Details

Note that Ratfor is not supported. If you have Ratfor source code, you need to convert it to FORTRAN. On many Solaris systems mixing Ratfor and FORTRAN code will work.

Objective C and Objective C++ support is optional and will work only if the corresponding compilers were available at R configure time.

Note

Some binary distributions of R have COMPILE in a separate bundle, e.g. an R-devel RPM.

See Also

SHLIB, dyn.load; the section on "Customizing compilation under Unix" in "R Administration and Installation" (see the 'doc/manual' subdirectory of the R source tree).

---

count.fields *Count the Number of Fields per Line*

Description

count.fields counts the number of fields, as separated by sep, in each of the lines of file read.

Usage

```
count.fields(file, sep = "", quote = "\"'", skip = 0,
             blank.lines.skip = TRUE, comment.char = "#")
```

Arguments

file	a character string naming an ASCII data file, or a connection, which will be opened if necessary, and if so closed at the end of the function call.
sep	the field separator character. Values on each line of the file are separated by this character. By default, arbitrary amounts of whitespace can separate fields.
quote	the set of quoting characters
skip	the number of lines of the data file to skip before beginning to read data.
blank.lines.skip	logical: if TRUE blank lines in the input are ignored.
comment.char	character: a character vector of length one containing a single character or an empty string.

Details

This used to be used by `read.table` and can still be useful in discovering problems in reading a file by that function.

For the handling of comments, see `scan`.

Value

A vector with the numbers of fields found.

See Also

`read.table`

Examples

```
cat("NAME", "1:John", "2:Paul", file = "foo", sep = "\n")
count.fields("foo", sep = ":")
unlink("foo")
```

`data` *Data Sets*

Description

Loads specified data sets, or list the available data sets.

Usage

```
data(..., list = character(0), package = NULL, lib.loc = NULL,
     verbose = getOption("verbose"), envir = .GlobalEnv)
```

Arguments

`...`	a sequence of names or literal character strings.
`list`	a character vector.
`package`	a character vector giving the package(s) to look in for data sets, or `NULL`. By default, all packages in the search path are used, then the '`data`' subdirectory (if present) of the current working directory.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known.
`verbose`	a logical. If `TRUE`, additional diagnostics are printed.
`envir`	the environment where the data should be loaded.

Details

Currently, four formats of data files are supported:

1. files ending '.R' or '.r' are `source()` d in, with the R working directory changed temporarily to the directory containing the respective file. (`data` ensures that the **utils** package is attached, in case it had been run *via* `utils::data`.)
2. files ending '.RData' or '.rda' are `load()` ed.
3. files ending '.tab', '.txt' or '.TXT' are read using `read.table(..., header = TRUE)`, and hence result in a data frame.
4. files ending '.csv' or '.CSV' are read using `read.table(..., header = TRUE, sep = ";")`, and also result in a data frame.

If more than one matching file name is found, the first on this list is used.

The data sets to be loaded can be specified as a sequence of names or character strings, or as the character vector `list`, or as both.

For each given data set, the first two types ('.R' or '.r', and '.RData' or '.rda' files) can create several variables in the load environment, which might all be named differently from the data set. The second two ('.tab', '.txt', or '.TXT', and '.csv' or '.CSV' files) will always result in the creation of a single variable with the same name as the data set.

If no data sets are specified, `data` lists the available data sets. It looks for a new-style data index in the 'Meta' or, if this is not found, an old-style '00Index' file in the 'data' directory of each specified package, and uses these files to prepare a listing. If there is a 'data' area but no index, available data files for loading are computed and included in the listing, and a warning is given: such packages are incomplete. The information about available data sets is returned in an object of class `"packageIQR"`. The structure of this class is experimental. Where the datasets have a different name from the argument that should be used to retrieve them the index will have an entry like `beaver1 (beavers)` which tells us that dataset `beaver1` can be retrieved by the call `data(beaver)`.

If `lib.loc` and `package` are both `NULL` (the default), the data sets are searched for in all the currently loaded packages then in the 'data' directory (if any) of the current working directory.

If `lib.loc = NULL` but `package` is specified as a character vector, the specified package(s) are searched for first amongst loaded packages and then in the default library/ies (see `.libPaths`).

If `lib.loc` *is* specified (and not `NULL`), packages are searched for in the specified library/ies, even if they are already loaded from another library.

To just look in the 'data' directory of the current working directory, set `package = character(0)` (and `lib.loc = NULL`, the default).

Value

a character vector of all data sets specified, or information about all available data sets in an object of class `"packageIQR"` if none were specified.

Note

The data files can be many small files. On some file systems it is desirable to save space, and the files in the 'data' directory of an installed package can be zipped up as a zip archive 'Rdata.zip'. You will need to provide a single-column file 'filelist' of file names in that directory.

One can take advantage of the search order and the fact that a '.R' file will change directory. If raw data are stored in 'mydata.txt' then one can set up 'mydata.R' to read 'mydata.txt' and pre-process it, e.g., using `transform`. For instance one can convert numeric vectors to factors with

the appropriate labels. Thus, the '.R' file can effectively contain a metadata specification for the plaintext formats.

See Also

`help` for obtaining documentation on data sets, `save` for *creating* the second ('.rda') kind of data, typically the most efficient one.

Examples

```
require(utils)
data()                        # list all available data sets
try(data(package = "rpart") )# list the data sets in the rpart package
data(USArrests, "VADeaths")  # load the data sets 'USArrests' and 'VADeaths'
help(USArrests)               # give information on data set 'USArrests'
```

---

dataentry *Spreadsheet Interface for Entering Data*

Description

A spreadsheet-like editor for entering or editing data.

Usage

```
data.entry(..., Modes = NULL, Names = NULL)
dataentry(data, modes)
de(..., Modes = list(), Names = NULL)
```

Arguments

`...`	A list of variables: currently these should be numeric or character vectors or list containing such vectors.
`Modes`	The modes to be used for the variables.
`Names`	The names to be used for the variables.
`data`	A list of numeric and/or character vectors.
`modes`	A list of length up to that of `data` giving the modes of (some of) the variables. `list()` is allowed.

Details

The data entry editor is only available on some platforms and GUIs. Where available it provides a means to visually edit a matrix or a collection of variables (including a data frame) as described in the Notes section.

`data.entry` has side effects, any changes made in the spreadsheet are reflected in the variables. The functions `de`, `de.ncols`, `de.setup` and `de.restore` are designed to help achieve these side effects. If the user passes in a matrix, `X` say, then the matrix is broken into columns before `dataentry` is called. Then on return the columns are collected and glued back together and the result assigned to the variable `X`. If you don't want this behaviour use dataentry directly.

The primitive function is `dataentry`. It takes a list of vectors of possibly different lengths and modes (the second argument) and opens a spreadsheet with these variables being the columns. The columns of the dataentry window are returned as vectors in a list when the spreadsheet is closed.

`de.ncols` counts the number of columns which are supplied as arguments to `data.entry`. It attempts to count columns in lists, matrices and vectors. `de.setup` sets things up so that on return the columns can be regrouped and reassigned to the correct name. This is handled by `de.restore`.

Value

`de` and `dataentry` return the edited value of their arguments. `data.entry` invisibly returns a vector of variable names but its main value is its side effect of assigning new version of those variables in the user's workspace.

Resources

The data entry window responds to X resources of class `R_dataentry`. Resources `foreground`, `background` and `geometry` are utilized.

Note

The details of interface to the data grid may differ by platform and GUI. The following description applies to the X11-based implementation under Unix.

You can navigate around the grid using the cursor keys or by clicking with the (left) mouse button on any cell. The active cell is highlighted by thickening the surrounding rectangle. Moving to the right or down will scroll the grid as needed: there is no constraint to the rows or columns currently in use.

There are alternative ways to navigate using the keys. Return and (keypad) Enter and LineFeed all move down. Tab moves right and Shift-Tab move left. Home moves to the top left.

PageDown or Control-F moves down a page, and PageUp or Control-B up by a page. End will show the last used column and the last few rows used (in any column).

Using any other key starts an editing process on the currently selected cell: moving away from that cell enters the edited value whereas Esc cancels the edit and restores the previous value. When the editing process starts the cell is cleared. In numerical columns (the default) only letters making up a valid number (including `-.eE`) are accepted, and entering an invalid edited value (such as blank) enters `NA` in that cell. The last entered value can be deleted using the BackSpace or Del(ete) key. Only a limited number of characters (currently 29) can be entered in a cell, and if necessary only the start or end of the string will be displayed, with the omissions indicated by > or <. (The start is shown except when editing.)

Entering a value in a cell further down a column than the last used cell extends the variable and fills the gap (if any) by `NA`s (not shown on screen).

The column names can only be selected by clicking in them. This gives a popup menu to select the column type (currently Real (numeric) or Character) or to change the name. Changing the type converts the current contents of the column (and converting from Character to Real may generate `NA`s.) If changing the name is selected the header cell becomes editable (and is cleared). As with all cells, the value is entered by moving away from the cell by clicking elsewhere or by any of the keys for moving down (only).

New columns are created by entering values in them (and not by just assigning a new name). The mode of the column is auto-detected from the first value entered: if this is a valid number it gives a numeric column. Unused columns are ignored, so adding data in `var5` to a three-column grid adds one extra variable, not two.

The `Copy` button copies the currently selected cell: `paste` copies the last copied value to the current cell, and right-clicking selects a cell *and* copies in the value. Initially the value is blank, and attempts to paste a blank value will have no effect.

Control-L will refresh the display, recalculating field widths to fit the current entries.

In the default mode the column widths are chosen to fit the contents of each column, with a default of 10 characters for empty columns. you can specify fixed column widths by setting option `de.cellwidth` to the required fixed width (in characters). (set it to zero to return to variable widths). The displayed width of any field is limited to 600 pixels (and by the window width).

See Also

`vi`, `edit`: `edit` uses `dataentry` to edit data frames.

Examples

```
# call data entry with variables x and y
## Not run: data.entry(x,y)
```

`debugger` *Post-Mortem Debugging*

Description

Functions to dump the evaluation environments (frames) and to examine dumped frames.

Usage

```
dump.frames(dumpto = "last.dump", to.file = FALSE)
debugger(dump = last.dump)
```

Arguments

`dumpto`	a character string. The name of the object or file to dump to.
`to.file`	logical. Should the dump be to an R object or to a file?
`dump`	An R dump object created by `dump.frames`.

Details

To use post-mortem debugging, set the option `error` to be a call to `dump.frames`. By default this dumps to an R object `"last.dump"` in the workspace, but it can be set to dump to a file (a dump of the object produced by a call to `save`). The dumped object contain the call stack, the active environments and the last error message as returned by `geterrmessage`.

When dumping to file, `dumpto` gives the name of the dumped object and the file name has '.rda' appended.

A dump object of class `"dump.frames"` can be examined by calling `debugger`. This will give the error message and a list of environments from which to select repeatedly. When an environment is selected, it is copied and the `browser` called from within the copy.

If `dump.frames` is installed as the error handler, execution will continue even in non-interactive sessions. See the examples for how to dump and then quit.

Value

Invisible `NULL`.

Note

Functions such as `sys.parent` and `environment` applied to closures will not work correctly inside `debugger`.

If the error occurred when computing the default value of a formal argument the debugger will report "recursive default argument reference" when trying to examine that environment.

Of course post-mortem debugging will not work if R is too damaged to produce and save the dump, for example if it has run out of workspace.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`options` for setting `error` options; `recover` is an interactive debugger working similarly to `debugger` but directly after the error occurs.

Examples

```
## Not run:
options(error=quote(dump.frames("testdump", TRUE)))

f <- function() {
    g <- function() stop("test dump.frames")
    g()
}
f()   # will generate a dump on file "testdump.rda"
options(error=NULL)

## possibly in another R session
load("testdump.rda")
debugger(testdump)
Available environments had calls:
1: f()
2: g()
3: stop("test dump.frames")

Enter an environment number, or 0 to exit
Selection: 1
Browsing in the environment with call:
f()
Called from: debugger.look(ind)
Browse[1]> ls()
[1] "g"
Browse[1]> g
function() stop("test dump.frames")
<environment: 759818>
Browse[1]>
Available environments had calls:
1: f()
```

```
2: g()
3: stop("test dump.frames")

Enter an environment number, or 0 to exit
Selection: 0

## A possible setting for non-interactive sessions
options(error=quote({dump.frames(to.file=TRUE); q()}))
## End(Not run)
```

---

demo *Demonstrations of R Functionality*

Description

demo is a user-friendly interface to running some demonstration R scripts. demo() gives the list of available topics.

Usage

```
demo(topic, package = NULL, lib.loc = NULL,
     character.only = FALSE, verbose = getOption("verbose"))
```

Arguments

topic	the topic which should be demonstrated, given as a name or literal character string, or a character string, depending on whether character.only is FALSE (default) or TRUE. If omitted, the list of available topics is displayed.
package	a character vector giving the packages to look into for demos, or NULL. By default, all packages in the search path are used.
lib.loc	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
character.only	logical; if TRUE, use topic as character string.
verbose	a logical. If TRUE, additional diagnostics are printed.

Details

If no topics are given, demo lists the available demos. The corresponding information is returned in an object of class "packageIQR". The structure of this class is experimental. In earlier versions of R, an empty character vector was returned along with listing available demos.

See Also

source which is called by demo.

Examples

```
demo() # for attached packages

## All available demos:
demo(package = .packages(all.available = TRUE))

demo(lm.glm, package="stats")
## Not run:
 ch <- "scoping"
 demo(ch, character = TRUE)
## End(Not run)
```

---

download.file *Download File from the Internet*

Description

This function can be used to download a file from the Internet.

Usage

```
download.file(url, destfile, method, quiet = FALSE, mode = "w",
              cacheOK = TRUE)
```

Arguments

url	A character string naming the URL of a resource to be downloaded.
destfile	A character string with the name where the downloaded file is saved. Tilde-expansion is performed.
method	Method to be used for downloading files. Currently download methods "internal", "wget" and "lynx" are available, and there is a value "auto": see 'Details'. The method can also be set through the option "download.file.method": see options().
quiet	If TRUE, suppress status messages (if any), and the progress bar.
mode	character. The mode with which to write the file. Useful values are "w", "wb" (binary), "a" (append) and "ab". Only used for the "internal" method.
cacheOK	logical. Is a server-side cached value acceptable? Implemented for the "internal" and "wget" methods.

Details

The function download.file can be used to download a single file as described by url from the internet and store it in destfile. The url must start with a scheme such as "http://", "ftp://" or "file://".

If method = "auto" is chosen (the default), the internal method is chosen for "file://" URLs, and for the others provided capabilities("http/ftp") is true (which it almost always is). Otherwise methods "wget" and "lynx" are tried in turn.

cacheOK = FALSE is useful for "http://" URLs, and will attempt to get a copy directly from the site rather than from an intermediate cache. (Not all platforms support it.) It is used by available.packages.

The remaining details apply to method "internal" only.

Note that https:// connections are not supported.

See url for how "file://" URLs are interpreted, especially on Windows. This function does decode encoded URLs.

The timeout for many parts of the transfer can be set by the option timeout which defaults to 60 seconds.

The level of detail provided during transfer can be set by the quiet argument and the internet.info option. The details depend on the platform and scheme, but setting internet.info to 0 gives all available details, including all server responses. Using 2 (the default) gives only serious messages, and 3 or more suppresses all messages.

A progress bar tracks the transfer. If the file length is known, an equals represents 2% of the transfer completed: otherwise a dot represents 10Kb.

Method "wget" can be used with proxy firewalls which require user/password authentication if proper values are stored in the configuration file for wget.

Value

An (invisible) integer code, 0 for success and non-zero for failure. For the "wget" and "lynx" methods this is the status code returned by the external program. The "internal" method can return 1, but will in most cases throw an error.

Setting Proxies

This applies to the internal code only.

Proxies can be specified via environment variables. Setting "no_proxy" to "*" stops any proxy being tried. Otherwise the setting of "http_proxy" or "ftp_proxy" (or failing that, the all upper-case version) is consulted and if non-empty used as a proxy site. For FTP transfers, the username and password on the proxy can be specified by "ftp_proxy_user" and "ftp_proxy_password". The form of "http_proxy" should be "http://proxy.dom.com/" or "http://proxy.dom.com:8080/" where the port defaults to 80 and the trailing slash may be omitted. For "ftp_proxy" use the form "ftp://proxy.dom.com:3128/" where the default port is 21. These environment variables must be set before the download code is first used: they cannot be altered later by calling Sys.setenv.

Usernames and passwords can be set for HTTP proxy transfers via environment variable http_proxy_user in the form user:passwd. Alternatively, http_proxy can be of the form "http://user:pass@proxy.dom.com:8080/" for compatibility with wget. Only the HTTP/1.0 basic authentication scheme is supported.

Note

Methods "wget" and "lynx" are for historical compatibility. They will block all other activity on the R process.

For methods "wget" and "lynx" a system call is made to the tool given by method, and the respective program must be installed on your system and be in the search path for executables.

See Also

options to set the HTTPUserAgent, timeout and internet.info options.

url for a finer-grained way to read data from URLs.

url.show, available.packages, download.packages for applications

edit *Invoke a Text Editor*

Description

Invoke a text editor on an R object.

Usage

```
## Default S3 method:
edit(name = NULL, file = "", title = NULL,
     editor = getOption("editor"), ...)

vi(name = NULL, file = "")
emacs(name = NULL, file = "")
pico(name = NULL, file = "")
xemacs(name = NULL, file = "")
xedit(name = NULL, file = "")
```

Arguments

name	a named object that you want to edit. If name is missing then the file specified by file is opened for editing.
file	a string naming the file to write the edited version to.
title	a display name for the object being edited.
editor	a string naming the text editor you want to use. On Unix the default is set from the environment variables EDITOR or VISUAL if either is set, otherwise vi is used. On Windows it defaults to notepad.
...	further arguments to be passed to or from methods.

Details

edit invokes the text editor specified by editor with the object name to be edited. It is a generic function, currently with a default method and one for data frames and matrices.

data.entry can be used to edit data, and is used by edit to edit matrices and data frames on systems for which data.entry is available.

It is important to realize that edit does not change the object called name. Instead, a copy of name is made and it is that copy which is changed. Should you want the changes to apply to the object name you must assign the result of edit to name. (Try fix if you want to make permanent changes to an object.)

In the form edit(name), edit deparses name into a temporary file and invokes the editor editor on this file. Quitting from the editor causes file to be parsed and that value returned. Should an error occur in parsing, possibly due to incorrect syntax, no value is returned. Calling edit(), with no arguments, will result in the temporary file being reopened for further editing.

Note that deparsing is not perfect, and the object recreated after editing can differ in subtle ways from that deparsed: see dput and .deparseOpts. (The deparse options used are the same as the defaults for dump.) Editing a function will preserve its environment. See edit.data.frame for further changes that can occur when editing a data frame or matrix.

Currently only the internal editor in Windows makes use of the title option; it displays the given name in the window header.

Note

The functions `vi`, `emacs`, `pico`, `xemacs`, `xedit` rely on the corresponding editor being available and being on the path. This is system-dependent.

See Also

`edit.data.frame`, `data.entry`, `fix`.

Examples

```
## Not run:
# use xedit on the function mean and assign the changes
mean <- edit(mean, editor = "xedit")

# use vi on mean and write the result to file mean.out
vi(mean, file = "mean.out")
## End(Not run)
```

`edit.data.frame` *Edit Data Frames and Matrices*

Description

Use data editor on data frame or matrix contents.

Usage

```
## S3 method for class 'data.frame':
edit(name, factor.mode = c("character", "numeric"),
     edit.row.names = any(row.names(name) != 1:nrow(name)), ...)

## S3 method for class 'matrix':
edit(name, edit.row.names = !is.null(dn[[1]]), ...)
```

Arguments

`name`	A data frame or (numeric, logical or characer) matrix.
`factor.mode`	How to handle factors (as integers or using character levels) in a data frame.
`edit.row.names`	logical. Show the row names (if they exist) be displayed as a separate editable column? It is an error to ask for this on a matrix with `NULL` row names.
`...`	further arguments passed to or from other methods.

Details

At present, this only works on simple data frames containing numeric, logical or character vectors and factors, and numeric, logical or character matrices. Any other mode of matrix will give an error, and a warning is given when the matrix has a class (which will be discarded).

Data frame columns are coerced on input to *character* unless numeric (in the sense of `is.numeric`), logical or factor. A warning is given when classes are discarded. Special characters (tabs, non-printing ASCII, etc.) will be displayed as escape sequences.

Factors columns are represented in the spreadsheet as either numeric vectors (which are more suitable for data entry) or character vectors (better for browsing). After editing, vectors are padded with NA to have the same length and factor attributes are restored. The set of factor levels can not be changed by editing in numeric mode; invalid levels are changed to NA and a warning is issued. If new factor levels are introduced in character mode, they are added at the end of the list of levels in the order in which they encountered.

It is possible to use the data-editor's facilities to select the mode of columns to swap between numerical and factor columns in a data frame. Changing any column in a numerical matrix to character will cause the result to be coerced to a character matrix. Changing the mode of logical columns is not supported.

For a data frame, the row names will be taken from the original object if `edit.row.names = FALSE` and the number of rows is unchanged, and from the edited output if `edit.row.names = TRUE` and there are no duplicates. (If the `row.names` column is incomplete, it is extended by entries like `row223`.) In all other cases the row names are replaced by `seq(length=nrows)`.

For a matrix, colnames will be added (of the form `col7`) if needed. The rownames will be taken from the original object if `edit.row.names = FALSE` and the number of rows is unchanged (otherwise NULL), and from the edited output if `edit.row.names = TRUE`. (If the `row.names` column is incomplete, it is extended by entries like `row223`.)

Editing a matrix or data frame will lose all attributes apart from the row and column names.

Value

The edited data frame or matrix.

Note

`fix(dataframe)` works for in-place editing by calling this function.

If the data editor is not available, a dump of the object is presented for editing using the default method of `edit`.

At present the data editor is limited to 65535 rows.

Author(s)

Peter Dalgaard

See Also

`data.entry`, `edit`

Examples

```
## Not run:
edit(InsectSprays)
edit(InsectSprays, factor.mode="numeric")
## End(Not run)
```

example *Run an Examples Section from the Online Help*

Description

Run all the R code from the **Examples** part of R's online help topic `topic` with two possible exceptions, `dontrun` and `dontshow`, see 'Details' below.

Usage

```
example(topic, package = NULL, lib.loc = NULL,
        local = FALSE, echo = TRUE,
        verbose = getOption("verbose"),
        setRNG = FALSE, ask = getOption("example.ask"),
        prompt.prefix = abbreviate(topic, 6))
```

Arguments

`topic`	name or literal character string: the online `help` topic the examples of which should be run.
`package`	a character vector giving the package names to look into for example code, or `NULL`. By default, all packages in the search path are used.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
`local`	logical: if `TRUE` evaluate locally, if `FALSE` evaluate in the workspace.
`echo`	logical; if `TRUE`, show the R input when sourcing.
`verbose`	logical; if `TRUE`, show even more when running example code.
`setRNG`	logical or expression; if not `FALSE`, the random number generator state is saved, then initialized to a specified state, the example is run and the (saved) state is restored. `setRNG = TRUE` sets the same state as `R CMD check` does for running a package's examples. This is currently equivalent to `setRNG = {RNGkind("default", "default"); set.seed(1)}`.
`ask`	logical (or `"default"`) indicating if `par(ask=TRUE)` and `grid.prompt(ask=TRUE)` should be called before graphical output happens from the example code. The value `"default"` (the factory-fresh default) means to ask if `echo == TRUE` and the graphics device appears to be interactive. This parameter applies both to any currently opened device and to any devices opened by the example code.
`prompt.prefix`	character; prefixes the prompt to be used if `echo = TRUE`.

Details

If `lib.loc` is not specified, the packages are searched for amongst those already loaded, then in the specified libraries. If `lib.loc` is specified, they are searched for only in the specified libraries, even if they are already loaded from another library.

An attempt is made to load the package before running the examples, but this will not replace a package loaded from another location.

If `local=TRUE` objects are not created in the workspace and so not available for examination after `example` completes: on the other hand they cannot clobber objects of the same name in the workspace.

As detailed in the manual *Writing* R *Extensions*, the author of the help page can markup parts of the examples for two exception rules

`dontrun` encloses code that should not be run.

`dontshow` encloses code that is invisible on help pages, but will be run both by the package checking tools, and the `example()` function. This was previously `testonly`, and that form is still accepted.

If the examples file contains non-ASCII characters the encoding used will matter. If in a UTF-8 locale `example` first tries UTF-8 and then Latin-1. (This can be overridden by setting the encoding in the '.Rd' file.)

Value

The value of the last evaluated expression.

Note

The examples can be many small files. On some file systems it is desirable to save space, and the files in the 'R-ex' directory of an installed package can be zipped up as a zip archive 'Rex.zip'.

Author(s)

Martin Maechler and others

See Also

`demo`

Examples

```
example(InsectSprays)
## force use of the standard package 'stats':
example("smooth", package="stats", lib.loc=.Library)

## set RNG *before* example as when R CMD check is run:

r1 <- example(quantile, setRNG = TRUE)
x1 <- rnorm(1)
u <- runif(1)
## identical random numbers
r2 <- example(quantile, setRNG = TRUE)
x2 <- rnorm(1)
stopifnot(identical(r1, r2))
## but x1 and x2 differ since the RNG state from before example()
## differs and is restored!
x1; x2
```

file.edit *Edit One or More Files*

Description

Edit one or more files in a text editor.

Usage

```
file.edit(..., title = file, editor = getOption("editor"))
```

Arguments

...	one or more character vectors containing the names of the files to be edited.
title	the title to use in the editor; defaults to the filename.
editor	the text editor to be used.

Details

Path expansion (see path.expand will be done on names.

The behaviour of this function is very system dependent. Currently files can be opened only one at a time on Unix; on Windows, the internal editor allows multiple files to be opened, but has a limit of 50 simultaneous edit windows.

The title argument is used for the window caption in Windows, and is ignored on other platforms.

See Also

files, file.show, edit, fix,

Examples

```
## Not run:
# open two R scripts for editing
file.edit("script1.R", "script2.R")
## End(Not run)
```

file_test *Shell-style Tests on Files*

Description

Utility for shell-style file tests.

Usage

```
file_test(op, x, y)
```

Arguments

`op`	a character string specifying the test to be performed. Unary tests (only `x` is used) are `"-f"` (existence and not being a directory) and `"-d"` (existence and directory). Binary tests are `"-nt"` (strictly newer than, using the modification dates) and `"-ot"` (strictly older than): in both cases the test is false unless both files exist.
`x,y`	character vectors giving file paths.

Details

'Existence' here means being on the file system and accessible by the `stat` system call (or a 64-bit extension) – on a Unix-alike this requires execute permission on all of the directories in the path that leads to the file, but no permissions on the file itself.

See Also

`file.exists` which only tests for existence (`test -e` on some systems) but not for not being a directory.

`file.path`, `file.info`

Examples

```
dir <- file.path(R.home(), "library", "stats")
file_test("-d", dir)
file_test("-nt", file.path(dir, "R"), file.path(dir, "demo"))
```

`fix` *Fix an Object*

Description

`fix` invokes `edit` on `x` and then assigns the new (edited) version of `x` in the user's workspace.

Usage

```
fix(x, ...)
```

Arguments

`x`	the name of an R object, as a name or a character string.
`...`	arguments to pass to editor: see `edit`.

Details

The name supplied as `x` need not exist as an R object, in which case a function with no arguments and an empty body is supplied for editing.

Editing an R object may change it in ways other than are obvious: see the comment under `edit`. See `edit.data.frame` for changes that can occur when editing a data frame or matrix.

See Also

edit, edit.data.frame

Examples

```
## Not run:
 ## Assume 'my.fun' is a user defined function :
 fix(my.fun)
 ## now my.fun is changed
 ## Also,
 fix(my.data.frame) # calls up data editor
 fix(my.data.frame, factor.mode="char") # use of ...
## End(Not run)
```

---

flush.console *Flush Output to A Console*

Description

This does nothing except on console-based versions of R. On the Mac OS X and Windows GUIs, it ensures that the display of output in the console is current, even if output buffering is on.

Usage

```
flush.console()
```

---

format *Format Unordered and Ordered Lists*

Description

Format unordered (itemize) and ordered (enumerate) lists.

Usage

```
formatUL(x, label = "*", offset = 0,
         width = 0.9 * getOption("width"))
formatOL(x, type = "arabic", offset = 0, start = 1,
         width = 0.9 * getOption("width"))
```

Arguments

x	a character vector of list items.
label	a character string used for labelling the items.
offset	a non-negative integer giving the offset (indentation) of the list.
width	a positive integer giving the target column for wrapping lines in the output.

type	a character string specifying the 'type' of the labels in the ordered list. If "arabic" (default), arabic numerals are used. For "Alph" or "alph", single upper or lower case letters are employed (in this case, the number of the last item must not exceed 26. Finally, for "Roman" or "roman", the labels are given as upper or lower case roman numerals (with the number of the last item maximally 3899). type can be given as a unique abbreviation of the above, or as one of the HTML style tokens "1" (arabic), "A"/"a" (alphabetic), or "I"/"i" (roman), respectively.
start	a positive integer specifying the starting number of the first item in an ordered list.

Value

A character vector with the formatted entries.

See Also

formatDL for formatting description lists.

Examples

```
## A simpler recipe.
x <- c("Mix dry ingredients thoroughly.",
       "Pour in wet ingredients.",
       "Mix for 10 minutes.",
       "Bake for one hour at 300 degrees.")
## Format and output as an unordered list.
writeLines(formatUL(x))
## Format and output as an ordered list.
writeLines(formatOL(x))
## Ordered list using lower case roman numerals.
writeLines(formatOL(x, type = "i"))
## Ordered list using upper case letters and some offset.
writeLines(formatOL(x, type = "A", offset = 5))
```

---

getAnywhere *Retrieve an R Object, Including from a Name Space*

Description

These functions locates all objects with name matching its argument, whether visible on the search path, registered as an S3 method or in a name space but not exported. getAnywhere() returns the objects and argsAnywhere() returns the arguments of any objects that are functions.

Usage

```
getAnywhere(x)
argsAnywhere(x)
```

Arguments

x	a character string or name.

Details

The function looks at all loaded name spaces, whether or not they are associated with a package on the search list.

Where functions are found as an S3 method, an attempt is made to find which name space registered them. This may not be correct, especially if a name space is unloaded.

Value

For `getAnywhere()` an object of class `"getAnywhere"`. This is a list with components

`name`	the name searched for.
`objs`	a list of objects found
`where`	a character vector explaining where the object(s) were found
`visible`	logical: is the object visible
`dups`	logical: is the object identical to one earlier in the list.

Normally the structure will be hidden by the `print` method. There is a `[` method to extract one or more of the objects found.

For `argsAnywhere()` one or more argument lists as returned by `args`.

See Also

`get`, `getFromNamespace`, `args`

Examples

```
getAnywhere("format.dist")
getAnywhere("simpleLoess") # not exported from stats
argsAnywhere(format.dist)
```

`getFromNamespace` *Utility functions for Developing Namespaces*

Description

Utility functions to access and replace the non-exported functions in a name space, for use in developing packages with name spaces.

Usage

```
getFromNamespace(x, ns, pos = -1, envir = as.environment(pos))

assignInNamespace(x, value, ns, pos = -1,
                  envir = as.environment(pos))

fixInNamespace(x, ns, pos = -1, envir = as.environment(pos), ...)
```

Arguments

x	an object name (given as a character string).
value	an R object.
ns	a name space, or character string giving the name space.
pos	where to look for the object: see get.
envir	an alternative way to specify an environment to look in.
...	arguments to pass to the editor: see edit.

Details

The name space can be specified in several ways. Using, for example, ns = "stats" is the most direct, but a loaded package with a name space can be specified via any of the methods used for get: ns can also be the environment printed as <namespace:foo>.

getFromNamespace is similar to (but predates) the ::: operator, but is more flexible in how the name space is specified.

fixInNamespace invokes edit on the object named x and assigns the revised object in place of the original object. For compatibility with fix, x can be unquoted.

Value

getFromNamespace returns the object found (or gives an error).

assignInNamespace and fixInNamespace are invoked for their side effect of changing the object in the name space.

Note

assignInNamespace and fixInNamespace change the copy in the name space, but not any copies already exported from the name space, in particular an object of that name in the package (if already attached) and any copies already imported into other name spaces. They are really intended to be used *only* for objects which are not exported from the name space. They do attempt to alter a copy registered as an S3 method if one is found.

They can only be used to change the values of objects in the name space, not to create new objects.

See Also

get, fix, getS3method

Examples

```
getFromNamespace("findGeneric", "utils")
## Not run:
fixInNamespace("predict.ppr", "stats")
stats:::predict.ppr
getS3method("predict", "ppr")
## alternatively
fixInNamespace("predict.ppr", pos = 3)
fixInNamespace("predict.ppr", pos = "package:stats")
## End(Not run)
```

getS3method *Get An S3 Method*

Description

Get a method for an S3 generic, possibly from a name space.

Usage

```
getS3method(f, class, optional = FALSE)
```

Arguments

f	character: name of the generic.
class	character: name of the class.
optional	logical: should failure to find the generic or a method be allowed?

Details

S3 methods may be hidden in packages with name spaces, and will not then be found by get: this function can retrieve such functions, primarily for debugging purposes.

Value

The function found, or NULL if no function is found and optional = TRUE.

See Also

methods, get

Examples

```
require(stats)
exists("predict.ppr") # false
getS3method("predict", "ppr")
```

glob2rx *Change Wildcard or Globbing Pattern into Regular Expression*

Description

Change *wildcard* aka *globbing* patterns into the corresponding regular expressions (regexp).

Usage

```
glob2rx(pattern, trim.head = FALSE, trim.tail = TRUE)
```

Arguments

pattern	character vector
trim.head	logical specifying if leading "^.*" should be trimmed from the result.
trim.tail	logical specifying if trailing ".*$" should be trimmed from the result.

Details

This takes a wildcard as used by most shells and returns an equivalent regular expression. ? is mapped to . (match a single character), * to .* (match any string, including an empty one), and the pattern is anchored (it must start at the beginning and end at the end). Optionally, the resulting regexp is simplified.

Note that now even (, [and { can be used in pattern, but glob2rx() may not work correctly with arbitrary characters in pattern.

Value

A character vector of the same length as the input pattern where each wildcard is translated to the corresponding regular expression.

Author(s)

Martin Maechler, Unix/sed based version, 1991; current: 2004

See Also

regexp about regular expression, sub, etc about substitutions using regexps.

Examples

```
stopifnot(glob2rx("abc.*") == "^abc\\.",
          glob2rx("a?b.*") == "^a.b\\.",
          glob2rx("a?b.*", trim.tail=FALSE) == "^a.b\\..*$",
          glob2rx("*.doc") == "^.*\\.doc$",
          glob2rx("*.doc", trim.head=TRUE) == "\\.doc$",
          glob2rx("*.t*")  == "^.*\\.t",
          glob2rx("*.t??") == "^.*\\.t..$",
          glob2rx("*[*")  == "^.*\\["
)
```

---

head *Return the First or Last Part of an Object*

Description

Returns the first or last parts of a vector, matrix, table, data frame or function. Since head() and tail() are generic functions, they may also have been extended to other classes.

Usage

```
head(x, ...)
## Default S3 method:
head(x, n = 6, ...)
## S3 method for class 'data.frame':
head(x, n = 6, ...)
## S3 method for class 'matrix':
head(x, n = 6, ...)
## S3 method for class 'ftable':
head(x, n = 6, ...)
## S3 method for class 'table':
head(x, n = 6, ...)
## S3 method for class 'function':
head(x, n = 6, ...)

tail(x, ...)
## Default S3 method:
tail(x, n = 6, ...)
## S3 method for class 'data.frame':
tail(x, n = 6, ...)
## S3 method for class 'matrix':
tail(x, n = 6, addrownums = TRUE, ...)
## S3 method for class 'ftable':
tail(x, n = 6, addrownums = FALSE, ...)
## S3 method for class 'table':
tail(x, n = 6, addrownums = TRUE, ...)
## S3 method for class 'function':
tail(x, n = 6, ...)
```

Arguments

`x`	an object
`n`	a single integer. If positive, size for the resulting object: number of elements for a vector (including lists), rows for a matrix or data frame or lines for a function. If negative, all but the `n` last/first number of elements of `x`.
`addrownums`	if there are no row names, create them from the row numbers.
`...`	arguments to be passed to or from other methods.

Details

For matrices, 2-dim tables and data frames, `head()` (`tail()`) returns the first (last) `n` rows when `n > 0` or all but the last (first) `n` rows when `n < 0`. `head.matrix()` and `tail.matrix()` are exported. For functions, the lines of the deparsed function are returned as character strings.

If a matrix has no row names, then `tail()` will add row names of the form `"[n,]"` to the result, so that it looks similar to the last lines of `x` when printed. Setting `addrownums = FALSE` suppresses this behaviour.

Value

An object (usually) like `x` but generally smaller. For `ftable` objects `x`, a transformed `format(x)`.

Author(s)

Patrick Burns, improved and corrected by R-Core. Negative argument added by Vincent Goulet.

Examples

```
head(letters)
head(letters, n = -6)

head(freeny.x, n = 10)
head(freeny.y)

tail(letters)
tail(letters, n = -6)

tail(freeny.x)
tail(freeny.y)

tail(library)

head(stats::ftable(Titanic))
```

`help` *Documentation*

Description

These functions provide access to documentation. Documentation on a topic with name `name` (typically, an R object or a data set) can be displayed with either `help("name")` or `?name`.

Usage

```
help(topic, offline = FALSE, package = NULL,
     lib.loc = NULL, verbose = getOption("verbose"),
     try.all.packages = getOption("help.try.all.packages"),
     chmhelp = getOption("chmhelp"),
     htmlhelp = getOption("htmlhelp"),
     pager = getOption("pager"))

?topic

type?topic
```

Arguments

`topic`	usually, a name or character string specifying the topic for which help is sought. A character string (enclosed in explicit single or double quotes) is always taken as naming a topic. For `help`, if the value of `topic` is a length-one character vector the topic is taken to be the value of the only element. Otherwise `topic` must be a name or a reserved word (if syntactically valid) or character string.

For ? the topic argument may also be a function call, to ask for documentation on a corresponding method: see the section on S4 method documentation. Unlike help, all names are deparsed to give the desired topic. The forms pkg::topic and pkg:::topic are treated specially, and look for help on topic in package **package**.

See 'Details' for what happens if this is omitted for help.

offline — a logical indicating whether documentation should be displayed on-line to the screen (the default) or hardcopy of it should be produced.

package — a name or character vector giving the packages to look into for documentation, or NULL. By default, all packages in the search path are used.

lib.loc — a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.

verbose — logical; if TRUE, the file name is reported.

try.all.packages — logical; see Note.

chmhelp — logical (or NULL). Only relevant under Windows. If TRUE the Compiled HTML version of the help on the topic will be shown in a help viewer. If none is available, plain text help will be shown.

htmlhelp — logical (or NULL). If TRUE (which is the default after help.start has been called), the HTML version (if available) of the help on the topic will be shown in the browser specified by options("browser"). See browseURL for details of the browsers that are supported. Where possible an existing browser window is re-used.

pager — the pager to be used for file.show.

type — the special type of documentation to use for this topic; for example, if the type is class, documentation is provided for the class with name topic. The function topicName returns the actual name used in this case. See the section on method documentation for the uses of type to get help on formal methods.

Details

topic is not optional: if it is omitted R will give a hint as to suitable topics if a package is specified, to available packages if lib.loc only is specified, and help on help itself if nothing is specified. (In all cases this will be text help.)

Some topics need to be quoted (by backticks) or given as a character string. There include those which cannot syntactically appear on their own such as unary and binary operators, function and control-flow reserved words (including if, else for, in, repeat, while, break and next. The other reserved words can be used as if they were names, for example TRUE, NA and Inf.

If multiple help files matching topic are found, in interactive use a menu is presented for the user to choose one: otherwise the first on the search path is used. (The menu will be a graphical menu if possible if getOption("menu.graphics") is true, the default.)

HTML help works best if help.start() has been called in the session. Otherwise there will be a warning about 'Using non-linked HTML file: hyperlinks may be incorrect' and cross-library links will most likely not be resolved.

If offline is TRUE, hardcopy of the documentation is produced by running the LaTeX version of the help page through latex and dvips. Depending on your dvips configuration, hardcopy will be sent to the printer or saved in a file. If the programs are in non-standard locations and hence were not found at compile time, you can either set the options latexcmd and dvipscmd,

or the environment variables R_LATEXCMD and R_DVIPSCMD appropriately. The appearance of the output can be customized through a file 'Rhelp.cfg' somewhere in your LaTeX search path: this will be input as a LaTeX style file after Rd.sty. Some environment variables are consulted, notably R_PAPERSIZE (*via* getOption("papersize")) and R_RD4DVI

If LaTeX versions of help pages were not built at the installation of the package, the print method will ask if conversion with R CMD Rdconv (which requires Perl) should be attempted.

S4 Method Documentation

The authors of formal ('S4') methods can provide documentation on specific methods, as well as overall documentation on the methods of a particular function. The "?" operator allows access to this documentation in three ways.

The expression methods ? f will look for the overall documentation methods for the function f. Currently, this means the documentation file containing the alias f-methods.

There are two different ways to look for documentation on a particular method. The first is to supply the topic argument in the form of a function call, omitting the type argument. The effect is to look for documentation on the method that would be used if this function call were actually evaluated. See the examples below. If the function is not a generic (no S4 methods are defined for it), the help reverts to documentation on the function name.

The "?" operator can also be called with type supplied as "method"; in this case also, the topic argument is a function call, but the arguments are now interpreted as specifying the class of the argument, not the actual expression that will appear in a real call to the function. See the examples below.

The first approach will be tedious if the actual call involves complicated expressions, and may be slow if the arguments take a long time to evaluate. The second approach avoids these difficulties, but you do have to know what the classes of the actual arguments will be when they are evaluated.

Both approaches make use of any inherited methods; the signature of the method to be looked up is found by using selectMethod (see the documentation for getMethod).

Note

Unless lib.loc is specified explicitly, the loaded packages are searched before those in the specified libraries. This ensures that if a library is loaded from a library not in the known library trees, then the help from the loaded library is used. If lib.loc is specified explicitly, the loaded packages are *not* searched.

If this search fails and argument try.all.packages is TRUE and neither packages nor lib.loc is specified, then all the packages in the known library trees are searched for help on topic and a list of (any) packages where help may be found is printed (but no help is shown). **N.B.** searching all packages can be slow.

The help files can be many small files. On some file systems it is desirable to save space, and the text files in the 'help' directory of an installed package can be zipped up as a zip archive 'Rhelp.zip'. Ensure that file 'AnIndex' remains un-zipped. Similarly, all the files in the 'latex' directory can be zipped to 'Rhelp.zip'.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`help.search()` or `??` for finding help pages on a vague topic; `help.start()` which opens the HTML version of the R help pages; `library()` for listing available packages and the user-level objects they contain; `data()` for listing available data sets; `methods()`.

See `prompt()` to get a prototype for writing `help` pages of private packages.

Examples

```
help()
help(help)               # the same

help(lapply)
?lapply                  # the same

help("for")              # or ?"for", but quotes/backticks are needed
?`+`

help(package="splines") # get help even when package is not loaded

data()                   # list all available data sets
?women                   # information about data set "women"

topi <- "women"
help(topi)

try(help("bs", try.all.packages=FALSE)) # reports not found (an error)
help("bs", try.all.packages=TRUE)       # reports can be found
                                        # in package 'splines'

## Not run:
require(methods)
## define a S4 generic function and some methods
combo <- function(x, y) c(x, y)
setGeneric("combo")
setMethod("combo", c("numeric", "numeric"), function(x, y) x+y)

## assume we have written some documentation
## for combo, and its methods ....

?combo  ## produces the function documentation

methods?combo  ## looks for the overall methods documentation

method?combo("numeric", "numeric")  ## documentation for the method above

?combo(1:10, rnorm(10))  ## ... the same method, selected according to
                         ## the arguments (one integer, the other numeric)

?combo(1:10, letters)    ## documentation for the default method
## End(Not run)
```

`help.request` *Send a Post to R-help*

Description

Prompts the user to check they have done all that is expected of them before sending a post to the R-help mailing list, provides a template for the post with session information included and optionally sends the email (on Unix systems).

Usage

```
help.request(subject = "",
             ccaddress = Sys.getenv("USER"),
             method = getOption("mailer"),
             address = "r-help@R-project.org",
             file = "R.help.request")
```

Arguments

`subject`	subject of the email. Please do not use single quotes (') in the subject! Post separate help requests for multiple queries.
`ccaddress`	optional email address for copies (default is current user). Use `ccaddress = FALSE` for no copies.
`method`	submission method: for Unix one of `"mailx"`, `"gnudoit"`, `"none"` or `"ess"`; for Windows either `"none"` (default) or `"mailto"`.
`address`	recipient's email address.
`file`	file to use for setting up the email (or storing it when method is `"none"` or sending mail fails).

Details

This function is not intended to replace the posting guide. Please read the guide before posting to R-help or using this function (see http://www.r-project.org/posting-guide.html).

The `help.request` function:

- asks whether the user has consulted relevant resources, stopping and opening the relevant url if a negative response if given.
- checks whether the current version of R is being used and whether the add-on packages are up-to-date, giving the option of updating where necessary.
- asks whether the user has prepared appropriate (minimal, reproducible, self-contained, commented) example code ready to paste into the post.

Once this checklist has been completed a template post is prepared including current session information.

If method is `"none"` or `NULL`, then the default text editor is opened for the user to complete the post. Which editor is used can be controlled using `options`, type `getOption("editor")` to see what editor is currently defined. Please use the help pages of the respective editor for details of usage. The report can then be copied to your favorite email program and sent to the r-help list.

On Windows systems there is an experimental `"mailto"` option, which sends the template post to the system's default email program for the user to edit and send.

On Unix systems there are three options for direct submission of the post. If the submission method is `"mailx"`, then the default editor is used to write the help request. After saving the help request (in the temporary file opened) and exiting the editor the report is mailed using a Unix command line mail utility such as `mailx`. A copy of the mail is sent to the current user. If method is `"gnudoit"`,

then an emacs mail buffer is opened and used for sending the email. If method is "ess" the body of the mail is simply sent to stdout.

Value

Nothing useful.

Author(s)

Heather Turner, based on code and help page of bug.report().

See Also

The posting guide (http://www.r-project.org/posting-guide.html), also sessionInfo() from which you may add to the help request.

---

help.search *Search the Help System*

Description

Allows for searching the help system for documentation matching a given character string in the (file) name, alias, title, concept or keyword entries (or any combination thereof), using either fuzzy matching or regular expression matching. Names and titles of the matched help entries are displayed nicely formatted.

Usage

```
help.search(pattern, fields = c("alias", "concept", "title"),
            apropos, keyword, whatis, ignore.case = TRUE,
            package = NULL, lib.loc = NULL,
            help.db = getOption("help.db"),
            verbose = getOption("verbose"),
            rebuild = FALSE, agrep = NULL)
??pattern
field??pattern
```

Arguments

pattern	a character string to be matched in the specified fields. If this is given, the arguments apropos, keyword, and whatis are ignored.
fields	a character vector specifying the fields of the help data bases to be searched. The entries must be abbreviations of "name", "title", "alias", "concept", and "keyword", corresponding to the help page's (file) name, its title, the topics and concepts it provides documentation for, and the keywords it can be classified to.
apropos	a character string to be matched in the help page topics and title.
keyword	a character string to be matched in the help page 'keywords'. 'Keywords' are really categories: the standard categories are listed in file 'RHOME/doc/KEYWORDS' (see also the example) and some package writers have defined their own. If keyword is specified, agrep defaults to FALSE.

whatis	a character string to be matched in the help page topics.
ignore.case	a logical. If TRUE, case is ignored during matching; if FALSE, pattern matching is case sensitive.
package	a character vector with the names of packages to search through, or NULL in which case *all* available packages in the library trees specified by lib.loc are searched.
lib.loc	a character vector describing the location of R library trees to search through, or NULL. The default value of NULL corresponds to all libraries currently known.
help.db	a character string giving the file path to a previously built and saved help data base, or NULL.
verbose	logical; if TRUE, the search process is traced.
rebuild	a logical indicating whether the help data base should be rebuilt. This will be done automatically if lib.loc or the search path is changed, or if package is used and a value is not found.
agrep	if NULL (the default unless keyword is used) and the character string to be matched consists of alphanumeric characters, whitespace or a dash only, approximate (fuzzy) matching via agrep is used unless the string has fewer than 5 characters; otherwise, it is taken to contain a regular expression to be matched via grep. If FALSE, approximate matching is not used. Otherwise, one can give a numeric or a list specifying the maximal distance for the approximate match, see argument max.distance in the documentation for agrep.
field	a single value of fields to search.

Details

Upon installation of a package, a pre-built help.search index is serialized as 'hsearch.rds' in the 'Meta' directory (provided the package has any help pages). These files are used to create the data base.

The arguments apropos and whatis play a role similar to the Unix commands with the same names.

Searching with agrep = FALSE will be several times faster than the default.

If possible, the help data base is saved in memory or (if memory limits have been set: see mem.limits) to a file in the session temporary directory for use by subsequent calls in the session.

Note that currently the aliases in the matching help files are not displayed.

Matching by agrep is by bytes even in multi-byte locales: character-by-character matching will be too slow with a thousand or more packages.

As with ?, in ?? the pattern may be prefixed with a package name followed by :: or ::: to limit the search to that package.

Value

The results are returned in a list object of class "hsearch", which has a print method for nicely formatting the results of the query. This mechanism is experimental, and may change in future versions of R.

In R.app on Mac OS X, this will show up a browser with selectable items. On exiting this browser, the help pages for the selected items will be shown in separate help windows.

The internal format of the class is undocumented and subject to change.

See Also

help; help.start for starting the hypertext (currently HTML) version of R's online documentation, which offers a similar search mechanism.

RSiteSearch to access an on-line search of R resources.

apropos uses regexps and has nice examples.

Examples

```
help.search("linear models")    # In case you forgot how to fit linear
                                # models
help.search("non-existent topic")

??utils::help  # All the topics matching "help" in the utils package

## Not run:
help.search("print")            # All help pages with topics or title
                                # matching 'print'
help.search(apropos = "print")  # The same

help.search(keyword = "hplot")  # All help pages documenting high-level
                                # plots.
file.show(file.path(R.home(), "doc", "KEYWORDS"))  # show all keywords

## Help pages with documented topics starting with 'try'.
help.search("\\btry", fields = "alias")
## End(Not run)
```

---

help.start *Hypertext Documentation*

Description

Start the hypertext (currently HTML) version of R's online documentation.

Usage

```
help.start(gui = "irrelevant", browser = getOption("browser"),
           remote = NULL, searchEngine = FALSE)
```

Arguments

gui	just for compatibility with S-PLUS.
browser	the name of the program to be used as hypertext browser. It should be in the PATH, or a full path specified. Alternatively, it can be a function which will be called with an URL as its only argument.
remote	A character giving a valid URL for the '*R_HOME*' directory on a remote location.
searchEngine	logical: if true go directly to the search engine page. A workaround for Firefox 3 users (see 'Notes').

Details

All the packages in the known library trees are linked to directory '.R' in the per-session temporary directory. The links are re-made each time `make.packages.html` or `help.start` is run, which should be done after packages are installed, updated or removed.

A side effect is to set `options(htmlhelp = TRUE)` so that future help requests (by default) are sent to the browser.

If the browser named by the `browser` argument is different from the default browser as specified by `options("browser")`, the default is changed to the given browser so that it gets used for all future help requests. (This only holds when `browser` is a name, not a function.)

Note

There is a Java-based search facility available from the HTML page that `help.start` brings up. Should this not work, please consult the 'R Installation and Administration' manual which is linked from that page.

Note to users of Firefox 3: the search results have links that are resolved incorrectly by that browser if starting from the normal HTML index page. The workaround is to start with `help.start(searchEngine = TRUE)`

See Also

`help()` for on- and off-line help in ASCII/Editor or PostScript format.

`browseURL` for how the help file is displayed.

`RSiteSearch` to access an on-line search of R resources.

Examples

```
## Not run:
help.start()
## End(Not run)
```

---

index.search *Search Indices for Help Files*

Description

Used to search the indices for help files, possibly under aliases.

Usage

```
index.search(topic, path, file="AnIndex", type = "help")
```

Arguments

`topic`	The keyword to be searched for in the indices.
`path`	The path(s) to the packages to be searched.
`file`	The index file to be searched. Normally '"AnIndex"'.
`type`	The type of file required.

Details

For each package in `path`, examine the file `file` in directory '`type`', and look up the matching file stem for topic `topic`, if any.

Value

A character vector of matching files, as if they are in directory `type` of the corresponding package. In the special cases of `type = "html"`, `"R-ex"` and `"latex"` the file extensions `".html"`, `".R"` and `".tex"` are added.

See Also

`help`, `example`

INSTALL *Install Add-on Packages*

Description

Utility for installing add-on packages.

Usage

```
R CMD INSTALL [options] [-l lib] pkgs
```

Arguments

`pkgs`	A space-separated list with the path names of the packages to be installed.
`lib`	the path name of the R library tree to install to.
`options`	a space-separated list of options through which in particular the process for building the help files can be controlled. Options should only be given once. Use `R CMD INSTALL --help` for the current list of options.

Details

This will stop at the first error, so if you want all the `pkgs` to be tried, call this via a shell `for` or `foreach` loop.

If used as `R CMD INSTALL pkgs` without explicitly specifying `lib`, packages are installed into the library tree rooted at the first directory in the library path which would be used by R run in the current environment.

To install into the library tree `lib`, use `R CMD INSTALL -l lib pkgs`. This prepends `lib` to `R_LIBS` for duration of the install, so required packages in the installation directory will be found (and used in preference to those in other libraries).

Both `lib` and the elements of `pkgs` may be absolute or relative path names of directories. `pkgs` may also contain names of package/bundle archive files of the form '`pkg_version.tar.gz`' as obtained from CRAN: these are then extracted in a temporary directory. Finally, binary package/bundle archive files (as created by `R CMD build --binary` can be supplied.

Some package sources contain a '`configure`' script that can be passed arguments or variables via the option '`--configure-args`' and '`--configure-vars`', respectively, if necessary. The

latter is useful in particular if libraries or header files needed for the package are in non-system directories. In this case, one can use the configure variables `LIBS` and `CPPFLAGS` to specify these locations (and set these via '`--configure-vars`'), see section "Configuration variables" in "R Installation and Administration" for more information. (If these are used more than once on the command line, only the last instance is used.) One can bypass the configure mechanism using the option '`--no-configure`'.

If '`--no-docs`' is given, no help files are built. Options '`--no-text`', '`--no-html`', and '`--no-latex`' suppress creating the text, HTML, and LaTeX versions, respectively. The default is to build help files in all three versions.

If the attempt to install the package fails, leftovers are removed. If the package was already installed, the old version is restored. This happens either if a command encounters an error or if the install is interrupted from the keyboard: after cleaning up the script terminates.

Use `R CMD INSTALL --help` for more usage information.

Packages using the methods package

Packages that require the methods package and make use functions such as `setMethod` or `setClass`, should be installed using lazy-loading (or, deprecated, by creating a binary image): use the field `LazyLoad` in the 'DESCRIPTION' file to ensure this.

Note

Some parts of the operation of `INSTALL` depend on '/tmp' having both write and execution access to the account running R. This is usually the case, but if '/tmp' has been mounted as `noexec`, environment variable `TMPDIR` needs to be set to a directory from which execution is allowed.

See Also

`REMOVE` and `library` for information on using several library trees; `update.packages` for automatic update of packages using the internet (or other R level installation of packages, such as by `install.packages`).

The section on "Add-on packages" in "R Installation and Administration" and the chapter on "Creating R packages" in "Writing R Extensions" (see `RShowDoc` and the 'doc/manual' subdirectory of the R source tree).

`installed.packages` *Find Installed Packages*

Description

Find (or retrieve) details of all packages installed in the specified libraries.

Usage

```
installed.packages(lib.loc = NULL, priority = NULL,
                   noCache = FALSE, fields = NULL)
```

Arguments

`lib.loc`	character vector describing the location of R library trees to search through.
`priority`	character vector or `NULL` (default). If non-null, used to select packages; `"high"` is equivalent to `c("base", "recommended")`. To select all packages without an assigned priority use `priority = "NA"`.
`noCache`	Do not use cached information.
`fields`	a character vector giving the fields to extract from each package's `DESCRIPTION` file in addition to the default ones, or `NULL` (default). Unavailable fields result in `NA` values.

Details

`installed.packages` scans the 'DESCRIPTION' files of each package found along `lib.loc` and returns a matrix of package names, library paths and version numbers.

Note: this works with package names, not bundle names, and for versioned installs with the name under which the package is installed, in the style `mypkg_1.3-7`.

The information found is cached (by library) for the R session and specified `fields` argument, and updated only if the top-level library directory has been altered, for example by installing or removing a package. If the cached information becomes confused, it can be refreshed by running `installed.packages(noCache = TRUE)`.

Value

A matrix with one row per package, row names the package names and column names `"Package"`, `"LibPath"`, `"Version"`, `"Priority"`, `"Bundle"`, `"Contains"`, `"Depends"`, `"Suggests"`, `"Imports"` and `"Built"` (the R version the package was built under). Additional columns can be specified using the `fields` argument.

See Also

`update.packages`, `INSTALL`, `REMOVE`.

Examples

```
str(ip <- installed.packages(priority = "high"))
ip[, c(1,3:5)]
plic <- installed.packages(priority = "high", fields="License")
## what licenses are there:
table( plic[,"License"] )
```

LINK *Create Executable Programs*

Description

Front-end for creating executable programs.

Usage

```
R CMD LINK [options] linkcmd
```

Arguments

linkcmd	a list of commands to link together suitable object files (include library objects) to create the executable program.
options	further options to control the linking, or for obtaining information about usage and version.

Details

The linker front-end is useful in particular when linking against the R shared library, in which case linkcmd must contain -lR but need not specify its library path.

Currently only works if the C compiler is used for linking, and no C++ code is used.

Use R CMD LINK --help for more usage information.

Note

Some binary distributions of R have LINK in a separate bundle, e.g. an R-devel RPM.

---

localeToCharset *Select a Suitable Encoding Name from a Locale Name*

Description

This functions aims to find a suitable coding for the locale named, by default the current locale, and if it is a UTF-8 locale a suitable single-byte encoding.

Usage

```
localeToCharset(locale = Sys.getlocale("LC_CTYPE"))
```

Arguments

locale	character string naming a locale.

Details

The operation differs by OS. Locale names are normaly like es_MX.iso88591. If final component indicates an encoding and it is not utf8 we just need to look up the equivalent encoding name. Otherwise, the language (here es) is used to choose a primary or fallback encoding.

In the C locale the answer will be "ASCII".

Value

A character vector naming an encoding and possibly a fallback single-encoding, NA if unknown.

Note

The encoding names are those used by libiconv, and ought also to work with glibc but maybe not with commercial Unixen.

See Also

Sys.getlocale, iconv.

Examples

```
localeToCharset()
```

`ls.str` *List Objects and their Structure*

Description

`ls.str` and `lsf.str` are variations of `ls` applying `str()` to each matched name: see section Value.

Usage

```
ls.str(pos = -1, name, envir, all.names = FALSE,
       pattern, mode = "any")

lsf.str(pos = -1, envir, ...)

## S3 method for class 'ls_str':
print(x, max.level = 1, give.attr = FALSE, ...)
```

Arguments

`pos`	integer indicating `search` path position.
`name`	optional name indicating `search` path position, see `ls`.
`envir`	environment to use, see `ls`.
`all.names`	logical indicating if names which begin with a `.` are omitted; see `ls`.
`pattern`	a regular expression passed to `ls`. Only names matching `pattern` are considered.
`max.level`	maximal level of nesting which is applied for displaying nested structures, e.g., a list containing sub lists. Default 0: Display all nesting levels.
`give.attr`	logical; if `TRUE` (default), show attributes as sub structures.
`mode`	character specifying the `mode` of objects to consider. Passed to `exists` and `get`.
`x`	an object of class `"ls_str"`.
`...`	further arguments to pass. and `lsf.str` passes them to `ls.str` which passes them on to `ls`. The (non-exported) print method `print.ls_str` passes them to `str`.

Value

`ls.str` and `lsf.str` return an object of class `"ls_str"`, basically the character vector of matching names (functions only for `lsf.str`), similarly to `ls`, with a `print()` method that calls `str()` on each object.

Author(s)

Martin Maechler

See Also

`str`, `summary`, `args`.

Examples

```
require(stats)

lsf.str()#- how do the functions look like which I am using?
ls.str(mode = "list") #- what are the structured objects I have defined?

## create a few objects
example(glm, echo = FALSE)
ll <- as.list(LETTERS)
print(ls.str(), max.level = 0)# don't show details

## which base functions have "file" in their name ?
lsf.str(pos = length(search()), pattern = "file")

## demonstrating that  ls.str() works inside functions
## ["browser/debug mode"]:
tt <- function(x, y=1) { aa <- 7; r <- x + y; ls.str() }
(nms <- sapply(strsplit(capture.output(tt(2))," *: *"), `[`, 1))
stopifnot(nms == c("aa", "r","x","y"))
```

`make.packages.html` *Update HTML Documentation Files*

Description

Functions to link and (optionally) re-create the HTML documentation files to reflect all installed packages.

Usage

```
make.packages.html(lib.loc = .libPaths(), packages = TRUE)
```

Arguments

`lib.loc`	character vector. List of libraries to be included.
`packages`	logical: should be package indices be create.

Details

This sets up the links from packages in libraries to the '.R' subdirectory of the per-session directory (see `tempdir`) and optionally creates the 'packages.html' and 'index.txt' files to point to those links. The slow part is creating those files (as all the package 'DESCRIPTION' files are read).

If a package is available in more than one library tree, all the copies are linked, after the first with suffix `.1` etc.

Value

Invisible logical, indicating if the files were created.

See Also

help.start

Examples

```
## Not run:
# to prefer HTML help, put in your .Rprofile
options(htmlhelp=TRUE)
make.packages.html(packages=FALSE)
# this can be slow for large numbers of installed packages.
## End(Not run)
```

---

make.socket *Create a Socket Connection*

Description

With `server = FALSE` attempts to open a client socket to the specified port and host. With `server = TRUE` listens on the specified port for a connection and then returns a server socket. It is a good idea to use `on.exit` to ensure that a socket is closed, as you only get 64 of them.

Usage

```
make.socket(host = "localhost", port, fail = TRUE, server = FALSE)
```

Arguments

`host`	name of remote host
`port`	port to connect to/listen on
`fail`	failure to connect is an error?
`server`	a server socket?

Value

An object of class `"socket"`.

`socket`	socket number. This is for internal use
`port`	port number of the connection
`host`	name of remote computer

Warning

I don't know if the connecting host name returned when `server = TRUE` can be trusted. I suspect not.

Author(s)

Thomas Lumley

References

Adapted from Luke Tierney's code for `XLISP-Stat`, in turn based on code from Robbins and Robbins "Practical UNIX Programming"

See Also

`close.socket`, `read.socket`

Examples

```
daytime <- function(host = "localhost"){
    a <- make.socket(host, 13)
    on.exit(close.socket(a))
    read.socket(a)
}
## Official time (UTC) from US Naval Observatory
## Not run: daytime("tick.usno.navy.mil")
```

`memory.size` *Report on Memory Allocation*

Description

`memory.size` and `memory.limit` are used to manage the total memory allocation on Windows. On other platforms these are stubs which report infinity with a warning.

Usage

```
memory.size(max = FALSE)

memory.limit(size = NA)
```

Arguments

`max`	logical. If true the maximum amount of memory obtained from the OS is reported, otherwise the amount currently in use.
`size`	numeric. If `NA` report the memory size, otherwise request a new limit, in Mb.

Details

To restrict memory usage on a Unix-alike use the facilities of the shell used to launch R, e.g. `limit` or `ulimit`.

Value

Size in bytes: always `Inf`.

See Also

Memory-limits for other limits.

menu *Menu Interaction Function*

Description

menu presents the user with a menu of choices labelled from 1 to the number of choices. To exit without choosing an item one can select '0'.

Usage

```
menu(choices, graphics = FALSE, title = "")
```

Arguments

choices	a character vector of choices
graphics	a logical indicating whether a graphics menu should be used if available.
title	a character string to be used as the title of the menu. NULL is also accepted.

Details

If graphics = TRUE and a windowing system is available (Windows, MacOS X or X11 *via* Tcl/Tk) a listbox widget is used, otherwise a text menu.

Ten or fewer items will be displayed in a single column, more in multiple columns if possible within the current display width.

No title is displayed if title is NULL or "".

Value

The number corresponding to the selected item, or 0 if no choice was made.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

select.list, which is used to implement the graphical menu, and allows multiple selections.

Examples

```
## Not run:
switch(menu(c("List letters", "List LETTERS")) + 1,
       cat("Nothing done\n"), letters, LETTERS)
## End(Not run)
```

`methods` *List Methods for S3 Generic Functions or Classes*

Description

List all available methods for an S3 generic function, or all methods for a class.

Usage

```
methods(generic.function, class)
```

Arguments

`generic.function`	a generic function, or a character string naming a generic function.
`class`	a symbol or character string naming a class: only used if `generic.function` is not supplied.

Details

Function `methods` can be used to find out about the methods for a particular generic function or class. The functions listed are those which *are named like methods* and may not actually be methods (known exceptions are discarded in the code). Note that the listed methods may not be user-visible objects, but often help will be available for them.

If `class` is used, we check that a matching generic can be found for each user-visible object named. If `generic.function` is given, there is a warning if it appears not to be a generic function. (The check for being generic used can be fooled.)

Value

An object of class `"MethodsFunction"`, a character vector of function names with an `"info"` attribute. There is a `print` method which marks with an asterisk any methods which are not visible: such functions can be examined by `getS3method` or `getAnywhere`.

The `"info"` attribute is a data frame, currently with a logical column, `visible` and a factor column `from` (indicating where the methods were found).

Note

This scheme is called *S3* (S version 3). For new projects, it is recommended to use the more flexible and robust *S4* scheme provided in the **methods** package. Functions can have both S3 and S4 methods, and function `showMethods` will list the S4 methods (possibly none).

The original `methods` function was written by Martin Maechler.

References

Chambers, J. M. (1992) *Classes and methods: object-oriented programming in S.* Appendix A of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

S3Methods, class, getS3method.

For S4, showMethods, Methods.

Examples

```
require(stats)

methods(summary)
methods(class = "aov")
methods("[[")    # uses C-internal dispatching
methods("$")
methods("$<-")   # replacement function
methods("+")     # binary operator
methods("Math")  # group generic
require(graphics)
methods("axis")  # looks like it has methods, but not generic
## Not run:
methods(print)   # over 100
## End(Not run)
## --> help(showMethods) for related examples
```

---

mirrorAdmin *Managing Repository Mirrors*

Description

Functions helping to maintain CRAN, some of them may also be useful for administrators of other repository networks.

Usage

```
mirror2html(mirrors = NULL, file = "mirrors.html",
  head = "mirrors-head.html", foot = "mirrors-foot.html")
checkCRAN(method)
```

Arguments

mirrors	A data frame, by default the CRAN list of mirrors is used.
file	A connection object or a character string.
head	Name of optional header file.
foot	Name of optional footer file.
method	Download method, see download.file.

Details

mirror2html creates the HTML file for the CRAN list of mirrors and invisibly returns the HTML text.

checkCRAN performs a sanity checks on all CRAN mirrors.

modifyList *Recursively Modify Elements of a List*

Description

Modifies a possibly nested list recursively by changing a subset of elements at each level to match a second list.

Usage

```
modifyList(x, val)
```

Arguments

x	a named list, possibly empty.
val	a named list with components to replace corresponding components in x.

Value

A modified version of x, with the modifications determined as follows (here, list elements are identified by their names). Elements in val which are missing from x are added to x. For elements that are common to both but are not both lists themselves, the component in x is replaced by the one in val. For common elements that are both lists, x[[name]] is replaced by modifyList(x[[name]], val[[name]]).

Author(s)

Deepayan Sarkar ⟨Deepayan.Sarkar@R-project.org⟩

Examples

```
foo <- list(a = 1, b = list(c = "a", d = FALSE))
bar <- modifyList(foo, list(e = 2, b = list(d = TRUE)))
str(foo)
str(bar)
```

normalizePath *Express File Paths in Canonical Form*

Description

Convert file paths to canonical form for the platform, to display them in a user-understandable form.

Usage

```
normalizePath(path)
```

Arguments

path	character vector of file paths.

Details

Where the platform supports it this turns paths into absolute paths in their canonical form (no `./`, `../` nor symbolic links).

If the path is not a real path the result is undefined. On Unix-alikes, this will likely be the corresponding input element. On Windows, it will likely result in an error being signalled.

Value

A character vector.

Examples

```
cat(normalizePath(c(R.home(), tempdir())), sep = "\n")
```

---

nsl *Look up the IP Address by Hostname*

Description

Interface to `gethostbyname`.

Usage

```
nsl(hostname)
```

Arguments

`hostname`	the name of the host.

Value

The IP address, as a character string, or `NULL` if the call fails.

Note

This was included as a test of internet connectivity, to fail if the node running R is not connected. It will also return `NULL` if BSD networking is not supported, including the header file 'arpa/inet.h'.

Examples

```
## Not run: nsl("www.r-project.org")
```

`object.size` *Report the Space Allocated for an Object*

Description

Provides an estimate of the memory that is being used to store an R object.

Usage

```
object.size(x)
```

Arguments

`x`	An R object.

Details

Exactly which parts of the memory allocation should be attributed to which object is not clear-cut. This function merely provides a rough indication: it should be reasonably accurate for atomic vectors, but does not detect if elements of a list are shared, for example. (Sharing amongst elements of a character vector is taken into account, but not that between character vectors in a single object.)

The calculation is of the size of the object, and excludes the space needed to store its name in the symbol table.

Associated space (e.g. the environment of a function and what the pointer in a `EXTPTRSXP` points to) is not included in the calculation.

Object sizes are larger on 64-bit platforms than 32-bit ones, but will very likely be the same on different platforms with the same word length and pointer size.

Value

An estimate of the memory allocation attributable to the object, in bytes.

See Also

`Memory-limits` for the design limitations on object size.

Examples

```
object.size(letters)
object.size(ls)
## find the 10 largest objects in the base package
z <- sapply(ls("package:base"), function(x)
            object.size(get(x, envir = baseenv())))
as.matrix(rev(sort(z))[1:10])
```

`package.skeleton` *Create a Skeleton for a New Source Package*

Description

`package.skeleton` automates some of the setup for a new source package. It creates directories, saves functions, data, and R code files to appropriate places, and creates skeleton help files and a 'Read-and-delete-me' file describing further steps in packaging.

Usage

```
package.skeleton(name = "anRpackage", list,
                 environment = .GlobalEnv,
                 path = ".", force = FALSE, namespace = FALSE,
                 code_files = character())
```

Arguments

`name`	character string: the package name and directory name for your package.
`list`	character vector naming the R objects to put in the package. Usually, at most one of `list`, `environment`, or `code_files` will be supplied. See 'Details'.
`environment`	an environment where objects are looked for. See 'Details'.
`path`	path to put the package directory in.
`force`	If `FALSE` will not overwrite an existing directory.
`namespace`	a logical indicating whether to add a name space for the package. If `TRUE`, a `NAMESPACE` file is created to export all objects whose names begin with a letter, plus all S4 methods and classes.
`code_files`	a character vector with the paths to R code files to build the package around. See 'Details'.

Details

The arguments `list`, `environment`, and `code_files` provide alternative ways to initialize the package. If `code_files` is supplied, the files so named will be sourced to form the environment, then used to generate the package skeleton. Otherwise `list` defaults to the non-hidden files in `environment` (those whose name does not start with `.`), but can be supplied to select a subset of the objects in that environment.

Stubs of help files are generated for functions, data objects, and S4 classes and methods, using the `prompt`, `promptClass`, and `promptMethods` functions.

The package sources are placed in subdirectory `name` of `path`. If `code_files` is supplied, these files are copied; otherwise, objects will be dumped into individual source files. The file names in `code_files` should have suffix `".R"` and be in the current working directory.

The filenames created for source and documentation try to be valid for all OSes known to run R. Invalid characters are replaced by '_', invalid names are preceded by 'zz', and finally the converted names are made unique by `make.unique(sep = "_")`. This can be done for code and help files but not data files (which are looked for by name). Also, the code and help files should have names starting with an ASCII letter or digit, and this is checked and if necessary `z` prepended.

When you are done, delete the 'Read-and-delete-me' file, as it should not be distributed.

Value

Used for its side-effects.

References

Read the *Writing R Extensions* manual for more details.

Once you have created a *source* package you need to install it: see the *R Installation and Administration* manual, `INSTALL` and `install.packages`.

See Also

`prompt`, `promptClass`, and `promptMethods`.

Examples

```
require(stats)
## two functions and two "data sets" :
f <- function(x,y) x+y
g <- function(x,y) x-y
d <- data.frame(a=1, b=2)
e <- rnorm(1000)

package.skeleton(list=c("f","g","d","e"), name="mypkg")
```

`packageDescription` *Package Description*

Description

Parses and returns the 'DESCRIPTION' file of a package.

Usage

```
packageDescription(pkg, lib.loc = NULL, fields = NULL,
                   drop = TRUE, encoding = "")
```

Arguments

`pkg`	a character string with the package name.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
`fields`	a character vector giving the tags of fields to return (if other fields occur in the file they are ignored).
`drop`	If `TRUE` and the length of `fields` is 1, then a single character string with the value of the respective field is returned instead of an object of class `"packageDescription"`.
`encoding`	If there is an `Encoding` field, to what encoding should re-encoding be attempted? If `NA`, no re-encoding. The other values are as used by `iconv`, so the default `""` indicates the encoding of the current locale.

Details

A package will not be 'found' unless it has a 'DESCRIPTION' file which contains a valid `Version` field. Different warnings are given when no package directory is found and when there is a suitable directory but no valid 'DESCRIPTION' file.

An attached environment named to look like a package (e.g. `package:utils2`) will be ignored as from R 2.6.0.

Value

If a 'DESCRIPTION' file for the given package is found and can successfully be read, `packageDescription` returns an object of class `"packageDescription"`, which is a named list with the values of the (given) fields as elements and the tags as names, unless `drop = TRUE`.

If parsing the 'DESCRIPTION' file was not successful, it returns a named list of `NA`s with the field tags as names if `fields` is not null, and `NA` otherwise.

See Also

`read.dcf`

Examples

```
packageDescription("stats")
packageDescription("stats", fields = c("Package", "Version"))

packageDescription("stats", fields = "Version")
packageDescription("stats", fields = "Version", drop = FALSE)
```

---

packageStatus *Package Management Tools*

Description

Summarize information about installed packages and packages available at various repositories, and automatically upgrade outdated packages.

Usage

```
packageStatus(lib.loc = NULL, repositories = NULL, method,
              type = getOption("pkgType"))

## S3 method for class 'packageStatus':
summary(object, ...)

## S3 method for class 'packageStatus':
update(object, lib.loc = levels(object$inst$LibPath),
       repositories = levels(object$avail$Repository), ...)

## S3 method for class 'packageStatus':
upgrade(object, ask = TRUE, ...)
```

Arguments

lib.loc	a character vector describing the location of R library trees to search through, or NULL. The default value of NULL corresponds to all libraries currently known.
repositories	a character vector of URLs describing the location of R package repositories on the Internet or on the local machine.
method	Download method, see download.file.
type	type of package distribution: see install.packages.
object	an object of class "packageStatus" as returned by packageStatus.
ask	if TRUE, the user is prompted which packages should be upgraded and which not.
...	currently not used.

Details

The URLs in repositories should be full paths to the appropriate contrib sections of the repositories. The default is contrib.url(getOption("repos")).

There are print and summary methods for the "packageStatus" objects: the print method gives a brief tabular summary and the summary method prints the results.

The update method updates the "packageStatus" object. The upgrade method is similar to update.packages: it offers to install the current versions of those packages which are not currently up-to-date.

Value

An object of class "packageStatus". This is a list with two components

inst	a data frame with columns as the *matrix* returned by installed.packages plus "Status", a factor with levels c("ok", "upgrade"). Only the newest version of each package is reported, in the first repository in which it appears.
avail	a data frame with columns as the *matrix* returned by available.packages plus "Status", a factor with levels c("installed", "not installed", "unavailable")..

See Also

installed.packages, available.packages

Examples

```
## Not run:
x <- packageStatus()
print(x)
summary(x)
upgrade(x)
x <- update(x)
print(x)
## End(Not run)
```

page *Invoke a Pager on an R Object*

Description

Displays a representation of the object named by x in a pager *via* file.show.

Usage

```
page(x, method = c("dput", "print"), ...)
```

Arguments

x	An R object, or a character string naming an object.
method	The default method is to dump the object *via* dput. An alternative is to use print and capture the output to be shown in the pager.
...	additional arguments for dput, print or file.show (such as title).

Details

If x is a length-one character vector, it is used as the name of an object to look up in the environment from which page is called. All other objects are displayed directly.

A default value of title is passed to file.show if one is not supplied in ....

See Also

file.show, edit, fix.

To go to a new page when graphing, see frame.

Examples

```
## Not run:
## four ways to look at the code of 'page'
page(page)              # as an object
page("page")            # a character string
v <- "page"; page(v)    # a length-one character vector
page(utils::page)       # a call
## End(Not run)
```

person *Person Names and Contact Information*

Description

A class and utility methods for holding information about persons like name and email address.

Usage

```
person(first = "", last = "", middle = "", email = "")
personList(...)
as.person(x)
as.personList(x)

## S3 method for class 'person':
as.character(x, ...)
## S3 method for class 'personList':
as.character(x, ...)

## S3 method for class 'person':
toBibtex(object, ...)
## S3 method for class 'personList':
toBibtex(object, ...)
```

Arguments

`first`	character string, first name
`middle`	character string, middle name(s)
`last`	character string, last name
`email`	character string, email address
`...`	for `personList` an arbitrary number of `person` objects
`x`	a character string or an object of class `person` or `personList`
`object`	an object of class `person` or `personList`

Examples

```
## create a person object directly
p1 <- person("Karl", "Pearson", email = "pearson@stats.heaven")
p1

## convert a string
p2 <- as.person("Ronald Aylmer Fisher")
p2

## create one object holding both
p <- personList(p1, p2)
ps <- as.character(p)
ps
as.personList(ps)

## convert to BibTeX author field
toBibtex(p)
```

Description

Utilities for checking whether the sources of an R add-on package work correctly, and for building a source or binary package from them.

Usage

```
R CMD build [options] pkgdirs
R CMD check [options] pkgdirs
```

Arguments

`pkgdirs`	a list of names of directories with sources of R add-on packages.
`options`	further options to control the processing, or for obtaining information about usage and version of the utility.

Details

`R CMD check` checks R add-on packages from their sources, performing a wide variety of diagnostic checks.

`R CMD build` builds R source or binary packages from their sources. The name(s) of the packages are taken from the 'DESCRIPTION' files and not from the directory names.

Use `R CMD foo --help` to obtain usage information on utility `foo`.

Several of the options to `build --binary` are passed to `INSTALL` so consult its help for the details.

See Also

The sections on "Checking and building packages" and "Processing Rd format" in "Writing R Extensions" (see the 'doc/manual' subdirectory of the R source tree).

`INSTALL` is called by `build --binary`.

`prompt` *Produce Prototype of an R Documentation File*

Description

Facilitate the constructing of files documenting R objects.

Usage

```
prompt(object, filename = NULL, name = NULL, ...)

## Default S3 method:
prompt(object, filename = NULL, name = NULL,
       force.function = FALSE, ...)

## S3 method for class 'data.frame':
prompt(object, filename = NULL, name = NULL, ...)
```

Arguments

`object`	an R object, typically a function for the default method.
`filename`	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is `name` followed by `".Rd"`. Can also be `NA` (see below).
`name`	a character string specifying the name of the object.
`force.function`	a logical. If `TRUE`, treat `object` as function in any case.
`...`	further arguments passed to or from other methods.

Details

Unless `filename` is `NA`, a documentation shell for `object` is written to the file specified by `filename`, and a message about this is given. For function objects, this shell contains the proper function and argument names. R documentation files thus created still need to be edited and moved into the 'man' subdirectory of the package containing the object to be documented.

If `filename` is `NA`, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to `cat(unlist(x), file = filename, sep = "\n")`, where `x` is the list-style representation.

When `prompt` is used in `for` loops or scripts, the explicit `name` specification will be useful.

Value

If `filename` is `NA`, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

Warning

The default filename may not be a valid filename under limited file systems (e.g. those on Windows).

Currently, calling `prompt` on a non-function object assumes that the object is in fact a data set and hence documents it as such. This may change in future versions of R. Use `promptData` to create documentation skeletons for data sets.

Note

The documentation file produced by `prompt.data.frame` does not have the same format as many of the data frame documentation files in the **base** package. We are trying to settle on a preferred format for the documentation.

Author(s)

Douglas Bates for `prompt.data.frame`

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

promptData, help and the chapter on "Writing R documentation" in "Writing R Extensions" (see the 'doc/manual' subdirectory of the R source tree).

For creation of many help pages (for a package), see package.skeleton.

To prompt the user for input, see readline.

Examples

```
require(graphics)
prompt(plot.default)
prompt(interactive, force.function = TRUE)
unlink("plot.default.Rd")
unlink("interactive.Rd")

prompt(women) # data.frame
unlink("women.Rd")

prompt(sunspots) # non-data.frame data
unlink("sunspots.Rd")
```

---

promptData *Generate a Shell for Documentation of Data Sets*

Description

Generates a shell of documentation for a data set.

Usage

```
promptData(object, filename = NULL, name = NULL)
```

Arguments

object	an R object to be documented as a data set.
filename	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is name followed by ".Rd". Can also be NA (see below).
name	a character string specifying the name of the object.

Details

Unless filename is NA, a documentation shell for object is written to the file specified by filename, and a message about this is given.

If filename is NA, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to cat(unlist(x), file = filename, sep = "\n"), where x is the list-style representation.

Currently, only data frames are handled explicitly by the code.

Value

If `filename` is `NA`, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

Warning

This function is still experimental. Both interface and value might change in future versions. In particular, it may be preferable to use a character string naming the data set and optionally a specification of where to look for it instead of using `object/name` as we currently do. This would be different from `prompt`, but consistent with other prompt-style functions in package **methods**, and also allow prompting for data set documentation without explicitly having to load the data set.

See Also

`prompt`

Examples

```
promptData(sunspots)
unlink("sunspots.Rd")
```

---

promptPackage *Generate a Shell for Documentation of a Package*

Description

Generates a shell of documentation for an installed or source package.

Usage

```
promptPackage(package, lib.loc = NULL, filename = NULL,
              name = NULL, final = FALSE)
```

Arguments

`package`	the name of an *installed* or *source* package to be documented.
`lib.loc`	a character vector describing the location of R library trees to search through, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. For a source package this should specify the parent directory of the package's sources.
`filename`	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is `name` followed by `".Rd"`. Can also be `NA` (see below).
`name`	a character string specifying the name of the help topic, typically of the form `<pkg>-package`.
`final`	a logical value indicating whether to attempt to create a usable version of the help topic, rather than just a shell.

Details

Unless `filename` is `NA`, a documentation shell for `package` is written to the file specified by `filename`, and a message about this is given.

If `filename` is `NA`, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to `cat(unlist(x), file = filename, sep = "\n")`, where `x` is the list-style representation.

If `final` is `TRUE`, the generated documentation will not include the place-holder slots for manual editing, it will be usable as-is. In most cases a manually edited file is preferable (but `final = TRUE` is certainly less work).

Value

If `filename` is `NA`, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

See Also

`prompt`

Examples

```
filename <- tempfile()
promptPackage("utils", filename = filename)
file.show(filename)
unlink(filename)
```

---

rcompgen *A Completion Generator for R*

Description

This package provides a mechanism to generate relevant completions from a partially completed command line. It is not intended to be useful by itself, but rather in conjunction with other mechanisms that use it as a backend. The functions listed in the usage section provide a simple control and query mechanism. The actual interface consists of a few unexported functions described further down.

Usage

```
rc.settings(ops, ns, args, func, ipck, S3, data, help,
            argdb, files)

rc.status()

rc.getOption(name)

rc.options(...)
```

Arguments

`ops, ns, args, func, ipck, S3, data, help, argdb, files`
: logical, turning some optional completion features on and off.

  ops: activates completion after the `$` and `@` operators

  ns: controls name space related completions

  args: enables completion of function arguments

  func: enables detection of functions. If enabled, a customizable extension (`"("` by default) is appended to function names. The process of determining whether a potential completion is a function requires evaluation, including for lazy loaded symbols. This is extremely undesirable for large objects, because of potentially wasteful use of memory in addition to the time overhead associated with loading. For this reason, this feature is disabled by default.

  S3: when `args=TRUE`, activates completion on arguments of all S3 methods (otherwise just the generic, which usually has very few arguments)

  ipck: enables completion of installed package names inside `library` and `require`

  data: enables completion of data sets (including those already visible) inside `data`

  help: enables completion of help requests starting with a question mark, by looking inside help index files

  argdb: when `args=TRUE`, completion is attempted on function arguments. Generally, the list of valid arguments is determined by dynamic calls to `args`. While this gives results that are technically correct, the use of the `...` argument often hides some useful arguments. To give more flexibility in this regard, `rcompgen` internally retains an optional table of valid arguments names for specific functions. Setting `argdb=TRUE` enables preferential lookup in this internal data base for functions with an entry in it. Of course, this is useful only when the data base contains information about the function of interest. Some functions are included in the package (the maintainer is happy to add more upon request), and more can be added by the user through the unexported function `.addFunctionInfo` (see below).

  files: enables filename completion in R code. This is initially set to `FALSE`, in which case the underlying completion front-end can take over (and hopefully do a better job than we would have done). For systems where no such facilities exist, this can be set to `TRUE` if file name completion is desired. This is currently experimental and may not work very well.

  All settings are turned on by default except `ipck`, `func` and `files`. Turn more off if your CPU cycles are valuable; you will still retain basic completion on names of objects in the search list. See below for additional details.

`name, ...`
: user-settable options. Currently valid names are

  function.suffix: default `"("`

  funarg.suffix: default `" = "`

  package.suffix default `"::"`

  See `options` for detailed usage description

Details

There are several types of completion, some of which can be disabled using `rc.settings`. The most basic level, which can not be turned off once the package is loaded, provides completion on names visible on the search path, along with a few special keywords (e.g. `TRUE`). This type of completion is not attempted if the partial 'word' (a.k.a. token) being completed is empty (since there would be too many completions). The more advanced types of completion are described below.

Completion after extractors `$` and `@`: When the `ops` setting is turned on, completion after `$` and `@` is attempted. This requires the prefix to be evaluated, which is attempted unless it involves an explicit function call (implicit function calls involving the use of `[`, `$`, etc *do not* inhibit evaluation).

Completion inside name spaces: When the `ns` setting is turned on, completion inside name spaces is attempted when a token is preceded by the `::` or `:::` operators. Additionally, the basic completion mechanism is extended to include attached name spaces, or more precisely, `foopkg::` becomes a valid completion of `foo` if the return value of `search()` includes the string `"package:foopkg"`.

The completion of package name spaces applies only to attached packages, i.e. if `MASS` is not attached (whether or not it is loaded), `MAS` will not complete to `MASS::`. However, attempted completion *inside* an apparent name space will attempt to load the name space if it is not already loaded, e.g. trying to complete on `MASS::fr` will load `MASS` (but not necessarily attach it) even if it is not already loaded.

Completion of function arguments: When the `args` setting is turned on, completion on function arguments is attempted whenever deemed appropriate. The mechanism used will currently fail if the relevant function (at the point where completion is requested) was entered on a previous prompt (which implies in particular that the current line is being typed in response to a continuation prompt, usually +). Note that separation by newlines is fine.

The list of possible argument completions that is generated can be misleading. There is no problem for non-generic functions (except that `...` is listed as a completion; this is intentional as it signals the fact that the function can accept further arguments). However, for generic functions, it is practically impossible to give a reliable argument list without evaluating arguments (and not even then, in some cases), which is risky (in addition to being difficult to code, which is the real reason it hasn't even been tried), especially when that argument is itself an inline function call. Our compromise is to consider arguments of *all* currently available methods of that generic. This has two drawbacks. First, not all listed completions may be appropriate in the call currently being constructed. Second, for generics with many methods (like `print` and `plot`), many matches will need to be considered, which may take a noticeable amount of time. Despite these drawbacks, we believe this behaviour to be more useful than the only other practical alternative, which is to list arguments of the generic only.

Only S3 methods are currently supported in this fashion, and that can be turned off using the `S3` setting.

Since arguments can be unnamed in R function calls, other types of completion are also appropriate whenever argument completion is. Since there are usually many many more visible objects than formal arguments of any particular function, possible argument completions are often buried in a bunch of other possibilities. However, recall that basic completion is suppressed for blank tokens. This can be useful to list possible arguments of a function. For example, trying to complete `seq([TAB]` and `seq(from = 1, [TAB])` will both list only the arguments of `seq` (or any of its methods), whereas trying to complete `seq(length[TAB]` will list both the `length.out` argument and the `length(` function as possible completions. Note that no attempt is made to remove arguments already supplied, as that would incur a further speed penalty.

Special functions: For a few special functions (`library`, `data`, etc), the first argument is treated specially, in the sense that normal completion is suppressed, and some function specific completions are enabled if so requested by the settings. The `ipck` setting, which controls whether `library` and `require` will complete on *installed packages*, is disabled by default because the first call to `installed.packages` is potentially time consuming (e.g. when packages are installed on a remote network file server). Note, however, that the results of a call to `installed.packages` is cached, so subsequent calls are usually fast, so turning this option on is not particularly onerous even in such situations.

Value

`rc.status` returns, as a list, the contents of an internal (unexported) environment that is used to record the results of the last completion attempt. This can be useful for debugging. For such use, one must resist the temptation to use completion when typing the call to `rc.status` itself, as that then becomes the last attempt by the time the call is executed.

The items of primary interest in the returned list are:

`comps`	the possible completions generated by the last call to `.completeToken`, as a character vector
`token`	the token that was (or, is to be) completed, as set by the last call to `.assignToken` (possibly inside a call to `.guessTokenFromLine`)
`linebuffer`	the full line, as set by the last call to `.assignLinebuffer`
`start`	the start position of the token in the line buffer, as set by the last call to `.assignStart`
`end`	the end position of the token in the line buffer, as set by the last call to `.assignEnd`
`fileName`	logical, indicating whether the cursor is currently inside quotes. If so, no completion is attempted. A reasonable default behaviour for the backend in that case is to fall back to filename completion.
`fguess`	the name of the function `rcompgen` thinks the cursor is currently inside
`isFirstArg`	logical. If cursor is inside a function, is it the first argument?

In addition, the components `settings` and `options` give the current values of settings and options respectively.

`rc.getOption` and `rc.options` behave much like `getOption` and `options` respectively.

Unexported API

There are several unexported functions in the package. Of these, a few are special because they provide the API through which other mechanisms can make use of the facilities provided by this package (they are unexported because they are not meant to be called directly by users). The usage of these functions are:

```
.assignToken(text)
.assignLinebuffer(line)
.assignStart(start)
.assignEnd(end)

.completeToken()
.retrieveCompletions()
```

```
.getFileComp()

.guessTokenFromLine()
.win32consoleCompletion(linebuffer, cursorPosition,
                        check.repeat = TRUE,
                        minlength = -1)

.addFunctionInfo(...)
```

The first four functions set up a completion attempt by specifying the token to be completed (`text`), and indicating where (`start` and `end`, which should be integers) the token is placed within the complete line typed so far (`line`).

Potenial completions of the token are generated by `.completeToken`, and the completions can be retrieved as an R character vector using `.retrieveCompletions`.

If the cursor is inside quotes, no completion is attempted. The function `.getFileComp` can be used after a call to `.completeToken` to determine if this is the case (returns `TRUE`), and alternative completions generated as deemed useful. In most cases, filename completion is a reasonable fallback.

The `.guessTokenFromLine` function is provided for use with backends that do not already break a line into tokens. It requires the linebuffer and endpoint (cursor position) to be already set, and itself sets the token and the start position. It returns the token as a character string. (This is used by the ESS completion hook example given in the `examples/altesscomp.el` file.)

The `.win32consoleCompletion` is similar in spirit, but is more geared towards the Windows GUI (or rather, any front-end that has no completion facilities of its own). It requires the linebuffer and cursor position as arguments, and returns a list with three components, `addition`, `possible` and `comps`. If there is an unambiguous extension at the current position, `addition` contains the additional text that should be inserted at the cursor. If there is more than one possibility, these are available either as a character vector of preformatted strings in `possible`, or as a single string in `comps`. `possible` consists of lines formatted using the current `width` option, so that printing them on the console one line at a time will be a reasonable way to list them. `comps` is a space separated (collapsed) list of the same completions, in case the front-end wishes to display it in some other fashion.

The `minlength` argument can be used to suppress completion when the token is too short (which can be useful if the front-end is set up to try completion on every keypress). If `check.repeat` is `TRUE`, it is detected if the same completion is being requested more than once in a row, and ambiguous completions are returned only in that case. This is an attempt to emulate GNU Readline behaviour, where a single TAB completes upto any unambiguous part, and multiple possibilities are reported only on two consecutive TABs.

As the various front-end interfaces evolve, the details of these functions are likely to change as well.

The function `.addFunctionInfo` can be used to add information about the permitted argument names for specific functions. Multiple named arguments are allowed in calls to it, where the tags are names of functions and values are character vectors representing valid arguments. When the `argdb` setting is `TRUE`, these are used as a source of valid argument names for the relevant functions.

Note

If you are uncomfortable with unsolicited evaluation of pieces of code, you should set `ops = FALSE`. Otherwise, trying to complete `foo@ba` will evaluate `foo`, trying to complete `foo[i,1:10]$ba` will evaluate `foo[i,1:10]`, etc. This should not be too bad, as explicit function calls (involving parentheses) are not evaluated in this manner. However, this *will* affect lazy loaded symbols (and presumably other promise type thingies).

Author(s)

Deepayan Sarkar, ⟨deepayan.sarkar@r-project.org⟩

`read.DIF` *Data Input from Spreadsheet*

Description

Reads a file in Data Interchange Format (DIF) and creates a data frame from it. DIF is a format for data matrices such as single spreadsheets.

Usage

```
read.DIF(file, header = FALSE,
           dec = ".", row.names, col.names,
           as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE,
           blank.lines.skip = TRUE,
           stringsAsFactors = default.stringsAsFactors(),
           transpose = FALSE)
```

Arguments

`file`	the name of the file which the data are to be read from, or a connection, or a complete URL.
`header`	a logical value indicating whether the spreadsheet contains the names of the variables as its first line. If missing, the value is determined from the file format: `header` is set to `TRUE` if and only if the first row contains only character values and the top left cell is empty.
`dec`	the character used in the file for decimal points.
`row.names`	a vector of row names. This can be a vector giving the actual row names, or a single number giving the column of the table which contains the row names, or character string giving the name of the table column containing the row names. If there is a header and the first row contains one fewer field than the number of columns, the first column in the input is used for the row names. Otherwise if `row.names` is missing, the rows are numbered. Using `row.names = NULL` forces row numbering.
`col.names`	a vector of optional names for the variables. The default is to use `"V"` followed by the column number.
`as.is`	the default behavior of `read.DIF` is to convert character variables (which are not converted to logical, numeric or complex) to factors. The variable `as.is` controls the conversion of columns not otherwise specified by `colClasses`. Its value is either a vector of logicals (values are recycled if necessary), or a vector of numeric or character indices which specify which columns should not be converted to factors. Note: to suppress all conversions including those of numeric columns, set `colClasses = "character"`. Note that `as.is` is specified per column (not per variable) and so includes the column of row names (if any) and any columns to be skipped.

`na.strings` a character vector of strings which are to be interpreted as `NA` values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields.

`colClasses` character. A vector of classes to be assumed for the columns. Recycled as necessary, or if the character vector is named, unspecified values are taken to be `NA`.

Possible values are `NA` (when `type.convert` is used), `"NULL"` (when the column is skipped), one of the atomic vector classes (logical, integer, numeric, complex, character, raw), or `"factor"`, `"Date"` or `"POSIXct"`. Otherwise there needs to be an `as` method (from package **methods**) for conversion from `"character"` to the specified formal class.

Note that `colClasses` is specified per column (not per variable) and so includes the column of row names (if any).

`nrows` the maximum number of rows to read in. Negative values are ignored.

`skip` the number of lines of the data file to skip before beginning to read data.

`check.names` logical. If `TRUE` then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names. If necessary they are adjusted (by `make.names`) so that they are, and also to ensure that there are no duplicates.

`blank.lines.skip`
logical: if `TRUE` blank lines in the input are ignored.

`stringsAsFactors`
logical: should character vectors be converted to factors?

`transpose` logical, indicating if the row and column interpretation should be transposed. Microsoft's Excel has been known to produce (non-standard conforming) DIF files which would need `transpose = TRUE` to be read correctly.

Value

A data frame (`data.frame`) containing a representation of the data in the file. Empty input is an error unless `col.names` is specified, when a 0-row data frame is returned: similarly giving just a header line if `header = TRUE` results in a 0-row data frame.

Note

The columns referred to in `as.is` and `colClasses` include the column of row names (if any).

Less memory will be used if `colClasses` is specified as one of the six atomic vector classes.

Author(s)

R Core; `transpose` option by Christoph Buser, ETH Zurich

References

The DIF format specification can be found by searching on `http://www.wotsit.org/`; the optional header fields are ignored. See also `http://en.wikipedia.org/wiki/Data_Interchange_Format`.

The term is likely to lead to confusion: Windows will have a 'Windows Data Interchange Format (DIF) data format' as part of its WinFX system, which may or may not be compatible.

See Also

The *R Data Import/Export* manual.

`scan`, `type.convert`, `read.fwf` for reading *f*ixed *w*idth *f*ormatted input; `read.table`; `data.frame`.

Examples

```
## read.DIF() needs transpose=TRUE for file exported from Excel
udir <- system.file("misc", package="utils")
dd <- read.DIF(file.path(udir, "exDIF.dif"), header= TRUE, transpose=TRUE)
dc <- read.csv(file.path(udir, "exDIF.csv"), header= TRUE)
stopifnot(identical(dd,dc), dim(dd) == c(4,2))
```

---

read.fortran *Read fixed-format data*

Description

Read fixed-format data files using Fortran-style format specifications.

Usage

```
read.fortran(file, format, ..., as.is = TRUE, colClasses = NA)
```

Arguments

`file`	File or connection to read from
`format`	Character vector or list of vectors. See 'Details' below.
`...`	Other arguments for `read.table`
`as.is`	Keep characters as characters?
`colClasses`	Variable classes to override defaults. See `read.table` for details.

Details

The format for a field is of one of the following forms: `rFl.d`, `rDl.d`, `rXl`, `rAl`, `rIl`, where `l` is the number of columns, `d` is the number of decimal places, and `r` is the number of repeats. `F` and `D` are numeric formats, `A` is character, `I` is integer, and `X` indicates columns to be skipped. The repeat code `r` and decimal place code `d` are always optional. The length code `l` is required except for `X` formats when `r` is present.

For a single-line record, `format` should be a character vector. For a multiline record it should be a list with a character vector for each line.

Skipped (`X`) columns are not passed to `read.table`, so `colClasses`, `col.names`, and similar arguments passed to `read.table` should not reference these columns.

Value

A data frame

See Also

`read.fwf`, `read.csv`

Examples

```
ff <- tempfile()
cat(file=ff, "123456", "987654", sep="\n")
read.fortran(ff, c("F2.1","F2.0","I2"))
read.fortran(ff, c("2F1.0","2X","2A1"))
unlink(ff)
cat(file=ff, "123456AB", "987654CD", sep="\n")
read.fortran(ff, list(c("2F3.1","A2"), c("3I2","2X")))
unlink(ff)
```

`read.fwf` *Read Fixed Width Format Files*

Description

Read a table of **f**ixed **w**idth **f**ormatted data into a `data.frame`.

Usage

```
read.fwf(file, widths, header = FALSE, sep = "\t",
         skip = 0, row.names, col.names, n = -1,
         buffersize = 2000, ...)
```

Arguments

`file`	the name of the file which the data are to be read from. Alternatively, `file` can be a `connection`, which will be opened if necessary, and if so closed at the end of the function call.
`widths`	integer vector, giving the widths of the fixed-width fields (of one line), or list of integer vectors giving widths for multiline records.
`header`	a logical value indicating whether the file contains the names of the variables as its first line. If present, the names must be delimited by `sep`.
`sep`	character; the separator used internally; should be a character that does not occur in the file (except in the header).
`skip`	number of initial lines to skip; see `read.table`.
`row.names`	see `read.table`.
`col.names`	see `read.table`.
`n`	the maximum number of records (lines) to be read, defaulting to no limit.
`buffersize`	Maximum number of lines to read at one time
`...`	further arguments to be passed to `read.table`. Useful further arguments include `as.is`, `na.strings`, `colClasses` and `strip.white`.

Details

Multiline records are concatenated to a single line before processing. Fields that are of zero-width or are wholly beyond the end of the line in `file` are replaced by `NA`.

Negative-width fields are used to indicate columns to be skipped, eg `-5` to skip 5 columns. These fields are not seen by `read.table` and so should not be included in a `col.names` or `colClasses` argument (nor in the header line, if present).

Reducing the `buffersize` argument may reduce memory use when reading large files with long lines. Increasing `buffersize` may result in faster processing when enough memory is available.

Value

A `data.frame` as produced by `read.table` which is called internally.

Author(s)

Brian Ripley for R version: original `Perl` by Kurt Hornik.

See Also

`scan` and `read.table`.

Examples

```
ff <- tempfile()
cat(file=ff, "123456", "987654", sep="\n")
read.fwf(ff, widths=c(1,2,3))    #> 1 23 456 \ 9 87 654
read.fwf(ff, widths=c(1,-2,3))   #> 1 456 \ 9 654
unlink(ff)
cat(file=ff, "123", "987654", sep="\n")
read.fwf(ff, widths=c(1,0, 2,3))    #> 1 NA 23 NA \ 9 NA 87 654
unlink(ff)
cat(file=ff, "123456", "987654", sep="\n")
read.fwf(ff, widths=list(c(1,0, 2,3), c(2,2,2))) #> 1 NA 23 456 98 76 54
unlink(ff)
```

`read.socket` *Read from or Write to a Socket*

Description

`read.socket` reads a string from the specified socket, `write.socket` writes to the specified socket. There is very little error checking done by either.

Usage

```
read.socket(socket, maxlen = 256, loop = FALSE)
write.socket(socket, string)
```

Arguments

socket	a socket object
maxlen	maximum length of string to read
loop	wait for ever if there is nothing to read?
string	string to write to socket

Value

read.socket returns the string read.

Author(s)

Thomas Lumley

See Also

close.socket, make.socket

Examples

```
finger <- function(user, host = "localhost", port = 79, print = TRUE)
{
    if (!is.character(user))
        stop("user name must be a string")
    user <- paste(user,"\r\n")
    socket <- make.socket(host, port)
    on.exit(close.socket(socket))
    write.socket(socket, user)
    output <- character(0)
    repeat{
        ss <- read.socket(socket)
        if (ss == "") break
        output <- paste(output, ss)
    }
    close.socket(socket)
    if (print) cat(output)
    invisible(output)
}
## Not run:
finger("root")  ## only works if your site provides a finger daemon
## End(Not run)
```

---

read.table *Data Input*

Description

Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

Usage

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", row.names, col.names,
           as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#",
           allowEscapes = FALSE, flush = FALSE,
           stringsAsFactors = default.stringsAsFactors(),
           encoding = "unknown")

read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",
         fill = TRUE, comment.char="", ...)

read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=",",
          fill = TRUE, comment.char="", ...)

read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",
           fill = TRUE, comment.char="", ...)

read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=",",
            fill = TRUE, comment.char="", ...)
```

Arguments

`file`	the name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an *absolute* path, the file name is *relative* to the current working directory, `getwd()`. Tilde-expansion is performed where supported. Alternatively, `file` can be a readable text-mode `connection` (which will be opened for reading if necessary, and if so `close`d (and hence destroyed) at the end of the function call). (If `stdin()` is used, the prompts for lines may be somewhat confusing. Terminate input with a blank line or an EOF signal, `Ctrl-D` on Unix and `Ctrl-Z` on Windows. Any pushback on `stdin()` will be cleared before return.) `file` can also be a complete URL. To read a data file not in the current encoding (for example a Latin-1 file in a UTF-8 locale or conversely) use a `file` connection setting the `encoding` argument.
`header`	a logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: `header` is set to `TRUE` if and only if the first row contains one fewer field than the number of columns.
`sep`	the field separator character. Values on each line of the file are separated by this character. If `sep = ""` (the default for `read.table`) the separator is 'white space', that is one or more spaces, tabs, newlines or carriage returns.
`quote`	the set of quoting characters. To disable quoting altogether, use `quote = ""`. See `scan` for the behaviour on quotes embedded in quotes. Quoting is only considered for columns read as character, which is all of them unless `colClasses` is specified.

`dec` the character used in the file for decimal points.

`row.names` a vector of row names. This can be a vector giving the actual row names, or a single number giving the column of the table which contains the row names, or character string giving the name of the table column containing the row names.

If there is a header and the first row contains one fewer field than the number of columns, the first column in the input is used for the row names. Otherwise if `row.names` is missing, the rows are numbered.

Using `row.names = NULL` forces row numbering. Missing or `NULL` `row.names` generate row names that are considered to be 'automatic' (and not preserved by `as.matrix`).

`col.names` a vector of optional names for the variables. The default is to use `"V"` followed by the column number.

`as.is` the default behavior of `read.table` is to convert character variables (which are not converted to logical, numeric or complex) to factors. The variable `as.is` controls the conversion of columns not otherwise specified by `colClasses`. Its value is either a vector of logicals (values are recycled if necessary), or a vector of numeric or character indices which specify which columns should not be converted to factors.

Note: to suppress all conversions including those of numeric columns, set `colClasses = "character"`.

Note that `as.is` is specified per column (not per variable) and so includes the column of row names (if any) and any columns to be skipped.

`na.strings` a character vector of strings which are to be interpreted as `NA` values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields.

`colClasses` character. A vector of classes to be assumed for the columns. Recycled as necessary, or if the character vector is named, unspecified values are taken to be `NA`.

Possible values are `NA` (when `type.convert` is used), `"NULL"` (when the column is skipped), one of the atomic vector classes (logical, integer, numeric, complex, character, raw), or `"factor"`, `"Date"` or `"POSIXct"`. Otherwise there needs to be an `as` method (from package **methods**) for conversion from `"character"` to the specified formal class.

Note that `colClasses` is specified per column (not per variable) and so includes the column of row names (if any).

`nrows` integer: the maximum number of rows to read in. Negative and other invalid values are ignored.

`skip` integer: the number of lines of the data file to skip before beginning to read data.

`check.names` logical. If `TRUE` then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names. If necessary they are adjusted (by `make.names`) so that they are, and also to ensure that there are no duplicates.

`fill` logical. If `TRUE` then in case the rows have unequal length, blank fields are implicitly added. See 'Details'.

`strip.white` logical. Used only when `sep` has been specified, and allows the stripping of leading and trailing white space from `character` fields (`numeric` fields are always stripped). See `scan` for further details, remembering that the columns may include the row names.

`blank.lines.skip`
: logical: if `TRUE` blank lines in the input are ignored.

`comment.char`
: character: a character vector of length one containing a single character or an empty string. Use `""` to turn off the interpretation of comments altogether.

`allowEscapes`
: logical. Should C-style escapes such as `\n` be processed or read verbatim (the default)? Note that if not within quotes these could be interpreted as a delimiter (but not as a comment character). For more details see `scan`.

`flush`
: logical: if `TRUE`, `scan` will flush to the end of the line after reading the last of the fields requested. This allows putting comments after the last field.

`stringsAsFactors`
: logical: should character vectors be converted to factors? Note that this is overridden bu `as.is` and `colClasses`, both of which allow finer control.

`encoding`
: encoding to be assumed for input strings. It is used to mark character strings as known to be in Latin-1 or UTF-8: it is not used to re-encode the input. For an example of how to do so, see the examples under `file`.

`...`
: Further arguments to be passed to `read.table`.

Details

This function is the principal means of reading tabular data into R.

A field or line is 'blank' if it contains nothing (except whitespace if no separator is specified) before a comment character or the end of the field or line.

If `row.names` is not specified and the header line has one less entry than the number of columns, the first column is taken to be the row names. This allows data frames to be read in from the format in which they are printed. If `row.names` is specified and does not refer to the first column, that column is discarded from such files.

The number of data columns is determined by looking at the first five lines of input (or the whole file if it has less than five lines), or from the length of `col.names` if it is specified and is longer. This could conceivably be wrong if `fill` or `blank.lines.skip` are true, so specify `col.names` if necessary.

`read.csv` and `read.csv2` are identical to `read.table` except for the defaults. They are intended for reading 'comma separated value' files ('.csv') or (`read.csv2`) the variant used in countries that use a comma as decimal point and a semicolon as field separator. Similarly, `read.delim` and `read.delim2` are for reading delimited files, defaulting to the TAB character for the delimiter. Notice that `header = TRUE` and `fill = TRUE` in these variants, and that the comment character is disabled.

The rest of the line after a comment character is skipped; quotes are not processed in comments. Complete comment lines are allowed provided `blank.lines.skip = TRUE`; however, comment lines prior to the header must have the comment character in the first non-blank column.

Quoted fields with embedded newlines are supported except after a comment character.

Note that unless `colClasses` is specified, all columns are read as character columns and then converted. This means that quotes are interpreted in all fields and that a column of values like `"42"` will result in an integer column.

Value

A data frame (`data.frame`) containing a representation of the data in the file.

Empty input is an error unless `col.names` is specified, when a 0-row data frame is returned: similarly giving just a header line if `header = TRUE` results in a 0-row data frame. Note that in either case tthe columns will logical unless `colClasses` was supplied.

Character strings in the result (including factor levels) will have a declared encoding if `encoding` is `"latin1"` or `"UTF-8"`.

Note

The columns referred to in `as.is` and `colClasses` include the column of row names (if any).

Less memory will be used if `colClasses` is specified as one of the six atomic vector classes. This can be particularly so when reading a column that takes many distinct numeric values, as storing each distinct value as a character string can take up to 14 times as much memory as storing it as an integer.

Using `nrows`, even as a mild over-estimate, will help memory usage.

Using `comment.char = ""` will be appreciably faster than the `read.table` default.

`read.table` is not the right tool for reading large matrices, especially those with many columns: it is designed to read *data frames* which may have columns of very different classes. Use `scan` instead.

References

Chambers, J. M. (1992) *Data for models.* Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

See Also

The *R Data Import/Export* manual.

`scan`, `type.convert`, `read.fwf` for reading *f*ixed *w*idth *f*ormatted input; `write.table`; `data.frame`.

`count.fields` can be useful to determine problems with reading files which result in reports of incorrect record lengths.

`recover` *Browsing after an Error*

Description

This function allows the user to browse directly on any of the currently active function calls, and is suitable as an error option. The expression `options(error=recover)` will make this the error option.

Usage

```
recover()
```

Details

When called, `recover` prints the list of current calls, and prompts the user to select one of them. The standard R `browser` is then invoked from the corresponding environment; the user can type ordinary S language expressions to be evaluated in that environment.

When finished browsing in this call, type `c` to return to `recover` from the browser. Type another frame number to browse some more, or type `0` to exit `recover`.

The use of `recover` largely supersedes `dump.frames` as an error option, unless you really want to wait to look at the error. If `recover` is called in non-interactive mode, it behaves like `dump.frames`. For computations involving large amounts of data, `recover` has the advantage that it does not need to copy out all the environments in order to browse in them. If you do decide to quit interactive debugging, call `dump.frames` directly while browsing in any frame (see the examples).

Value

Nothing useful is returned. However, you *can* invoke `recover` directly from a function, rather than through the error option shown in the examples. In this case, execution continues after you type `0` to exit `recover`.

Compatibility Note

The R `recover` function can be used in the same way as the S-Plus function of the same name; therefore, the error option shown is a compatible way to specify the error action. However, the actual functions are essentially unrelated and interact quite differently with the user. The navigating commands `up` and `down` do not exist in the R version; instead, exit the browser and select another frame.

References

John M. Chambers (1998). *Programming with Data*; Springer.
See the compatibility note above, however.

See Also

`browser` for details about the interactive computations; `options` for setting the error option; `dump.frames` to save the current environments for later debugging.

Examples

```
## Not run:

options(error = recover) # setting the error option

### Example of interaction

> myFit <- lm(y ~ x, data = xy, weights = w)
Error in lm.wfit(x, y, w, offset = offset, ...) :
        missing or negative weights not allowed

Enter a frame number, or 0 to exit
1:lm(y ~ x, data = xy, weights = w)
2:lm.wfit(x, y, w, offset = offset, ...)
Selection: 2
Called from: eval(expr, envir, enclos)
Browse[1]> objects() # all the objects in this frame
[1] "method" "n"      "ny"     "offset" "tol"    "w"
[7] "x"      "y"
Browse[1]> w
[1] -0.5013844  1.3112515  0.2939348 -0.8983705 -0.1538642
[6] -0.9772989  0.7888790 -0.1919154 -0.3026882
Browse[1]> dump.frames() # save for offline debugging
```

```
Browse[1]> c # exit the browser

Enter a frame number, or 0 to exit
1:lm(y ~ x, data = xy, weights = w)
2:lm.wfit(x, y, w, offset = offset, ...)
Selection: 0 # exit recover
>

## End(Not run)
```

`relist` *Allow Re-Listing an unlist()ed Object*

Description

`relist()` is an S3 generic function with a few methods in order to allow easy inversion of `unlist(obj)` when that is used with an object `obj` of (S3) class `"relistable"`.

Usage

```
relist(flesh, skeleton)
## Default S3 method:
relist(flesh, skeleton = attr(flesh, "skeleton"))
## S3 method for class 'factor':
relist(flesh, skeleton = attr(flesh, "skeleton"))
## S3 method for class 'list':
relist(flesh, skeleton = attr(flesh, "skeleton"))
## S3 method for class 'matrix':
relist(flesh, skeleton = attr(flesh, "skeleton"))

as.relistable(x)
is.relistable(x)

## S3 method for class 'relistable':
unlist(x, recursive = TRUE, use.names = TRUE)
```

Arguments

`flesh`	a vector to be relisted
`skeleton`	a list, the structure of which determines the structure of the result
`x`	an R object, typically a list (or vector).
`recursive`	logical. Should unlisting be applied to list components of `x`?
`use.names`	logical. Should names be preserved?

Details

Some functions need many parameters, which are most easily represented in complex structures, e.g., nested lists. Unfortunately, many mathematical functions in R, including `optim` and `nlm` can only operate on functions whose domain is a vector. R has `unlist()` to convert nested list objects into a vector representation. `relist()`, it's methods and the functionality mentioned here provide

the inverse operation to convert vectors back to the convenient structural representation. This allows structured functions (such as `optim()`) to have simple mathematical interfaces.

For example, a likelihood function for a multivariate normal model needs a variance-covariance matrix and a mean vector. It would be most convenient to represent it as a list containing a vector and a matrix. A typical parameter might look like

```
list(mean=c(0, 1), vcov=cbind(c(1, 1), c(1, 0))).
```

However, `optim` cannot operate on functions that take lists as input; it only likes numeric vectors. The solution is conversion. Given a function `mvdnorm(x, mean, vcov, log=FALSE)` which computes the required probability density, then

```
ipar <- list(mean=c(0, 1), vcov=cbind(c(1, 1), c(1, 0)))
initial.param <- as.relistable(ipar)

ll <- function(param.vector)
{
   param <- relist(param.vector, skeleton=ipar))
   -sum(mvdnorm(x, mean = param$mean, vcov = param$vcov,
                log = TRUE))
}

optim(unlist(initial.param), ll)
```

`relist` takes two parameters: skeleton and flesh. Skeleton is a sample object that has the right `shape` but the wrong content. `flesh` is a vector with the right content but the wrong shape. Invoking

```
relist(flesh, skeleton)
```

will put the content of flesh on the skeleton. You don't need to specify skeleton explicitly if the skeleton is stored as an attribute inside flesh. In particular, if flesh was created from some object obj with `unlist(as.relistable(obj))` then the skeleton attribute is automatically set. (Note that this does not apply to the example here, as `optim` is creating a new vector to pass to `ll` and not its `par` argument.)

As long as `skeleton` has the right shape, it should be a precise inverse of `unlist`. These equalities hold:

```
relist(unlist(x), x) == x
unlist(relist(y, skeleton)) == y

x <- as.relistable(x)
relist(unlist(x)) == x
```

Value

an object of (S3) class `"relistable"` (and `"list"`).

Author(s)

R Core, based on a code proposal by Andrew Clausen.

See Also

unlist

Examples

```
ipar <- list(mean=c(0, 1), vcov=cbind(c(1, 1), c(1, 0)))
initial.param <- as.relistable(ipar)
ul <- unlist(initial.param)
relist(ul)
stopifnot(identical(relist(ul), initial.param))
```

REMOVE *Remove Add-on Packages*

Description

Utility for removing add-on packages.

Usage

```
R CMD REMOVE [options] [-l lib] pkgs
```

Arguments

pkgs	a list with the names of the packages to be removed.
lib	the path name of the R library tree to remove from. May be absolute or relative.
options	further options.

Details

If used as `R CMD REMOVE pkgs` without explicitly specifying `lib`, packages are removed from the library tree rooted at the first directory in the library path which would be used by R run in the current environment.

To remove from the library tree `lib`, use `R CMD REMOVE -l lib pkgs`.

Use `R CMD REMOVE --help` for more usage information.

Note

Some binary distributions of R have `INSTALL` in a separate bundle, e.g. an `R-devel` RPM.

See Also

INSTALL, remove.packages

remove.packages *Remove Installed Packages*

Description

Removes installed packages/bundles and updates index information as necessary.

Usage

```
remove.packages(pkgs, lib, version)
```

Arguments

pkgs	a character vector with the names of the package(s) or bundle(s) to be removed.
lib	a character vector giving the library directories to remove the packages from. If missing, defaults to the first element in `.libPaths()`.
version	A character vector specifying version(s) with versioned installs of the package(s) to remove. If none is provided, the system will remove an unversioned install of the package if one is found, otherwise the latest versioned install.

Details

If an element of `pkgs` matches a bundle name, all the packages in the bundle will be removed. This takes precedence over matching a package name.

`pkgs` and `version` will be recycled if necessary to the length of the longer one.

See Also

`REMOVE` for a command line version; `install.packages` for installing packages.

RHOME *R Home Directory*

Description

Returns the location of the R home directory, which is the root of the installed R tree.

Usage

```
R RHOME
```

roman *Roman Numerals*

Description

Manipulate integers as roman numerals.

Usage

```
as.roman(x)
```

Arguments

x	a numeric vector, or a character vector of arabic or roman numerals.

Details

`as.roman` creates objects of class `"roman"` which are internally represented as integers, and have suitable methods for printing, formatting, subsetting, and coercion to `character`.

Only numbers between 1 and 3899 have a unique representation as roman numbers.

References

Wikipedia contributors (2006). Roman numerals. Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Roman_numerals\&oldid=78252134. Accessed September 29, 2006.

Examples

```
## First five roman 'numbers'.
(y <- as.roman(1 : 5))
## Middle one.
y[3]
## Current year as a roman number.
(y <- as.roman(format(Sys.Date(), "%Y")))
## 10 years ago ...
y - 10
```

Rprof *Enable Profiling of R's Execution*

Description

Enable or disable profiling of the execution of R expressions.

Usage

```
Rprof(filename = "Rprof.out", append = FALSE, interval = 0.02,
      memory.profiling=FALSE)
```

Arguments

`filename`	The file to be used for recording the profiling results. Set to `NULL` or `""` to disable profiling.
`append`	logical: should the file be over-written or appended to?
`interval`	real: time interval between samples.
`memory.profiling`	logical: write memory use information to the file?

Details

Enabling profiling automatically disables any existing profiling to another or the same file.

Profiling works by writing out the call stack every `interval` seconds, to the file specified. Either the `summaryRprof` function or the Perl script `R CMD Rprof` can be used to process the output file to produce a summary of the usage; use `R CMD Rprof --help` for usage information.

Note that the timing interval cannot be too small: once the timer goes off, the information is not recorded until the next timing click (probably in the range 1–10msecs).

Note

Profiling is not available on all platforms. By default, it is attempted to compile support for profiling. Configure R with '`--disable-R-profiling`' to change this.

As R profiling uses the same mechanisms as C profiling, the two cannot be used together, so do not use `Rprof` in an executable built for profiling.

See Also

The chapter on "Tidying and profiling R code" in "Writing R Extensions" (see the 'doc/manual' subdirectory of the R source tree).

`summaryRprof`

`tracemem`, `Rprofmem` for other ways to track memory use.

Examples

```
## Not run:
Rprof()
## some code to be profiled
Rprof(NULL)
## some code NOT to be profiled
Rprof(append=TRUE)
## some code to be profiled
Rprof(NULL)
...
## Now post-process the output as described in Details
## End(Not run)
```

`Rprofmem` *Enable Profiling of R's Memory Use*

Description

Enable or disable reporting of memory allocation in R.

Usage

```
Rprofmem(filename = "Rprofmem.out", append = FALSE, threshold = 0)
```

Arguments

`filename`	The file to be used for recording the memory allocations. Set to `NULL` or `""` to disable reporting.
`append`	logical: should the file be over-written or appended to?
`threshold`	numeric: allocations on R's "large vector" heap larger than this number of bytes will be reported.

Details

Enabling profiling automatically disables any existing profiling to another or the same file.

Profiling writes the call stack to the specified file every time `malloc` is called to allocate a large vector object or to allocate a page of memory for small objects. The size of a page of memory and the size above which `malloc` is used for vectors are compile-time constants, by default 2000 and 128 bytes respectively.

The profiler tracks allocations, some of which will be to previously used memory and will not increase the total memory use of R.

Value

None

Note

The memory profiler slows down R even when not in use, and so is a compile-time option. The memory profiler can be used at the same time as other R and C profilers.

See Also

The R sampling profiler, `Rprof` also collects memory information.

`tracemem` traces duplications of specific objects.

The "Writing R Extensions" manual section on "Tidying and profiling R code"

Examples

```
## Not run:
## not supported unless R is compiled to support it.
Rprofmem("Rprofmem.out", threshold=1000)
example(glm)
Rprofmem(NULL)
noquote(readLines("Rprofmem.out", n=5))
## End(Not run)
```

Rscript *Scripting Front-End for R*

Description

This is an alternative front end for use in `#!` scripts and other scripting applications.

Usage

```
Rscript [options] [-e expression] file [args]
```

Arguments

`options`	A list of options beginning with `--`. These can be any of the options of the standard R front-end, and also those described in the details.
`expression`	a R expression.
`file`	The name of a file containing R commands. `-` indicates 'stdin'.
`args`	Arguments to be passed to the script in `file`.

Details

`Rscript --help` gives details of usage, and `Rscript --version` gives the version of `Rscript`.

Other invocations invoke the R front-end with selected options. This front-end is convenient for writing `#!` scripts since it is an executable and takes `file` directly as an argument. Options `--slave --no-restore` are always supplied: these imply `--no-save`.

Either one or more '`-e`' options or `file` should be supplied. When using '`-e`' options be aware of the quoting rules in the shell used: see the examples.

Additional options accepted (before `file` or `args`) are

`-verbose` gives details of what `Rscript` is doing. Also passed on to R.

`-default-packages=list` where `list` is a comma-separated list of package names or `NULL`. Sets the environment variable `R_DEFAULT_PACKAGES` which determines the packages loaded on startup. The default for `Rscript` omits **methods** as it takes about 60% of the startup time.

Normally the version of R is determined at installation, but this can be overridden by setting the environment variable `RHOME`.

`stdin()` refers to the input file, and `file("stdin")` to the `stdin` file stream of the process.

Note

`Rscript` is only supported on systems with the `execv` system call.

Examples

```
## Not run:
Rscript -e 'date()' -e 'format(Sys.time(), "%a %b %d %X %Y")'

## example #! script for a Unix-alike

#! /path/to/Rscript --vanilla --default-packages=utils
args <- commandArgs(TRUE)
res <- try(install.packages(args))
if(inherits(res, "try-error")) q(status=1) else q()

## End(Not run)
```

`RShowDoc` *Show R Manuals and Other Documentation*

Description

Utility function to find and display R documentation.

Usage

```
RShowDoc(what, type = c("pdf", "html", "txt"), package)
```

Arguments

`what`	a character string: see 'Details'.
`type`	an optional character string giving the preferred format.
`package`	an optional character string specifying the name of a package within which to look for documentation.

Details

`what` can specify one of several different sources of documentation, including the R manuals (`R-admin`, `R-data`, `R-exts`, `R-intro`, `R-ints`, `R-lang`), `NEWS`, `COPYING` (the GPL licence), `FAQ` (also available as `R-FAQ`), and the files in '*R_HOME*/doc'.

If `package` is supplied, documentation is looked for in the 'doc' and top-level directories of an installed package of that name.

If `what` is missing a brief usage message is printed.

The documentation types are tried in turn starting with the first specified in `type` (or `"pdf"` if none is specified).

Value

A invisible character string given the path to the file found.

Examples

```
## Not run:
RShowDoc("R-lang")
RShowDoc("FAQ", type="html")
RShowDoc("frame", package="grid")
RShowDoc("changes.txt", package="grid")
RShowDoc("NEWS", package="MASS")
## End(Not run)
```

`RSiteSearch` *Search for Key Words or Phrases in the R-help Mailing List Archives or Documentation*

Description

Search for key words or phrases in the R-help mailing list archives, or R manuals and help pages, using the search engine at `http://search.r-project.org` and view them in a web browser.

Usage

```
RSiteSearch(string,
            restrict = c("Rhelp02a", "functions", "docs"),
            format = c("normal", "short"),
            sortby = c("score", "date:late", "date:early",
                       "subject", "subject:descending",
                       "from", "from:descending",
                       "size", "size:descending"),
            matchesPerPage = 20)
```

Arguments

`string`	word(s) or phrase to search. If the words are to be searched as one entity, enclose all words in braces (see example).
`restrict`	a character vector, typically of length larger than one: What areas to search in: `Rhelp02a` for R-help mailing list archive since 2002, `Rhelp01` for mailing list archive before 2002, `docs` for R manuals, `functions` for help pages. `R-devel` for R-devel mailing list. Use `c()` to specify more than one.
`format`	`normal` or `short` (no excerpts); can be abbreviated.
`sortby`	character string (can be abbreviated) indicating how to sort the search results: (`score`, `date:late` for sorting by date with latest results first, `date:early` for earliest first, `subject` for subject in alphabetical order, `subject:descending` for reverse alphabetical order, `from` or `from:descending` for sender (when applicable), `size` or `size:descending` for size.)
`matchesPerPage`	How many items to show per page.

Details

This function is designed to work with the search site at http://search.r-project.org, and depends on that site continuing to be made available (thanks to Jonathan Baron and the School of Arts and Sciences of the University of Pennslyvania).

Unique partial matches will work for all arguments. Each new browser window will stay open unless you close it.

Value

(Invisibly) the complete URL passed to the browser, including the query string.

Author(s)

Andy Liaw and Jonathan Baron

See Also

`help.search`, `help.start` for local searches.

`browseURL` for how the help file is displayed.

Examples

```
 # need Internet connection
RSiteSearch("{logistic regression}") # matches exact phrase
Sys.sleep(5) # allow browser to open, take a quick look
RSiteSearch("Baron Liaw", restrict = "Rhelp02a")
## Search in R-devel archive and documents  (and store the query-string):
Sys.sleep(5)
fullquery <- RSiteSearch("S4", restrict = c("R-dev", "docs"))
fullquery # a string of ~ 116 characters
## the latest purported bug reports, responses ...
Sys.sleep(5)
RSiteSearch("bug", restrict = "R-devel", sortby = "date:late")
```

`Rtangle` *R Driver for Stangle*

Description

A driver for `Stangle` that extracts R code chunks.

Usage

```
Rtangle()
RtangleSetup(file, syntax, output = NULL, annotate = TRUE,
             split = FALSE, prefix = TRUE, quiet = FALSE)
```

Arguments

file	Name of Sweave source file.
syntax	An object of class SweaveSyntax.
output	Name of output file, default is to remove extension '.nw', '.Rnw' or '.Snw' and to add extension '.R'. Any directory names in file are also removed such that the output is created in the current working directory.
annotate	By default, code chunks are separated by comment lines specifying the names and numbers of the code chunks. If FALSE, only the code chunks without any decorating comments are extracted.
split	Split output in single files per code chunk?
prefix	If split = TRUE, prefix the chunk labels by the basename of the input file to get output file names?
quiet	If TRUE all progress messages are suppressed.

Author(s)

Friedrich Leisch

References

Friedrich Leisch: Sweave User Manual, 2008
http://www.stat.uni-muenchen.de/~leisch/Sweave

See Also

Sweave, RweaveLatex

RweaveLatex *R/LaTeX Driver for Sweave*

Description

A driver for Sweave that translates R code chunks in LaTeX files.

Usage

```
RweaveLatex()

RweaveLatexSetup(file, syntax, output = NULL, quiet = FALSE,
                 debug = FALSE, echo = TRUE, eval = TRUE,
                 keep.source = FALSE, split = FALSE,
                 stylepath, pdf = TRUE, eps = TRUE)
```

Arguments

`file`	Name of Sweave source file.
`syntax`	An object of class `SweaveSyntax`.
`output`	Name of output file, default is to remove extension '.nw', '.Rnw' or '.Snw' and to add extension '.tex'. Any directory names in `file` are also removed such that the output is created in the current working directory.
`quiet`	If `TRUE` all progress messages are suppressed.
`debug`	If `TRUE`, input and output of all code chunks is copied to the console.
`stylepath`	See 'Details'.
`echo`	set default for option `echo`, see details below.
`eval`	set default for option `eval`, see details below.
`keep.source`	set default for option `keep.source`, see details below.
`split`	set default for option `split`, see details below.
`pdf`	set default for option `pdf`, see details below.
`eps`	set default for option `eps`, see details below.

Details

The LaTeX file generated needs to contain `\usepackage{Sweave}`, and if this is not present in the Sweave source file, it is inserted by the `RweaveLatex` driver. If `stylepath = TRUE`, a hard-coded path to the file 'Sweave.sty' in the R installation is set in place of `Sweave`. The hard-coded path makes the TeX file less portable, but avoids the problem of installing the current version of 'Sweave.sty' to some place in your TeX input path. However, TeX may not be able to process the hard-coded path if it contains spaces (as it often will under Windows) or TeX special characters.

The default in R prior to 2.7.0 was `stylepath = TRUE`. It is now taken from the environment variable `SWEAVE_STYLEPATH_DEFAULT`, or is `FALSE` it that is unset or empty. If set, it should be exactly `TRUE` or `FALSE`: any other values are taken as `FALSE`.

Supported Options

`RweaveLatex` supports the following options for code chunks (the values in parentheses show the default values):

echo: logical (`TRUE`). Include S code in the output file?

keep.source: logical (`FALSE`). When echoing, if `keep.source == TRUE` the original source is copied to the file. Otherwise, deparsed source is echoed.

eval: logical (`TRUE`). If `FALSE`, the code chunk is not evaluated, and hence no text or graphical output produced.

results: character string (`verbatim`). If `verbatim`, the output of S commands is included in the verbatim-like Soutput environment. If `tex`, the output is taken to be already proper latex markup and included as is. If `hide` then all output is completely suppressed (but the code executed during the weave).

print: logical (`FALSE`) If `TRUE`, each expression in the code chunk is wrapped into a `print()` statement before evaluation, such that the values of all expressions become visible.

term: logical (`TRUE`). If `TRUE`, visibility of values emulates an interactive R session: values of assignments are not printed, values of single objects are printed. If `FALSE`, output comes only from explicit `print` or `cat` statements.

split: logical (FALSE). If TRUE, text output is written to separate files for each code chunk.

strip.white: character string (false). If true, blank lines at the beginning and end of output are removed. If all, then all blank lines are removed from the output.

prefix: logical (TRUE). If TRUE generated filenames of figures and output have a common prefix.

prefix.string: a character string, default is the name of the '.Snw' source file.

include: logical (TRUE), indicating whether input statements for text output and includegraphics statements for figures should be auto-generated. Use include = FALSE if the output should appear in a different place than the code chunk (by placing the input line manually).

fig: logical (FALSE), indicating whether the code chunk produces graphical output. Note that only one figure per code chunk can be processed this way.

eps: logical (TRUE), indicating whether EPS figures should be generated. Ignored if fig = FALSE.

pdf: logical (TRUE), indicating whether PDF figures should be generated. Ignored if fig = FALSE.

pdf.version, pdf.encoding: passed to pdf to set the version and encoding. Defaults taken fron pdf.options().

width: numeric (6), width of figures in inches.

height: numeric (6), height of figures in inches.

expand: logical (TRUE). Expand references to other chunks so that only R code appears in the output file. If FALSE, the chunk reference (e.g. <<chunkname>>) will appear. The expand=FALSE option requires keep.source = TRUE or it will have no effect.

concordance: logical (FALSE). Write a concordance file to link the input line numbers to the output line numbers. This is an experimental feature; see the source code for the output format, which is subject to change in future releases.

Author(s)

Friedrich Leisch

References

Friedrich Leisch: Sweave User Manual, 2008
http://www.stat.uni-muenchen.de/~leisch/Sweave

See Also

Sweave, Rtangle

---

savehistory *Load or Save or Display the Commands History*

Description

Load or save or display the commands history.

Usage

```
loadhistory(file = ".Rhistory")
savehistory(file = ".Rhistory")

history(max.show = 25, reverse = FALSE, pattern, ...)

timestamp(stamp = date(),
          prefix = "##------ ", suffix = " ------##",
          quiet = FALSE)
```

Arguments

file	The name of the file in which to save the history, or from which to load it. The path is relative to the current working directory.
max.show	The maximum number of lines to show. Inf will give all of the currently available history.
reverse	logical. If true, the lines are shown in reverse order. Note: this is not useful when there are continuation lines.
pattern	A character string to be matched against the lines of the history
...	Arguments to be passed to grep when doing the matching.
stamp	A value or vector of values to be written into the history.
prefix	A prefix to apply to each line.
suffix	A suffix to apply to each line.
quiet	If TRUE, suppress printing timestamp to the console.

Details

There are several history mechanisms available for the different R consoles, which work in similar but not identical ways. Other uses of R, in particular embedded uses, may have no history. This works under the readline and GNOME and MacOS X consoles, but not otherwise (for example, in batch use or in an embedded application).

The readline history mechanism is controlled by two environment variables: R_HISTSIZE controls the number of lines that are saved (default 512), and R_HISTFILE sets the filename used for the loading/saving of history if requested at the beginning/end of a session (but not the default for these functions). There is no limit on the number of lines of history retained during a session, so setting R_HISTSIZE to a large value has no penalty unless a large file is actually generated.

These variables are read at the time of saving, so can be altered within a session by the use of Sys.setenv.

history shows only unique matching lines if pattern is supplied.

The timestamp function writes a timestamp (or other message) into the history and echos it to the console. On platforms that do not support a history mechanism (where the mechanism does not yet support timestamps) only the console message is printed.

Note

If you want to save the history (almost) every session, you can put a call to savehistory() in .Last. See the examples.

Examples

```
## Not run:
.Last <- function()
    if(interactive()) try(savehistory("~/.Rhistory"))
## End(Not run)
```

---

select.list *Select Items from a List*

Description

Select item(s) from a character vector.

Usage

```
select.list(list, preselect = NULL, multiple = FALSE, title = NULL)
```

Arguments

list	character. A list of items.
preselect	a character vector, or NULL. If non-null and if the string(s) appear in the list, the item(s) are selected initially.
multiple	logical: can more than one item be selected?
title	optional character string for window title.

Details

Under the AQUA interface for MacOS X this brings up a modal dialog box with a (scrollable) list of items, which can be selected by the mouse.

Otherwise it displays a text list from which the user can choose by number(s). The multiple = FALSE case uses menu. Preselection is only supported for multiple = TRUE, where it is indicated by a "+" preceding the item.

Value

A character vector of selected items. If multiple is false and no item was selected (or Cancel was used), "" is returned. If multiple is true and no item was selected (or Cancel was used) then a character vector of length 0 is returned.

See Also

menu, tk_select.list for a graphical version using Tcl/Tk.

Examples

```
## Not run:
select.list(sort(.packages(all.available = TRUE)))
## End(Not run)
```

sessionInfo *Collect Information About the Current R Session*

Description

Print version information about R and attached or loaded packages.

Usage

```
sessionInfo(package=NULL)
## S3 method for class 'sessionInfo':
print(x, ...)
## S3 method for class 'sessionInfo':
toLatex(object, ...)
```

Arguments

package	a character vector naming installed packages. By default all attached packages are used.
x	an object of class "sessionInfo".
object	an object of class "sessionInfo".
...	currently not used.

See Also

R.version

Examples

```
sessionInfo()
toLatex(sessionInfo())
```

setRepositories *Select Package Repositories*

Description

Interact with the user to choose the package repositories to be used.

Usage

```
setRepositories(graphics = getOption("menu.graphics"), ind = NULL)
```

Arguments

graphics	Logical. If true and **tcltk** and an X server are available, use a Tk widget, or if under the AQUA interface use a MacOS X widget, otherwise use a text list in the console.
ind	NULL or a vector of integer indices, which have the same effect as if they were entered at the prompt for graphics=FALSE.

Details

The default list of known repositories is stored in the file '*R_HOME*/etc/repositories'. That file can be edited for a site, or a user can have a personal copy in '*HOME*/.R/repositories' which will take precedence.

The items that are preselected are those that are currently in `options("repos")` plus those marked as default in the list of known repositories.

This function requires the R session to be interactive unless `ind` is supplied.

Value

This function is invoked mainly for its side effect of updating `options("repos")`. It returns (invisibly) the previous `repos` options setting (as a `list` with component `repos`) or `NULL` if no changes were applied.

See Also

`chooseCRANmirror`, `install.packages`.

SHLIB *Build Shared Library for Dynamic Loading*

Description

Compile the given source files and then link all specified object files into a shared library which can be loaded into R using `dyn.load` or `library.dynam`.

Usage

```
R CMD SHLIB [options] [-o libname] files
```

Arguments

`files`	a list specifying the object files to be included in the shared library. You can also include the name of source files (for which the object files are automagically made from their sources) and library linking commands.
`libname`	the full name of the shared library to be built, including the extension (typically '.so' on Unix systems). If not given, the name of the library is taken from the first file.
`options`	Further options to control the processing. Use `R CMD SHLIB --help` for a current list.

Details

`R CMD SHLIB` is the mechanism used by `INSTALL` to compile source code in packages. Please consult section 'Creating shared objects' in the manual 'Writing R Extensions' for how to customize it (for example to add `cpp` flags and to add libraries to the link step) and for details of some of its quirks.

Items in `files` with extensions '.c', '.cpp', '.cc', '.C', '.f', '.f90', '.f95', '.m', '.M' and '.mm' are regarded as source files, and those with extension '.o' as object files. All other items are passed to the linker.

Note

Some binary distributions of R have `SHLIB` in a separate bundle, e.g., an `R-devel` RPM.

See Also

`COMPILE`, `dyn.load`, `library.dynam`.

The section on "Customizing compilation under Unix" in "R Administration and Installation" (see the 'doc/manual' subdirectory of the R source tree).

The 'R Installation and Administration' and 'Writing R Extensions' manuals.

Examples

```
## Not run:
R CMD SHLIB -o mylib.so a.f b.f -L/opt/acml3.5.0/gnu64/lib -lacml
## End(Not run)
```

---

stack *Stack or Unstack Vectors from a Data Frame or List*

Description

Stacking vectors concatenates multiple vectors into a single vector along with a factor indicating where each observation originated. Unstacking reverses this operation.

Usage

```
stack(x, ...)
## Default S3 method:
stack(x, ...)
## S3 method for class 'data.frame':
stack(x, select, ...)

unstack(x, ...)
## Default S3 method:
unstack(x, form, ...)
## S3 method for class 'data.frame':
unstack(x, form, ...)
```

Arguments

`x`	object to be stacked or unstacked
`select`	expression, indicating variables to select from a data frame
`form`	a two-sided formula whose left side evaluates to the vector to be unstacked and whose right side evaluates to the indicator of the groups to create. Defaults to `formula(x)` in `unstack.data.frame`.
`...`	further arguments passed to or from other methods.

Details

The `stack` function is used to transform data available as separate columns in a data frame or list into a single column that can be used in an analysis of variance model or other linear model. The `unstack` function reverses this operation.

Value

`unstack` produces a list of columns according to the formula `form`. If all the columns have the same length, the resulting list is coerced to a data frame.

`stack` produces a data frame with two columns

`values`	the result of concatenating the selected vectors in `x`
`ind`	a factor indicating from which vector in `x` the observation originated

Author(s)

Douglas Bates

See Also

`lm`, `reshape`

Examples

```
require(stats)
formula(PlantGrowth)          # check the default formula
pg <- unstack(PlantGrowth)    # unstack according to this formula
pg
stack(pg)                     # now put it back together
stack(pg, select = -ctrl)     # omitting one vector
```

`str` *Compactly Display the Structure of an Arbitrary R Object*

Description

Compactly display the internal **str**ucture of an R object, a diagnostic function and an alternative to `summary` (and to some extent, `dput`). Ideally, only one line for each 'basic' structure is displayed. It is especially well suited to compactly display the (abbreviated) contents of (possibly nested) lists. The idea is to give reasonable output for **any** R object. It calls `args` for (non-primitive) function objects.

`strOptions()` is a convenience function for setting `options(str = .)`, see the examples.

Usage

```
str(object, ...)

## S3 method for class 'data.frame':
str(object, ...)

## Default S3 method:
```

```
str(object, max.level = NA,
    vec.len  = strO$vec.len, digits.d = strO$digits.d,
    nchar.max = 128, give.attr = TRUE,
    give.head = TRUE, give.length = give.head,
    width = getOption("width"), nest.lev = 0,
    indent.str = paste(rep.int(" ", max(0, nest.lev + 1)),
                       collapse = ".."),
    comp.str="$ ", no.list = FALSE, envir = baseenv(),
    strict.width = strO$strict.width,
    formatNum = strO$formatNum, ...)

strOptions(strict.width = "no", digits.d = 3, vec.len = 4,
           formatNum = function(x, ...)
                       format(x, trim=TRUE, drop0trailing=TRUE, ...))
```

Arguments

`object`	any R object about which you want to have some information.
`max.level`	maximal level of nesting which is applied for displaying nested structures, e.g., a list containing sub lists. Default NA: Display all nesting levels.
`vec.len`	numeric (>= 0) indicating how many 'first few' elements are displayed of each vector. The number is multiplied by different factors (from .5 to 3) depending on the kind of vector. Defaults to the `vec.len` component of option `"str"` (see `options`) which defaults to 4.
`digits.d`	number of digits for numerical components (as for `print`). Defaults to the `digits.d` component of option `"str"` which defaults to 3.
`nchar.max`	maximal number of characters to show for `character` strings. Longer strings are truncated, see `longch` example below.
`give.attr`	logical; if `TRUE` (default), show attributes as sub structures.
`give.length`	logical; if `TRUE` (default), indicate length (as `[1:...]`).
`give.head`	logical; if `TRUE` (default), give (possibly abbreviated) mode/class and length (as `<type>[1:...]`).
`width`	the page width to be used. The default is the currently active `options("width")`; note that this has only a weak effect, unless `strict.width` is not `"no"`.
`nest.lev`	current nesting level in the recursive calls to `str`.
`indent.str`	the indentation string to use.
`comp.str`	string to be used for separating list components.
`no.list`	logical; if true, no 'list of ...' nor the class are printed.
`envir`	the environment to be used for *promise* (see `delayedAssign`) objects only.
`strict.width`	string indicating if the `width` argument's specification should be followed strictly, one of the values `c("no", "cut", "wrap")`. Defaults to the `strict.width` component of option `"str"` (see `options`) which defaults to `"no"` for back compatibility reasons; `"wrap"` uses `strwrap(*, width=width)` whereas `"cut"` cuts directly to `width`. Note that a small `vec.length` may be better than setting `strict.width = "wrap"`.
`formatNum`	a function such as `format` for formatting numeric vectors. It defaults to the `formatNum` component of option `"str"`, see "Usage" of `strOptions()` above, which is almost back compatible to R version $\leq$ 2.7.x, however, using `formatC` may be slightly better.

`...` potential further arguments (required for Method/Generic reasons).

Value

`str` does not return anything, for efficiency reasons. The obvious side effect is output to the terminal.

Author(s)

Martin Maechler ⟨maechler@stat.math.ethz.ch⟩ since 1990.

See Also

`ls.str` for *listing* objects with their structure; `summary`, `args`.

Examples

```
require(stats); require(grDevices); require(graphics)
## The following examples show some of 'str' capabilities
str(1:12)
str(ls)
str(args) #- more useful than  args(args) !
str(freeny)
str(str)
str(.Machine, digits.d = 20)
str( lsfit(1:9,1:9))
str( lsfit(1:9,1:9), max.level = 1)
str( lsfit(1:9,1:9), width = 60, strict.width = "cut")
str( lsfit(1:9,1:9), width = 60, strict.width = "wrap")
op <- options(); str(op) # save first;
                         # otherwise internal options() is used.
need.dev <-
  !exists(".Device") || is.null(.Device) || .Device == "null device"
{ if(need.dev) postscript()
  str(par())
  if(need.dev) graphics.off()
}
ch <- letters[1:12]; is.na(ch) <- 3:5
str(ch) # character NA's

nchar(longch <- paste(rep(letters,100), collapse=""))
str(longch)
str(longch, nchar.max = 52)

str(longch, strict.width = "wrap")

## Settings for narrow transcript :
op <- options(width = 60,
              str = strOptions(strict.width = "wrap"))
str(lsfit(1:9,1:9))
str(options())
## reset to previous:
options(op)


str(quote( { A+B; list(C,D) } ))
```

```
## S4 classes :
require(stats4)
x <- 0:10; y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
ll <- function(ymax=15, xh=6)
      -sum(dpois(y, lambda=ymax/(1+x/xh), log=TRUE))
fit <- mle(ll)
str(fit)
```

`summaryRprof` *Summarise Output of R Sampling Profiler*

Description

Summarise the output of the `Rprof` function to show the amount of time used by different R functions.

Usage

```
summaryRprof(filename = "Rprof.out", chunksize = 5000,
             memory=c("none","both","tseries","stats"),
             index=2, diff=TRUE, exclude=NULL)
```

Arguments

`filename`	Name of a file produced by `Rprof()`
`chunksize`	Number of lines to read at a time
`memory`	Summaries for memory information. See 'Details' below
`index`	How to summarize the stack trace for memory information. See 'Details' below.
`diff`	If `TRUE` memory summaries use change in memory rather than current memory
`exclude`	Functions to exclude when summarizing the stack trace for memory summaries

Details

This function is an alternative to `R CMD Rprof`. It provides the convenience of an all-R implementation but will be slower for large files.

As the profiling output file could be larger than available memory, it is read in blocks of `chunksize` lines. Increasing `chunksize` will make the function run faster if sufficient memory is available.

When called with `memory.profiling=TRUE`, the profiler writes information on three aspects of memory use: vector memory in small blocks on the R heap, vector memory in large blocks (from `malloc`), memory in nodes on the R heap. It also records the number of calls to the internal function `duplicate` in the time interval. `duplicate` is called by C code when arguments need to be copied. Note that the profiler does not track which function actually allocated the memory.

With `memory="both"` the change in total memory (truncated at zero) is reported in addition to timing data.

With `memory="tseries"` or `memory="stats"` the `index` argument specifies how to summarize the stack trace. A positive number specifies that many calls from the bottom of

the stack; a negative number specifies the number of calls from the top of the stack. With `memory="tseries"` the index is used to construct labels and may be a vector to give multiple sets of labels. With `memory="stats"` the index must be a single number and specifies how to aggregate the data to the maximum and average of the memory statistics. With both `memory="tseries"` and `memory="stats"` the argument `diff=TRUE` asks for summaries of the increase in memory use over the sampling interval and `diff=FALSE` asks for the memory use at the end of the interval.

Value

If `memory="none"`,a list with components

`by.self` Timings sorted by 'self' time

`by.total` Timings sorted by 'total' time

`sampling.time` Total length of profiling run

If `memory="both"` the same list but with memory consumption in Mb in addition to the timings.

If `memory="tseries"` a data frame giving memory statistics over time

If `memory="stats"` a `by` object giving memory statistics by function.

See Also

The chapter on "Tidying and profiling R code" in "Writing R Extensions" (see the 'doc/manual' subdirectory of the R source tree).

`Rprof`

`tracemem` traces copying of an object via the C function `duplicate`.

`Rprofmem` is a non-sampling memory use profiler.

`http://developer.r-project.org/memory-profiling.html`

Examples

```
## Not run:
## Rprof() is not available on all platforms
Rprof(tmp <- tempfile())
example(glm)
Rprof()
summaryRprof(tmp)
unlink(tmp)
## End(Not run)
```

`Sweave` *Automatic Generation of Reports*

Description

`Sweave` provides a flexible framework for mixing text and S code for automatic report generation. The basic idea is to replace the S code with its output, such that the final document only contains the text and the output of the statistical anlysis.

Usage

```
Sweave(file, driver = RweaveLatex(),
       syntax = getOption("SweaveSyntax"), ...)

Stangle(file, driver = Rtangle(),
        syntax = getOption("SweaveSyntax"), ...)
```

Arguments

`file`	Name of Sweave source file.
`driver`	The actual workhorse, see details below.
`syntax`	An object of class `SweaveSyntax` or a character string with its name. The default installation provides `SweaveSyntaxNoweb` and `SweaveSyntaxLatex`.
`...`	Further arguments passed to the driver's setup function.

Details

Automatic generation of reports by mixing word processing markup (like latex) and S code. The S code gets replaced by its output (text or graphs) in the final markup file. This allows a report to be re-generated if the input data change and documents the code to reproduce the analysis in the same file that also produces the report.

`Sweave` combines the documentation and code chunks together (or their output) into a single document. `Stangle` extracts only the code from the Sweave file creating a valid S source file (that can be run using `source`). Code inside `\Sexpr{}` statements is ignored by `Stangle`.

`Stangle` is just a frontend to `Sweave` using a simple driver by default, which discards the documentation and concatenates all code chunks the current S engine understands.

Hook Functions

Before each code chunk is evaluated, a number of hook functions can be executed. If `getOption("SweaveHooks")` is set, it is taken to be a collection of hook functions. For each logical option of a code chunk (`echo`, `print`, `...`) a hook can be specified, which is executed if and only if the respective option is `TRUE`. Hooks must be named elements of the list returned by `getOption("SweaveHooks")` and be functions taking no arguments. E.g., if option `"SweaveHooks"` is defined as `list(fig = foo)`, and `foo` is a function, then it would be executed before the code in each figure chunk. This is especially useful to set defaults for the graphical parameters in a series of figure chunks.

Note that the user is free to define new Sweave options and associate arbitrary hooks with them. E.g., one could define a hook function for option `clean` that removes all objects in the global environment. Then all code chunks with `clean = TRUE` would start operating on an empty workspace.

Syntax Definition

Sweave allows a very flexible syntax framework for marking documentation and text chunks. The default is a noweb-style syntax, as alternative a latex-style syntax can be used. See the user manual for details.

Author(s)

Friedrich Leisch

References

Friedrich Leisch: Dynamic generation of statistical reports using literate data analysis. In W. Härdle and B. Rönz, editors, Compstat 2002 - Proceedings in Computational Statistics, pages 575–580. Physika Verlag, Heidelberg, Germany, 2002. ISBN 3-7908-1517-9.

Friedrich Leisch: Sweave User Manual, 2008
http://www.stat.uni-muenchen.de/~leisch/Sweave

See Also

RweaveLatex, Rtangle

Examples

```
testfile <- system.file("Sweave", "Sweave-test-1.Rnw", package = "utils")

## enforce par(ask=FALSE)
options(device.ask.default=FALSE)

## create a LaTeX file
Sweave(testfile)

## This can be compiled to PDF by
## Not run: tools::texi2dvi("Sweave-test-1.tex", pdf=TRUE)
## or outside R by
## R CMD texi2dvi Sweave-test-1.tex
## which sets the appropriate TEXINPUTS path.

## create an S source file from the code chunks
Stangle(testfile)
## which can be sourced, e.g.
source("Sweave-test-1.R")
```

SweaveSyntConv *Convert Sweave Syntax*

Description

This function converts the syntax of files in Sweave format to another Sweave syntax definition.

Usage

```
SweaveSyntConv(file, syntax, output = NULL)
```

Arguments

file	Name of Sweave source file.
syntax	An object of class SweaveSyntax or a character string with its name giving the target syntax to which the file is converted.
output	Name of output file, default is to remove the extension from the input file and to add the default extension of the target syntax. Any directory names in file are also removed such that the output is created in the current working directory.

Author(s)

Friedrich Leisch

References

Friedrich Leisch: Sweave User Manual, 2008
http://www.stat.uni-muenchen.de/~leisch/Sweave

See Also

RweaveLatex, Rtangle

Examples

```
testfile <- system.file("Sweave", "Sweave-test-1.Rnw", package = "utils")

## convert the file to latex syntax
SweaveSyntConv(testfile, SweaveSyntaxLatex)

## and run it through Sweave
Sweave("Sweave-test-1.Stex")
```

---

toLatex *Converting R Objects to BibTeX or LaTeX*

Description

These methods convert R objects to character vectors with BibTeX or LaTeX markup.

Usage

```
toBibtex(object, ...)
toLatex(object, ...)
## S3 method for class 'Bibtex':
print(x, prefix="", ...)
## S3 method for class 'Latex':
print(x, prefix="", ...)
```

Arguments

object	object of a class for which a toBibtex or toLatex method exists.
x	object of class "Bibtex" or "Latex".
prefix	a character string which is printed at the beginning of each line, mostly used to insert whitespace for indentation.
...	currently not used in the print methods.

Details

Objects of class "Bibtex" or "Latex" are simply character vectors where each element holds one line of the corresponding BibTeX or LaTeX file.

See Also

citEntry and sessionInfo for examples

---

txtProgressBar *Text Progress Bar*

Description

Text progress bar in the R console.

Usage

```
txtProgressBar(min = 0, max = 1, initial = 0, char = "=",
               width = NA, title, label, style = 1)

getTxtProgressBar(pb)
setTxtProgressBar(pb, value, title = NULL, label = NULL)
## S3 method for class 'txtProgressBar':
close(con, ...)
```

Arguments

min, max	(finite) numeric values for the extremes of the progress bar.
initial, value	initial or new value for the progress bar.
char	the character (or character string) to form the progress bar.
width	the width of the progress bar, as a multiple of the width of char. If NA, the default, the number of characters is that which fits into getOption("width").
style	the 'style' of the bar – see 'Details'.
pb, con	an object of class "txtProgressBar".
title, label	ignored, for compatibility with other progress bars.
...	for consistency with the generic.

Details

txtProgressBar will display a progress bar on the R console via a text representation.

setTxtProgessBar will update the value. Missing (NA) and out-of-range values of value will be (silently) ignored.

The progress bar should be closed when finished with: this outputs the final newline character.

style = 1 and style = 2 just shows a line of char. They differ in that style = 2 redraws the line each time, which is useful if other code might be writing to the R console. style = 3 marks the end of the range by | and gives a percentage to the right of the bar.

Value

For txtProgressBar an object of class "txtProgressBar".

For getTxtProgressBar and setTxtProgressBar, a length-one numeric vector giving the previous value (invisibly for setTxtProgressBar).

Note

Using `style` 2 or 3 or reducing the value with `style = 1` uses `\r` to return to the left margin – the interpretation of carriage return is up to the terminal or console in which R is running.

See Also

`tkProgressBar`

Examples

```
 # slow
testit <- function(x = sort(runif(20)), ...)
{
    pb <- txtProgressBar(...)
    for(i in c(0, x, 1)) {Sys.sleep(0.5); setTxtProgressBar(pb, i)}
    Sys.sleep(1)
    close(pb)
}
testit()
testit(runif(10))
testit(style=3)
```

---

type.convert *Type Conversion on Character Variables*

Description

Convert a character vector to logical, integer, numeric, complex or factor as appropriate.

Usage

```
type.convert(x, na.strings = "NA", as.is = FALSE, dec = ".")
```

Arguments

`x`	a character vector.
`na.strings`	a vector of strings which are to be interpreted as `NA` values. Blank fields are also considered to be missing values in logical, integer, numeric or complex vectors.
`as.is`	logical. See 'Details'.
`dec`	the character to be assumed for decimal points.

Details

This is principally a helper function for `read.table`. Given a character vector, it attempts to convert it to logical, integer, numeric or complex, and failing that converts it to factor unless `as.is = TRUE`. The first type that can accept all the non-missing values is chosen.

Vectors which are entirely missing values are converted to logical, since `NA` is primarily logical.

Since this is a helper function, the caller should always pass an appropriate value of `as.is`.

Value

A vector of the selected class, or a factor.

See Also

read.table

---

update.packages *Download Packages from CRAN-like repositories*

Description

These functions can be used to automatically compare the version numbers of installed packages with the newest available version on the repositories and update outdated packages on the fly.

Usage

```
update.packages(lib.loc = NULL, repos = getOption("repos"),
                contriburl = contrib.url(repos, type),
                method, instlib = NULL,
                ask = TRUE, available = NULL,
                oldPkgs = NULL, ...,
                checkBuilt = FALSE, type = getOption("pkgType"))

available.packages(contriburl = contrib.url(getOption("repos")),
                   method, fields = NULL)

old.packages(lib.loc = NULL, repos = getOption("repos"),
             contriburl = contrib.url(repos),
             instPkgs = installed.packages(lib.loc = lib.loc),
             method, available = NULL, checkBuilt = FALSE)

new.packages(lib.loc = NULL, repos = getOption("repos"),
             contriburl = contrib.url(repos, type),
             instPkgs = installed.packages(lib.loc = lib.loc),
             method, available = NULL, ask = FALSE, ...,
             type = getOption("pkgType"))

download.packages(pkgs, destdir, available = NULL,
                  repos = getOption("repos"),
                  contriburl = contrib.url(repos, type),
                  method, type = getOption("pkgType"), ...)

install.packages(pkgs, lib, repos = getOption("repos"),
                 contriburl = contrib.url(repos, type),
                 method, available = NULL, destdir = NULL,
                 installWithVers = FALSE, dependencies = NA,
                 type = getOption("pkgType"),
                 configure.args = character(0),
                 clean = FALSE, ...)

contrib.url(repos, type = getOption("pkgType"))
```

Arguments

lib.loc	character vector describing the location of R library trees to search through (and update packages therein).
repos	character vector, the base URL(s) of the repositories to use, i.e., the URL of the CRAN master such as "http://cran.r-project.org" or its Statlib mirror, "http://lib.stat.cmu.edu/R/CRAN". Can be NULL to install from local files ('.tar.gz' for source packages).
contriburl	URL(s) of the contrib sections of the repositories. Use this argument only if your repository mirror is incomplete, e.g., because you burned only the 'contrib' section on a CD. Overrides argument repos. As repos, can also be NULL to install from local files.
method	Download method, see download.file.
pkgs	character vector of the short names of packages/bundles whose current versions should be downloaded from the repositories. If repos = NULL, a character vector of file paths of '.tar.gz' files. These can be source archives or binary package/bundle archive files (as created by R CMD build --binary). Tilde-expansion will be done on the file paths. If this is a zero-length character vector, a listbox of available packages (including those contained in bundles) is presented where possible.
destdir	directory where downloaded packages are stored.
available	an object listing packages available at the repositories as returned by available.packages.
lib	character vector giving the library directories where to install the packages. Recycled as needed. If missing, defaults to .libPaths()[1].
ask	logical indicating whether to ask user before packages are actually downloaded and installed, or the character string "graphics", which brings up a widget to allow the user to (de-)select from the list of packages which could be updated. The latter only works on systems with a GUI version of select.list, and is otherwise equivalent to ask = TRUE.
installWithVers	If TRUE, will invoke the install of the package such that it can be referenced by package version.
checkBuilt	If TRUE, a package built under an earlier minor version of R is considered to be 'old'.
instlib	character string giving the library directory where to install the packages.
dependencies	logical indicating to also install uninstalled packages on which these packages depend/suggest/import (and so on recursively). Not used if repos = NULL. Can also be a character vector, a subset of c("Depends", "Imports", "Suggests"). Note that only if the dependent package is *installed* is checked, not if it meets the version requirements. Only supported if lib is of length one (or missing), so it is unambiguous where to install the dependent packages. The default, NA, means c("Depends", "Imports") if lib is unambiguous, and FALSE otherwise.
configure.args	a character vector or a named list. If a character vector with no names is supplied, the elements are concatenated into a single string (separated by a space) and used as the value for the --configure-args flag in the call to R CMD

INSTALL. If the character vector has names, these are assumed to identify values for --configure-args for individual packages. This allows one to specify settings for an entire collection of packages which will be used if any of those packages are installed. These settings can therefore be reused and act as default settings. A named list can be used also to the same effect, and that allows multi-element character strings for each package which are concatenated to a single string to be used as the value for --configure-args.

oldPkgs — if specified as non-NULL, update.packages() only considers these packages for updating.

instPkgs — by default all installed packages, installed.packages(lib.loc=lib.loc). A subset can be specified; currently this must be in the same (character matrix) format as returned by installed.packages().

... — (for update.packages). Arguments such as destdir, installWithVers and dependencies to be passed to install.packages.

(for new.packages). Arguments such as destdir and dependencies to be passed to install.packages.

(for install.packages and download.packages) arguments to be passed to download.file.

type — character, indicating the type of package to download and install. Possible values are "source" (the default except under the CRAN Mac OS X build), "mac.binary" and "win.binary" (which can be downloaded but not installed).

clean — a logical value indicating whether to specify to add the --clean flag to the call to R CMD INSTALL. This is sometimes used to perform additional operations at the end of the package installation in addition to removing intermediate files.

fields — a character vector giving the fields to extract from the PACKAGES file(s) in addition to the default ones, or NULL (default). Unavailable fields result in NA values.

Details

All of these functions work with the names of a package or bundle (and not the component packages of a bundle, except for install.packages if the repository provides the necessary information).

available.packages returns a matrix of details corresponding to packages/bundles currently available at one or more repositories. The current list of packages is downloaded over the internet (or copied from a local mirror). It returns only packages whose version requirements are met by the running version of R.

old.packages compares the information from available.packages with that from instPkgs (installed.packages by default) and reports installed packages/bundles that have newer versions on the repositories or, if checkBuilt = TRUE, that were built under an earlier minor version of R (for example built under 2.0.x when running R 2.1.1).

new.packages does the same comparison but reports uninstalled packages/bundles that are available at the repositories. (It checks the front-end package **gnomeGUI** separately.) It will give warnings about incompletely installed bundles (provided the information is available) and bundles whose contents has changed. If ask != FALSE it asks which packages should be installed in the first element of lib.loc. NB: versioned installs are not installs of a named package.

download.packages takes a list of package/bundle names and a destination directory, downloads the newest versions and saves them in destdir. If the list of available packages is not

given as argument, it is obtained from repositories. If a repository is local, i.e. the URL starts with "file:", then the packages are not downloaded but used directly. Both "file:" and "file:///" are allowed as prefixes to a file path. Use the latter only for URLs: see url for their interpretation. (Other forms of "file://" URLs are not supported.)

The main function of the set is update.packages. First a list of all packages/bundles found in lib.loc is created and compared with those available at the repositories. If ask = TRUE (the default) packages/bundles with a newer version are reported and for each one the user can specify if it should be updated. If so, the package sources are downloaded from the repositories and installed in the respective library path (or instlib if specified) using the R INSTALL mechanism.

install.packages can be used to install new packages/bundles. It takes a vector of names and a destination library, downloads the packages from the repositories and installs them. (If the library is omitted it defaults to the first directory in .libPaths(), with a warning if there is more than one.) If lib is omitted or is of length one and is not a (group) writeable directory, the code offers to create a personal library tree (the first element of Sys.getenv("R_LIBS_USER")) and install there.

If a repository is used (rather than local '.tar.gz' files), an attempt is made to install the packages in an order that respects their dependencies. This does assume that all the entries in lib are on the default library path for installs (set by R_LIBS).

contrib.url adds the appropriate type-specific path within a repository to each URL in repos.

For install.packages, destdir is the directory to which packages will be downloaded. If it is NULL (the default) a directory downloaded_packages of the session temporary directory will be used (and the files will be deleted at the end of the session).

If repos or contriburl is a vector of length greater than one, the newest version of the package compatible with this version of R is fetched from the first repository on the list within which it is found.

Value

For available.packages, a matrix with one row per package/bundle, row names the package names and column names "Package", "Version", "Priority", "Bundle", "Depends", "Imports", "Suggests" "Contains" and "Repository". Additional columns can be specified using the fields argument.

For old.packages, NULL or a matrix with one row per package/bundle, row names the package names and column names "Package", "LibPath", "Installed" (the version), "Built" (the version built under), "ReposVer" and "Repository".

For new.packages a character vector of package/bundle names, *after* any have been installed.

For download.packages, a two-column matrix of names and destination file names, for those packages/bundles successfully downloaded. If packages are not available or there is a problem with the download, suitable warnings are given.

install.packages and update.packages have no return value.

Warning

Take care when using dependencies with update.packages, for it is unclear where new dependencies should be installed. The current implementation will only allow it if all the packages to be updated are in a single library, when that library will be used.

You are advised to run update.packages before install.packages to ensure that any installed dependencies have their latest versions.

Note

Some binary distributions of R have `INSTALL` in a separate bundle, e.g. an `R-devel` RPM. `install.packages` will give an error if called on such a system.

See Also

`installed.packages`, `remove.packages`

See `download.file` for how to handle proxies and other options to monitor file transfers.

`INSTALL`, `REMOVE`, `library`, `.packages`, `read.dcf`

The 'R Installation and Administration' manual for how to set up a repository.

Examples

```
## Not run:
install.packages(
    c("XML_0.99-5.tar.gz",
      "../../Interfaces/Perl/RSPerl_0.8-0.tar.gz"),
    repos = NULL,
    configure.args = c(XML = '--with-xml-config=xml-config',
                       RSPerl = "--with-modules='IO Fcntl'"))
## End(Not run)
```

`url.show` *Display a text URL*

Description

Extension of `file.show` to display text files from a remote server.

Usage

```
url.show(url, title = url, file = tempfile(),
         delete.file = TRUE, method, ...)
```

Arguments

`url`	The URL to read from.
`title`	Title for the browser.
`file`	File to copy to.
`delete.file`	Delete the file afterwards?
`method`	File transfer method: see `download.file`
`...`	Arguments to pass to `file.show`.

See Also

`url`, `file.show`, `download.file`

Examples

```
## Not run: url.show("http://lib.stat.cmu.edu/datasets/csb/ch3a.txt")
```

`URLencode` *Encode or Decode a (partial) URL*

Description

Functions to encode or decode characters in URLs.

Usage

```
URLencode(URL, reserved = FALSE)
URLdecode(URL)
```

Arguments

`URL`	A character string.
`reserved`	should reserved characters be encoded? See 'Details'.

Details

Characters in a URL other than the English alphanumeric characters and `$ - _ . + ! * ' ( ) ,` should be encoded as `%` plus a two-digit hexadecimal representation, and any single-byte character can be so encoded. (Multi-byte characters are encoded as byte-by-byte.)

In addition, `; / ? : @ = &` are reserved characters, and should be encoded unless used in their reserved sense, which is scheme specific. The default in `URLencode` is to leave them alone, which is appropriate for `file://` URLs, but probably not for `http://` ones.

Value

A character string.

References

RFC1738, http://www.rfc-editor.org/rfc/rfc1738.txt

Examples

```
(y <- URLencode("a url with spaces and / and @"))
URLdecode(y)
(y <- URLencode("a url with spaces and / and @", reserved=TRUE))
URLdecode(y)
URLdecode("ab%20cd")
```

utils-deprecated *Deprecated Functions in Package utils*

Description

These functions are provided for compatibility with older versions of R only, and may be defunct as soon as of the next release.

Usage

```
CRAN.packages(CRAN = getOption("repos"), method,
              contriburl = contrib.url(CRAN))
```

Arguments

CRAN	character, an earlier way to specify a repository.
method	Download method, see download.file.
contriburl	URL(s) of the contrib section of the repositories. Use this argument only if your CRAN mirror is incomplete, e.g., because you burned only the 'contrib' section on a CD. Overrides argument repos.

See Also

Deprecated, Defunct

View *Invoke a Data Viewer*

Description

Invoke a spreadsheet-style data viewer on a matrix-like R object.

Usage

```
View(x, title)
```

Arguments

x	an R object which can be coerced to a data frame with non-zero numbers of rows and columns.
title	title for viewer window. Defaults to name of x.

Details

Object `x` is coerced (if possible) to a data frame, and all non-numeric columns are then coerced to character. The object is then viewed in a spreadsheet-like data viewer, a read-only version of `data.entry`.

If there are row names on the data frame that are not `1:nrow`, they are displayed in a separate first column called `row.names`.

Objects with zero columns or zero rows are not accepted.

The array of cells can be navigated by the cursor keys and Home, End, Page Up and Page Down (where supported by X11) as well as Enter and Tab.

Value

Invisible `NULL`. The functions puts up a window and returns immediately: the window can be closed via its controls or menus.

See Also

`edit.data.frame`, `data.entry`.

`vignette` *View or List Vignettes*

Description

View a specified vignette, or list the available ones.

Usage

```
vignette(topic, package = NULL, lib.loc = NULL, all = TRUE)

## S3 method for class 'vignette':
print(x, ...)
## S3 method for class 'vignette':
edit(name, ...)
```

Arguments

`topic`	a character string giving the (base) name of the vignette to view. If omitted, all vignettes from all installed packages are listed.
`package`	a character vector with the names of packages to search through, or `NULL` in which "all" packages (as defined by argument `all`) are searched.
`lib.loc`	a character vector of directory names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known.
`all`	logical; if `TRUE` search all available packages in the library trees specified by `lib.loc`, and if `FALSE`, search only attached packages.
`x, name`	Object of class `vignette`.
`...`	Ignored by the `print` method, passed on to `file.edit` by the `edit` method.

Details

Function `vignette` returns an object of the same class, the print method opens a viewer for it. Currently, only PDF versions of vignettes can be viewed. The program specified by the `pdfviewer` option is used for this. If several vignettes have PDF versions with base name identical to `topic`, the first one found is used.

If no topics are given, all available vignettes are listed. The corresponding information is returned in an object of class `"packageIQR"`.

The `edit` method extracts the R code from the vignette to a temporary file and opens the file in an editor (see `edit`). This makes it very easy to execute the commands line by line, modify them in any way you want to help you test variants, etc.. An alternative way of extracting the R code from the vignette is to run `Stangle` on the source code of the vignette, see the examples below.

Examples

```
## List vignettes from all *attached* packages
vignette(all = FALSE)

## List vignettes from all *installed* packages (can take a long time!):
vignette(all = TRUE)

## Not run:
## Open the grid intro vignette
vignette("grid")

## The same
v1 <- vignette("grid")
print(v1)

## Now let us have a closer look at the code
edit(v1)

## An alternative way of extracting the code,
## R file is written to current working directory
Stangle(v1$file)

## A package can have more than one vignette (package grid has several):
vignette(package="grid")
vignette("rotated")
## The same, but without searching for it:
vignette("rotated", package="grid")
## End(Not run)
```

---

withVisible *Return both a value and its visibility*

Description

This function evaluates an expression, returning it in a two element list containing its value and a flag showing whether it would automatically print.

Usage

```
withVisible(x)
```

Arguments

x	An expression to be evaluated.

Details

The argument is evaluated in the caller's context.

Value

value	The value of x after evaluation.
visible	logical; whether the value would auto-print.

See Also

invisible, eval

Examples

```
x <- 1
withVisible(x <- 1)
x
withVisible(x)

# Wrap the call in evalq() for special handling

df <- data.frame(a=1:5, b=1:5)
evalq(withVisible(a + b), envir=df)
```

---

write.table *Data Output*

Description

write.table prints its required argument x (after converting it to a data frame if it is not one nor a matrix) to a file or connection.

Usage

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"))

write.csv(...)
write.csv2(...)
```

Arguments

x	the object to be written, preferably a matrix or data frame. If not, it is attempted to coerce x to a data frame.
file	either a character string naming a file or a connection open for writing. "" indicates output to the console.
append	logical. Only relevant if file is a character string. If TRUE, the output is appended to the file. If FALSE, any existing file of the name is destroyed.
quote	a logical value (TRUE or FALSE) or a numeric vector. If TRUE, any character or factor columns will be surrounded by double quotes. If a numeric vector, its elements are taken as the indices of columns to quote. In both cases, row and column names are quoted if they are written. If FALSE, nothing is quoted.
sep	the field separator string. Values within each row of x are separated by this string.
eol	the character(s) to print at the end of each line (row). For example, eol="\r\n" will produce Windows' line endings on a Unix-alike OS, and eol="\r" will produce files as expected by MacOS Excel 2004.
na	the string to use for missing values in the data.
dec	the string to use for decimal points in numeric or complex columns: must be a single character.
row.names	either a logical value indicating whether the row names of x are to be written along with x, or a character vector of row names to be written.
col.names	either a logical value indicating whether the column names of x are to be written along with x, or a character vector of column names to be written. See the section on 'CSV files' for the meaning of col.names = NA.
qmethod	a character string specifying how to deal with embedded double quote characters when quoting strings. Must be one of "escape" (default), in which case the quote character is escaped in C style by a backslash, or "double", in which case it is doubled. You can specify just the initial letter.
...	arguments to write.table: col.names, sep, dec and qmethod cannot be altered.

Details

If the table has no columns the rownames will be written only if row.names=TRUE, and *vice versa*.

Real and complex numbers are written to the maximal possible precision.

If a data frame has matrix-like columns these will be converted to multiple columns in the result (*via* as.matrix) and so a character col.names or a numeric quote should refer to the columns in the result, not the input. Such matrix-like columns are unquoted by default.

Any columns in a data frame which are lists or have a class (e.g. dates) will be converted by the appropriate as.character method: such columns are unquoted by default. On the other hand, any class information for a matrix is discarded and non-atomic (e.g. list) matrices are coerced to character.

Only columns which have been converted to character will be quoted if specified by quote.

The dec argument only applies to columns that are not subject to conversion to character because they have a class or are part of a matrix-like column (or matrix), in particular to columns protected by I(). Use options("OutDec") to control such conversions.

In almost all cases the conversion of numeric quantities is governed by the option `"scipen"` (see `options`), but with the internal equivalent of `digits=15`. For finer control, use `format` to make a character matrix/data frame, and call `write.table` on that.

These functions check for a user interrupt every 1000 lines of output.

If `file` is not open for writing, an attempt is made to open it and then close it after use.

CSV files

By default there is no column name for a column of row names. If `col.names = NA` and `row.names = TRUE` a blank column name is added, which is the convention used for CSV files to be read by spreadsheets.

`write.csv` and `write.csv2` provide convenience wrappers for writing CSV files. They set `sep`, `dec` and `qmethod`, and `col.names` to `NA` if `row.names = TRUE` and `TRUE` otherwise.

`write.csv` uses `"."` for the decimal point and a comma for the separator.

`write.csv2` uses a comma for the decimal point and a semicolon for the separator, the Excel convention for CSV files in some Western European locales.

These wrappers are deliberately inflexible: they are designed to ensure that the correct conventions are used to write a valid file. Attempts to change `col.names`, `sep`, `dec` or `qmethod` are ignored, with a warning.

Note

`write.table` can be slow for data frames with large numbers (hundreds or more) of columns: this is inevitable as each column could be of a different class and so must be handled separately. If they are all of the same class, consider using a matrix instead.

See Also

The 'R Data Import/Export' manual.

`read.table`, `write`.

`write.matrix` in package **MASS**.

Examples

```
## Not run:
## To write a CSV file for input to Excel one might use
x <- data.frame(a = I("a \" quote"), b = pi)
write.table(x, file = "foo.csv", sep = ",", col.names = NA,
            qmethod = "double")
## and to read this file back into R one needs
read.table("foo.csv", header = TRUE, sep = ",", row.names = 1)
## NB: you do need to specify a separator if qmethod = "double".

### Alternatively
write.csv(x, file = "foo.csv")
read.csv("foo.csv", row.names = 1)
## or without row names
write.csv(x, file = "foo.csv", row.names = FALSE)
read.csv("foo.csv")
## End(Not run)
```

`zip.file.extract` *Extract File from a Zip Archive*

Description

This will extract the file named `file` from the zip archive, if possible, and write it in a temporary location.

Usage

```
zip.file.extract(file, zipname = "R.zip",
                 unzip = getOption("unzip"))
```

Arguments

`file`	A file name. (If a path is given, see 'Note'.)
`zipname`	The file name (not path) of a `zip` archive, including the `".zip"` extension if required.
`unzip`	character string: the method to be used, an empty string indicates `"internal"`.

Details

All platforms support an `"internal"` unzip: this is the default under Windows and the fall-back under Unix if no `unzip` program was found during configuration and `R_UNZIPCMD` is not set.

The file will be extracted if it is in the archive and any required `unzip` utility is available. It will be extracted to the directory given by `tempdir`, overwriting any existing file of that name.

Value

The name of the original or extracted file. Success is indicated by returning a different name.

Note

The `"internal"` method is very simple, and will not set file dates.

This is a helper function for `help`, `example` and `data`. As such, it handles file paths in an unusual way. Any path component of `zipname` is ignored, and the path to `file` is used only to determine the directory within which to find `zipname`.

Index

! *(Logic)*, 246
!= *(Comparison)*, 67
∗Topic **IO**
rcompgen, 1501
∗Topic **NA**
complete.cases, 1027
factor, 150
NA, 276
na.action, 1188
na.fail, 1190
naprint, 1191
naresid, 1191
∗Topic **algebra**
backsolve, 33
chol, 57
chol2inv, 59
colSums, 64
crossprod, 84
eigen, 126
matrix, 263
qr, 325
QR.Auxiliaries, 327
solve, 397
svd, 435
∗Topic **aplot**
abline, 659
arrows, 661
Axis, 664
axis, 665
box, 673
bxp, 677
contour, 683
coplot, 686
filled.contour, 691
frame, 695
grid, 696
Hershey, 603
image, 704
Japanese, 607
legend, 708
lines, 712
matplot, 715
mtext, 720
persp, 732
plot.window, 748
plot.xy, 749
plotmath, 619
points, 750
polygon, 753
rect, 755
rect.hclust, 1287
rug, 757
screen, 758
segments, 760
symbols, 772
text, 775
title, 777
xspline, 779
∗Topic **arith**
all.equal, 10
approxfun, 983
Arithmetic, 17
colSums, 64
cumsum, 85
diff, 109
Extremes, 148
findInterval, 159
gl, 192
matmult, 262
ppoints, 1249
prod, 322
range, 338
roman, 1521
Round, 365
sign, 392
sort, 398
sum, 432
tabulate, 456
∗Topic **array**
addmargins, 967
aggregate, 969
aperm, 13
apply, 15
array, 19
backsolve, 33
cbind, 50

cbind2, 875
chol, 57
chol2inv, 59
col, 62
colSums, 64
contrast, 1031
cor, 1036
crossprod, 84
data.matrix, 92
det, 105
diag, 108
dim, 112
dimnames, 113
drop, 118
eigen, 126
expand.grid, 138
Extract, 140
Extract.data.frame, 144
isSymmetric, 219
kronecker, 222
lm.fit, 1146
lower.tri, 249
margin.table, 255
mat.or.vec, 256
matmult, 262
matplot, 715
matrix, 263
maxCol, 265
merge, 270
nrow, 285
outer, 305
prop.table, 323
qr, 325
QR.Auxiliaries, 327
row, 368
row/colnames, 370
scale, 376
slice.index, 395
svd, 435
sweep, 436
t, 453

∗Topic **attribute**
attr, 30
attributes, 31
call, 45
comment, 66
length, 227
mode, 274
name, 277
names, 278
NULL, 290
numeric, 290
structure, 426
typeof, 478
which, 493

∗Topic **category**
aggregate, 969
by, 43
cut, 86
Extract.factor, 146
factor, 150
ftable, 1094
ftable.formula, 1096
gl, 192
interaction, 208
levels, 228
loglin, 1157
nlevels, 282
plot.table, 747
read.ftable, 1285
split, 405
table, 454
tapply, 457
xtabs, 1396

∗Topic **character**
abbreviate, 6
agrep, 7
char.expand, 52
character, 53
charmatch, 55
chartr, 56
delimMatch, 1406
Encoding, 129
format, 165
format.info, 170
formatC, 172
gettext, 190
glob2rx, 1465
grep, 193
iconv, 201
make.names, 251
make.unique, 252
nchar, 281
paste, 309
pmatch, 310
regex, 355
sprintf, 407
sQuote, 410
strsplit, 423
strtrim, 425
strwidth, 769
strwrap, 427
substr, 431
symnum, 1357

utf8Conversion, 487
∗Topic **chron**
as.POSIX∗, 23
axis.POSIXct, 667
cut.POSIXt, 88
Dates, 94
DateTimeClasses, 95
difftime, 110
format.Date, 168
hist.POSIXt, 700
Ops.Date, 296
rep, 360
round.POSIXt, 367
seq.Date, 384
seq.POSIXt, 385
strptime, 419
Sys.time, 448
timezones, 466
weekdays, 492
∗Topic **classes**
as, 866
as.data.frame, 20
BasicClasses, 870
callGeneric, 871
callNextMethod, 873
canCoerce, 875
character, 53
class, 60
Classes, 877
classRepresentation-class, 880
data.class, 89
data.frame, 90
Documentation, 881
dotsMethods, 883
double, 115
environment-class, 885
findClass, 886
findMethods, 888
fixPre1.8, 889
genericFunction-class, 890
GenericFunctions, 891
getClass, 895
getMethod, 896
integer, 207
is, 903
is.object, 214
is.recursive, 216
is.single, 217
isSealedMethod, 908
language-class, 909
LinearMethodsList-class, 910
logical, 248
makeClassRepresentation, 911
MethodDefinition-class, 913
Methods, 914
MethodsList-class, 918
MethodWithNext-class, 919
new, 920
numeric, 290
ObjectsWithPackage-class, 922
promptClass, 923
raw, 341
rawConversion, 344
real, 352
representation, 925
row.names, 369
S3Part, 927
SClassExtension-class, 932
setClass, 933
setClassUnion, 936
setMethod, 942
signature-class, 953
slot, 954
StructureClasses, 956
TraceClasses, 957
validObject, 958
vector, 488
∗Topic **cluster**
as.hclust, 1000
cophenetic, 1035
cutree, 1043
dist, 1059
hclust, 1108
identify.hclust, 1118
kmeans, 1133
rect.hclust, 1287
∗Topic **color**
col2rgb, 580
colorRamp, 581
colors, 583
convertColor, 585
gray, 599
gray.colors, 600
hcl, 601
hsv, 606
make.rgb, 608
palette, 611
Palettes, 612
rgb, 641
rgb2hsv, 642
∗Topic **complex**
complex, 69
∗Topic **connection**

cat, 48
connections, 74
dput, 117
dump, 119
gzcon, 198
parse, 307
pushBack, 324
rawConnection, 343
read.00Index, 1417
read.DIF, 1506
read.fortran, 1508
read.fwf, 1509
read.table, 1511
readBin, 346
readChar, 349
readLines, 351
scan, 377
seek, 381
showConnections, 389
sink, 393
socketSelect, 396
source, 400
textConnection, 464
write, 497
writeLines, 498
∗Topic **datagen**
simulate, 1307
∗Topic **datasets**
ability.cov, 501
airmiles, 502
AirPassengers, 503
airquality, 504
anscombe, 505
attenu, 506
attitude, 507
austres, 508
beavers, 508
BJsales, 509
BOD, 510
cars, 511
charsets, 1400
ChickWeight, 512
chickwts, 513
CO2, 514
co2, 515
crimtab, 515
data, 1445
discoveries, 517
DNase, 518
esoph, 519
euro, 520
eurodist, 521
EuStockMarkets, 521
faithful, 522
Formaldehyde, 523
freeny, 524
HairEyeColor, 525
Harman23.cor, 526
Harman74.cor, 526
Indometh, 527
infert, 528
InsectSprays, 529
iris, 529
islands, 531
JohnsonJohnson, 531
LakeHuron, 532
lh, 532
LifeCycleSavings, 533
Loblolly, 534
longley, 534
lynx, 535
morley, 536
mtcars, 537
nhtemp, 537
Nile, 538
nottem, 539
occupationalStatus, 540
Orange, 541
OrchardSprays, 542
PlantGrowth, 543
precip, 543
presidents, 544
pressure, 545
Puromycin, 545
quakes, 547
randu, 547
rivers, 548
rock, 549
sleep, 549
stackloss, 550
state, 551
sunspot.month, 552
sunspot.year, 553
sunspots, 553
swiss, 554
Theoph, 555
Titanic, 556
ToothGrowth, 557
treering, 558
trees, 559
UCBAdmissions, 560
UKDriverDeaths, 561
UKgas, 562
UKLungDeaths, 563

USAccDeaths, 563
USArrests, 564
USJudgeRatings, 564
USPersonalExpenditure, 565
uspop, 566
VADeaths, 566
volcano, 567
warpbreaks, 568
women, 569
WorldPhones, 569
WWWusage, 570
∗Topic **data**
apropos, 1428
as.environment, 21
assign, 26
assignOps, 27
attach, 28
autoload, 32
bquote, 40
delayedAssign, 101
deparse, 102
detach, 106
environment, 130
eval, 134
exists, 136
force, 160
get, 182
getAnywhere, 1462
getFromNamespace, 1463
getS3method, 1465
libPaths, 230
library, 231
library.dynam, 235
ns-load, 288
search, 380
substitute, 429
sys.parent, 442
with, 495
zpackages, 499
∗Topic **debugging**
recover, 1515
srcfile, 412
trace, 469
∗Topic **design**
contrast, 1031
contrasts, 1032
TukeyHSD, 1377
∗Topic **device**
.Device, 1
cairo, 576
dev.interactive, 587
dev.xxx, 588
dev2, 590
Devices, 594
embedFonts, 595
grDevices-package, 573
pdf, 613
pdf.options, 617
pictex, 618
png, 623
postscript, 627
postscriptFonts, 632
ps.options, 635
quartz, 636
quartzFonts, 638
recordGraphics, 639
screen, 758
Type1Font, 645
x11, 647
X11Fonts, 650
xfig, 651
∗Topic **distribution**
bandwidth, 1002
Beta, 1005
Binomial, 1008
birthday, 1013
Cauchy, 1018
chisq.test, 1019
Chisquare, 1022
density, 1051
Exponential, 1070
FDist, 1080
fivenum, 1087
GammaDist, 1097
Geometric, 1100
hist, 697
Hypergeometric, 1117
IQR, 1126
Logistic, 1155
Lognormal, 1159
Multinomial, 1187
NegBinomial, 1192
Normal, 1209
Poisson, 1240
ppoints, 1249
qqnorm, 1279
r2dtable, 1284
Random, 332
Random.user, 336
sample, 372
SignRank, 1306
stem, 766
TDist, 1361
Tukey, 1376

Uniform, 1379
Weibull, 1386
Wilcoxon, 1392
∗Topic **documentation**
apropos, 1428
args, 16
browseVignettes, 1433
buildVignettes, 1399
checkTnF, 1403
checkVignettes, 1404
codoc, 1405
data, 1445
Defunct, 100
demo, 1451
Deprecated, 105
Documentation, 881
example, 1457
help, 1468
help.search, 1473
help.start, 1475
NotYet, 284
NumericConstants, 292
prompt, 1497
promptData, 1499
promptPackage, 1500
QC, 1414
Quotes, 330
Rdindex, 1415
Rdutils, 1416
readNEWS, 1418
Reserved, 362
RShowDoc, 1525
RSiteSearch, 1526
str, 1536
Syntax, 438
tools-package, 1399
undoc, 1421
vignette, 1553
∗Topic **dplot**
absolute.size, 784
approxfun, 983
arrow, 785
axTicks, 669
boxplot.stats, 574
clip, 682
cm, 579
col2rgb, 580
colors, 583
contourLines, 584
convertNative, 785
convertXY, 685
convolve, 1033
dataViewport, 786
dev.size, 588
devAskNewPage, 594
drawDetails, 787
ecdf, 1063
editDetails, 788
expression, 139
extendrange, 596
fft, 1081
gEdit, 789
getNames, 790
gpar, 790
gPath, 792
Grid, 793
Grid Viewports, 794
grid.add, 797
grid.arrows, 798
grid.circle, 801
grid.clip, 802
grid.collection, 803
grid.convert, 804
grid.copy, 806
grid.curve, 807
grid.display.list, 809
grid.draw, 810
grid.edit, 811
grid.frame, 812
grid.get, 813
grid.grab, 814
grid.grill, 815
grid.grob, 816
grid.layout, 818
grid.lines, 819
grid.locator, 821
grid.ls, 822
grid.move.to, 824
grid.newpage, 826
grid.null, 826
grid.pack, 827
grid.place, 829
grid.plot.and.legend, 830
grid.points, 830
grid.polygon, 831
grid.pretty, 833
grid.prompt, 833
grid.record, 834
grid.rect, 835
grid.refresh, 836
grid.remove, 836
grid.segments, 837
grid.set, 839
grid.show.layout, 840

grid.show.viewport, 841
grid.text, 842
grid.xaxis, 844
grid.xspline, 845
grid.yaxis, 848
grobName, 849
grobWidth, 849
grobX, 850
hcl, 601
hist, 697
hist.POSIXt, 700
hsv, 606
jitter, 220
layout, 706
n2mfrow, 609
Palettes, 612
panel.smooth, 724
par, 725
plot.density, 1228
plotViewport, 851
pop.viewport, 851
ppoints, 1249
pretty, 313
push.viewport, 852
Querying the Viewport Tree, 853
rgb2hsv, 642
screen, 758
splinefun, 1321
stepfun, 1339
stringWidth, 854
strwidth, 769
trans3d, 645
unit, 854
unit.c, 856
unit.length, 857
unit.pmin, 858
unit.rep, 858
units, 778
validDetails, 859
vpPath, 860
widthDetails, 861
Working with Viewports, 861
xDetails, 864
xy.coords, 653
xyTable, 655
xyz.coords, 656

∗Topic **environment**
apropos, 1428
as.environment, 21
browser, 41
commandArgs, 65
debug, 99
eapply, 125
gc, 179
gctorture, 181
interactive, 209
is.R, 215
layout, 706
ls, 250
Memory, 267
Memory-limits, 269
options, 297
par, 725
quit, 328
R.Version, 331
reg.finalizer, 354
remove, 359
Startup, 414
stop, 416
stopifnot, 418
Sys.getenv, 439
Sys.setenv, 445
taskCallback, 458
taskCallbackManager, 460
taskCallbackNames, 462

∗Topic **error**
bug.report, 1434
conditions, 70
debugger, 1449
help.request, 1471
options, 297
stop, 416
stopifnot, 418
warning, 490
warnings, 491

∗Topic **file**
.Platform, 4
basename, 35
browseURL, 1432
cat, 48
connections, 74
count.fields, 1444
dataentry, 1447
dcf, 98
dput, 117
dump, 119
file.access, 152
file.choose, 154
file.info, 154
file.path, 155
file.show, 156
file_test, 1459
files, 157

fileutils, 1408
glob2rx, 1465
gzcon, 198
list.files, 239
load, 241
package.skeleton, 1491
parse, 307
path.expand, 310
rawConnection, 343
read.00Index, 1417
read.DIF, 1506
read.fortran, 1508
read.fwf, 1509
read.table, 1511
readBin, 346
readChar, 349
readLines, 351
readNEWS, 1418
save, 373
scan, 377
seek, 381
sink, 393
source, 400
Sys.glob, 440
sys.source, 447
system, 449
system.file, 451
tempfile, 463
textConnection, 464
unlink, 481
url.show, 1550
write, 497
write.table, 1555
write_PACKAGES, 1423
writeLines, 498
zip.file.extract, 1558
∗Topic **graphs**
chull, 578
∗Topic **hplot**
assocplot, 662
barplot, 670
biplot, 1010
biplot.princomp, 1011
boxplot, 674
cdplot, 680
contour, 683
coplot, 686
cpgram, 1042
curve, 689
dendrogram, 1048
dotchart, 690
ecdf, 1063
filled.contour, 691
fourfoldplot, 694
heatmap, 1111
hist, 697
hist.POSIXt, 700
image, 704
interaction.plot, 1124
lag.plot, 1141
matplot, 715
monthplot, 1184
mosaicplot, 717
pairs, 722
panel.smooth, 724
persp, 732
pie, 736
plot, 737
plot.acf, 1227
plot.data.frame, 739
plot.default, 740
plot.design, 742
plot.factor, 744
plot.formula, 744
plot.histogram, 746
plot.isoreg, 1230
plot.lm, 1231
plot.ppr, 1234
plot.spec, 1236
plot.stepfun, 1237
plot.table, 747
plot.ts, 1239
qqnorm, 1279
spineplot, 761
stars, 763
stripchart, 767
sunflowerplot, 770
symbols, 772
termplot, 1363
∗Topic **htest**
ansari.test, 979
bartlett.test, 1004
binom.test, 1007
chisq.test, 1019
cor.test, 1038
fisher.test, 1084
fligner.test, 1088
friedman.test, 1092
kruskal.test, 1135
ks.test, 1137
mantelhaen.test, 1170
mauchly.test, 1172
mcnemar.test, 1174
mood.test, 1186

oneway.test, 1212
p.adjust, 1222
pairwise.prop.test, 1224
pairwise.t.test, 1224
pairwise.table, 1225
pairwise.wilcox.test, 1226
power.anova.test, 1244
power.prop.test, 1245
power.t.test, 1247
print.power.htest, 1270
prop.test, 1276
prop.trend.test, 1278
quade.test, 1280
shapiro.test, 1305
t.test, 1359
var.test, 1383
wilcox.test, 1389
∗Topic **interface**
.Script, 5
browseEnv, 1430
dyn.load, 122
getDLLRegisteredRoutines, 183
getLoadedDLLs, 185
getNativeSymbolInfo, 186
getNumCConverters, 188
Internal, 210
Primitive, 315
system, 449
∗Topic **iplot**
dev.xxx, 588
frame, 695
getGraphicsEvent, 597
identify, 702
identify.hclust, 1118
layout, 706
locator, 714
par, 725
plot.histogram, 746
recordPlot, 640
∗Topic **iteration**
apply, 15
by, 43
combn, 1441
Control, 82
dendrapply, 1046
eapply, 125
identical, 204
lapply, 225
rapply, 340
sweep, 436
tapply, 457
∗Topic **list**
clearNames, 1024
eapply, 125
Extract, 140
lapply, 225
list, 237
NULL, 290
rapply, 340
relist, 1517
setNames, 1304
unlist, 482
∗Topic **loess**
loess, 1152
loess.control, 1154
∗Topic **logic**
all, 9
all.equal, 10
any, 12
Comparison, 67
complete.cases, 1027
Control, 82
duplicated, 121
identical, 204
ifelse, 206
Logic, 246
logical, 248
match, 257
NA, 276
unique, 479
which, 493
∗Topic **manip**
addmargins, 967
append, 14
c, 44
cbind, 50
cbind2, 875
Colon, 63
cut.POSIXt, 88
deparse, 102
dimnames, 113
duplicated, 121
expand.model.frame, 1069
getInitial, 1101
head, 1466
list, 237
mapply, 254
match, 257
merge, 270
model.extract, 1178
NA, 276
NLSstAsymptotic, 1206
NLSstClosestX, 1206
NLSstLfAsymptote, 1207

NLSstRtAsymptote, 1208
NULL, 290
order, 303
order.dendrogram, 1221
relist, 1517
reorder.dendrogram, 1288
rep, 360
replace, 362
reshape, 1292
rev, 363
rle, 364
row/colnames, 370
rowsum, 371
seq, 382
seq.Date, 384
seq.POSIXt, 385
sequence, 386
slotOp, 395
sort, 398
sortedXyData, 1315
stack, 1535
structure, 426
subset, 428
transform, 475
type.convert, 1545
unique, 479
unlist, 482
xtfrm, 499

∗Topic **math**
.Machine, 2
Bessel, 36
convolve, 1033
deriv, 1054
fft, 1081
Hyperbolic, 200
integrate, 1122
is.finite, 211
kappa, 221
log, 244
MathFun, 261
nextn, 1194
poly, 1242
polyroot, 312
Special, 402
splinefun, 1321
Trig, 476

∗Topic **methods**
.BasicFunsList, 866
as, 866
as.data.frame, 20
callGeneric, 871
callNextMethod, 873
canCoerce, 875
class, 60
Classes, 877
data.class, 89
data.frame, 90
Documentation, 881
dotsMethods, 883
findClass, 886
findMethods, 888
GenericFunctions, 891
getMethod, 896
groupGeneric, 196
implicitGeneric, 900
initialize-methods, 902
InternalMethods, 210
is, 903
is.object, 214
isSealedMethod, 908
method.skeleton, 912
Methods, 914
methods, 1486
methods-package, 865
MethodsList-class, 918
na.action, 1188
noquote, 283
plot.data.frame, 739
predict, 1255
promptMethods, 924
row.names, 369
S4groupGeneric, 930
setClass, 933
setGeneric, 937
setMethod, 942
setOldClass, 946
showMethods, 951
summary, 433
UseMethod, 484

∗Topic **misc**
base-deprecated, 34
citation, 1438
citEntry, 1439
close.socket, 1441
contributors, 81
copyright, 83
license, 237
make.socket, 1483
mirrorAdmin, 1487
person, 1495
read.socket, 1510
sessionInfo, 1533
sets, 387
stats-deprecated, 1337

toLatex, 1543
tools-deprecated, 1420
url.show, 1550
utils-deprecated, 1552
∗Topic **models**
add1, 965
AIC, 971
alias, 972
anova, 973
anova.glm, 974
anova.lm, 976
anova.mlm, 977
aov, 982
AsIs, 25
asOneSidedFormula, 1001
C, 1015
case/variable.names, 1017
coef, 1026
confint, 1028
deviance, 1057
df.residual, 1058
dummy.coef, 1062
eff.aovlist, 1066
effects, 1067
expand.grid, 138
extractAIC, 1071
factor.scope, 1075
family, 1076
fitted, 1086
formula, 1090
formula.nls, 1092
getInitial, 1101
glm, 1102
glm.control, 1106
glm.summaries, 1107
is.empty.model, 1127
labels, 224
lm.summaries, 1149
logLik, 1156
loglin, 1157
make.link, 1167
makepredictcall, 1168
manova, 1169
mauchly.test, 1172
model.extract, 1178
model.frame, 1179
model.matrix, 1181
model.tables, 1182
naprint, 1191
naresid, 1191
nls, 1199
nls.control, 1204
numericDeriv, 1211
offset, 1212
plot.profile.nls, 1235
power, 1243
predict.glm, 1257
predict.nls, 1264
preplot, 1267
profile, 1273
profile.nls, 1273
proj, 1274
relevel, 1288
replications, 1291
residuals, 1295
se.contrast, 1300
selfStart, 1302
simulate, 1307
SSasymp, 1324
SSasympOff, 1325
SSasympOrig, 1326
SSbiexp, 1327
SSD, 1328
SSfol, 1329
SSfpl, 1330
SSgompertz, 1331
SSlogis, 1332
SSmicmen, 1333
SSweibull, 1334
stat.anova, 1336
step, 1337
summary.aov, 1346
summary.glm, 1348
summary.lm, 1350
summary.manova, 1352
summary.nls, 1353
terms, 1365
terms.formula, 1366
terms.object, 1367
tilde, 466
TukeyHSD, 1377
update, 1381
update.formula, 1382
vcov, 1386
∗Topic **multivariate**
anova.mlm, 977
as.hclust, 1000
biplot, 1010
biplot.princomp, 1011
cancor, 1016
cmdscale, 1025
cophenetic, 1035
cor, 1036
cov.wt, 1041

cutree, 1043
dendrogram, 1048
dist, 1059
factanal, 1072
hclust, 1108
kmeans, 1133
loadings, 1151
mahalanobis, 1166
mauchly.test, 1172
prcomp, 1253
princomp, 1267
screeplot, 1299
SSD, 1328
stars, 763
summary.princomp, 1355
symbols, 772
varimax, 1385
∗Topic **nonlinear**
deriv, 1054
getInitial, 1101
nlm, 1195
nls, 1199
nls.control, 1204
optim, 1214
plot.profile.nls, 1235
predict.nls, 1264
profile.nls, 1273
vcov, 1386
∗Topic **nonparametric**
sunflowerplot, 770
∗Topic **optimize**
constrOptim, 1029
glm.control, 1106
nlm, 1195
nlminb, 1197
optim, 1214
optimize, 1219
uniroot, 1380
∗Topic **package**
base-package, 1
datasets-package, 501
graphics-package, 659
grDevices-package, 573
grid-package, 783
methods-package, 865
stats-package, 961
tools-package, 1399
utils-package, 1427
∗Topic **print**
cat, 48
dcf, 98
format, 165, 1461
format.info, 170
format.pval, 171
formatC, 172
formatDL, 174
hexmode, 200
labels, 224
loadings, 1151
ls.str, 1481
noquote, 283
octmode, 294
options, 297
plot.isoreg, 1230
print, 315
print.data.frame, 317
print.default, 318
printCoefmat, 1271
prmatrix, 320
sprintf, 407
str, 1536
write.table, 1555
∗Topic **programming**
.BasicFunsList, 866
.Machine, 2
all.equal, 10
all.names, 11
as, 866
as.function, 22
autoload, 32
body, 39
bquote, 40
browser, 41
call, 45
callCC, 46
callGeneric, 871
callNextMethod, 873
check.options, 577
checkFF, 1401
Classes, 877
commandArgs, 65
conditions, 70
Control, 82
debug, 99
delayedAssign, 101
delete.response, 1045
deparse, 102
deparseOpts, 103
do.call, 114
Documentation, 881
dotsMethods, 883
dput, 117
environment, 130
eval, 134

expression, 139
findClass, 886
findMethods, 888
fixPre1.8, 889
force, 160
Foreign, 161
formals, 164
format.info, 170
function, 176
funprog, 177
GenericFunctions, 891
getClass, 895
getMethod, 896
getNumCConverters, 188
getPackageName, 898
hasArg, 899
identical, 204
identity, 206
ifelse, 206
implicitGeneric, 900
initialize-methods, 902
interactive, 209
invisible, 211
is, 903
is.finite, 211
is.function, 213
is.language, 214
is.recursive, 216
isS4, 218
isSealedMethod, 908
Last.value, 227
makeClassRepresentation, 911
match.arg, 258
match.call, 259
match.fun, 260
menu, 1485
message, 272
method.skeleton, 912
Methods, 914
missing, 273
model.extract, 1178
name, 277
nargs, 280
new, 920
ns-dblcolon, 286
ns-topenv, 289
on.exit, 295
Paren, 306
parse, 307
promptClass, 923
promptMethods, 924
R.Version, 331
Recall, 353
recover, 1515
reg.finalizer, 354
representation, 925
Reserved, 362
S3Part, 927
setClass, 933
setClassUnion, 936
setGeneric, 937
setMethod, 942
setOldClass, 946
show, 950
slot, 954
source, 400
stop, 416
stopifnot, 418
substitute, 429
switch, 437
Syntax, 438
sys.parent, 442
tools-package, 1399
trace, 469
traceback, 472
try, 477
utils-package, 1427
validObject, 958
warning, 490
warnings, 491
with, 495
withVisible, 1554

∗Topic **regression**

anova, 973
anova.glm, 974
anova.lm, 976
anova.mlm, 977
aov, 982
case/variable.names, 1017
coef, 1026
contrast, 1031
contrasts, 1032
df.residual, 1058
effects, 1067
expand.model.frame, 1069
fitted, 1086
glm, 1102
glm.summaries, 1107
influence.measures, 1119
isoreg, 1128
line, 1142
lm, 1143
lm.fit, 1146
lm.influence, 1148

lm.summaries, 1149
ls.diag, 1162
ls.print, 1163
lsfit, 1164
nls, 1199
nls.control, 1204
plot.lm, 1231
plot.profile.nls, 1235
ppr, 1250
predict.glm, 1257
predict.lm, 1260
predict.nls, 1264
profile.nls, 1273
residuals, 1295
stat.anova, 1336
summary.aov, 1346
summary.glm, 1348
summary.lm, 1350
summary.nls, 1353
termplot, 1363
weighted.residuals, 1388
∗Topic **robust**
fivenum, 1087
IQR, 1126
line, 1142
mad, 1165
median, 1175
medpolish, 1176
runmed, 1295
smooth, 1308
smoothEnds, 1313
∗Topic **smooth**
bandwidth, 1002
density, 1051
isoreg, 1128
ksmooth, 1139
loess, 1152
loess.control, 1154
lowess, 1161
predict.loess, 1262
predict.smooth.spline, 1265
runmed, 1295
scatter.smooth, 1297
smooth, 1308
smooth.spline, 1310
smoothEnds, 1313
sunflowerplot, 770
supsmu, 1356
∗Topic **sysdata**
.Machine, 2
colors, 583
commandArgs, 65
Constants, 80
NULL, 290
palette, 611
R.Version, 331
Random, 332
Random.user, 336
∗Topic **tree**
dendrogram, 1048
∗Topic **ts**
acf, 962
acf2AR, 964
ar, 985
ar.ols, 988
arima, 990
arima.sim, 993
arima0, 995
ARMAacf, 998
ARMAtoMA, 999
Box.test, 1014
cpgram, 1042
decompose, 1044
diffinv, 1058
embed, 1068
filter, 1082
HoltWinters, 1114
KalmanLike, 1129
kernapply, 1131
kernel, 1132
lag, 1140
lag.plot, 1141
monthplot, 1184
na.contiguous, 1189
plot.acf, 1227
plot.HoltWinters, 1229
plot.spec, 1236
plot.ts, 1239
PP.test, 1248
predict.Arima, 1256
predict.HoltWinters, 1259
print.ts, 1270
spec.ar, 1315
spec.pgram, 1317
spec.taper, 1319
spectrum, 1320
start, 1335
stl, 1341
stlmethods, 1343
StructTS, 1344
time, 1368
toeplitz, 1369
ts, 1370
ts-methods, 1371

ts.plot, 1372
ts.union, 1373
tsdiag, 1374
tsp, 1375
tsSmooth, 1375
window, 1394
∗Topic **univar**
ave, 1001
cor, 1036
Extremes, 148
fivenum, 1087
IQR, 1126
is.unsorted, 217
mad, 1165
mean, 266
median, 1175
nclass, 610
order, 303
quantile, 1282
range, 338
rank, 339
sd, 1300
sort, 398
stem, 766
weighted.mean, 1388
xtfrm, 499
∗Topic **utilities**
.Platform, 4
.checkMFClasses, 961
alarm, 1427
all.equal, 10
as.graphicsAnnot, 573
as.POSIX∗, 23
axis.POSIXct, 667
BATCH, 1429
bindenv, 38
bug.report, 1434
buildVignettes, 1399
builtins, 42
capabilities, 47
capture.output, 1436
check.options, 577
checkFF, 1401
checkMD5sums, 1402
checkTnF, 1403
checkVignettes, 1404
chooseCRANmirror, 1437
combn, 1441
compareVersion, 1443
COMPILE, 1443
conflicts, 73
Cstack_info, 85
dataentry, 1447
date, 94
Dates, 94
DateTimeClasses, 95
debugger, 1449
Defunct, 100
demo, 1451
Deprecated, 105
dev2bitmap, 592
difftime, 110
download.file, 1452
edit, 1454
edit.data.frame, 1455
encoded_text_to_latex, 1407
encodeString, 127
Encoding, 129
EnvVar, 132
example, 1457
file.edit, 1459
findInterval, 159
fix, 1460
flush.console, 1461
format.Date, 168
gc.time, 180
getDepList, 1410
getpid, 189
gettext, 190
getwd, 191
glob2rx, 1465
grep, 193
help.request, 1471
iconv, 201
icuSetCollate, 203
index.search, 1476
INSTALL, 1477
installed.packages, 1478
installFoundDepends, 1411
integrate, 1122
is.R, 215
isSymmetric, 219
jitter, 220
l10n_info, 224
LINK, 1479
localeconv, 242
locales, 243
localeToCharset, 1480
ls.str, 1481
make.packages.html, 1482
makeLazyLoading, 1412
manglePackageName, 253
mapply, 254
maxCol, 265

md5sum, 1413
memory.profile, 269
memory.size, 1484
menu, 1485
modifyList, 1488
n2mfrow, 609
noquote, 283
normalizePath, 1488
NotYet, 284
ns-hooks, 287
ns-load, 288
nsl, 1489
numeric_version, 293
object.size, 1490
Ops.Date, 296
package.dependencies, 1413
package.skeleton, 1491
packageDescription, 1492
packageStatus, 1493
page, 1495
PkgUtils, 1496
pos.to.env, 313
proc.time, 321
QC, 1414
rcompgen, 1501
Rdindex, 1415
RdUtils, 345
Rdutils, 1416
readline, 350
relevel, 1288
REMOVE, 1519
remove.packages, 1520
reorder.factor, 1289
RHOME, 1520
Rhome, 364
Rprof, 1521
Rprofmem, 1523
Rscript, 1524
RSiteSearch, 1526
Rtangle, 1527
RweaveLatex, 1528
savehistory, 1530
savePlot, 644
select.list, 1532
setRepositories, 1533
setTimeLimit, 388
SHLIB, 1534
showNonASCII, 1419
shQuote, 390
Signals, 392
srcfile, 412
str, 1536
strptime, 419
strtrim, 425
summaryRprof, 1539
Sweave, 1540
SweaveSyntConv, 1542
symnum, 1357
Sys.getenv, 439
Sys.glob, 440
Sys.info, 441
Sys.setenv, 445
Sys.sleep, 446
sys.source, 447
Sys.time, 448
Sys.which, 449
system, 449
system.file, 451
system.time, 452
texi2dvi, 1420
timezones, 466
toString, 468
tracemem, 473
txtProgressBar, 1544
unname, 483
update.packages, 1546
URLencode, 1551
UserHooks, 486
utf8Conversion, 487
View, 1552
vignetteDepends, 1422
which.min, 494
write_PACKAGES, 1423
xgettext, 1424
zutils, 500
' *(Quotes)*, 330
* *(Arithmetic)*, 17
** *(Arithmetic)*, 17
*.difftime *(difftime)*, 110
+ *(Arithmetic)*, 17
+.Date *(Ops.Date)*, 296
+.POSIXt *(DateTimeClasses)*, 95
- *(Arithmetic)*, 17
-.Date *(Ops.Date)*, 296
-.POSIXt *(DateTimeClasses)*, 95
-> *(assignOps)*, 27
->> *(assignOps)*, 27
.AutoloadEnv *(autoload)*, 32
.Autoloaded *(autoload)*, 32
.BaseNamespaceEnv *(environment)*, 130
.BasicFunsList, 866
.C, *116*, *123*, *125*, *183*, *186–189*, *1402*
.C *(Foreign)*, 161

.Call, *123*, *125*, *183*, *186–188*, *473*
.Call *(Foreign)*, 161
.Class *(UseMethod)*, 484
.Defunct *(Defunct)*, 100
.Deprecated *(Deprecated)*, 105
.Device, 1, *587*, *637*
.Devices *(.Device)*, 1
.External, *123*, *125*, *183*, *186–188*, *210*
.External *(Foreign)*, 161
.First, *209*, *329*
.First *(Startup)*, 414
.First.lib, *125*, *236, 237*, *287*
.First.lib *(library)*, 231
.Fortran, *116*, *123–125*, *183*, *186–188*, *1402*
.Fortran *(Foreign)*, 161
.Generic *(UseMethod)*, 484
.GlobalEnv, *381*, *430*, *443*
.GlobalEnv *(environment)*, 130
.Group *(groupGeneric)*, 196
.InitTraceFunctions *(TraceClasses)*, 957
.Internal, *42*, *315*, *485*
.Internal *(Internal)*, 210
.Last, *392*, *414, 415*, *1531*
.Last *(quit)*, 328
.Last.lib, *236*, *287*
.Last.lib *(library)*, 231
.Last.value *(Last.value)*, 227
.Library *(libPaths)*, 230
.MFclass, *1367*
.MFclass *(.checkMFClasses)*, 961
.Machine, 2, *4*, *347*, *1219*
.Method *(UseMethod)*, 484
.NotYetImplemented *(NotYet)*, 284
.NotYetUsed *(NotYet)*, 284
.OldClassesList *(setOldClass)*, 946
.OptRequireMethods *(Startup)*, 414
.Options *(options)*, 297
.Pars *(par)*, 725
.Platform, *3*, 4, *48*, *332*, *442*, *451*, *1409*
.Primitive, *210*, *306*
.Primitive *(Primitive)*, 315
.Random.seed, *1210*, *1307*, *1380*
.Random.seed *(Random)*, 332
.Renviron *(Startup)*, 414
.Rprofile, *298*
.Rprofile *(Startup)*, 414
.S3PrimitiveGenerics *(InternalMethods)*, 210
.Script, 5
.Traceback *(traceback)*, 472
.__H__.cbind *(cbind)*, 50
.__H__.rbind *(cbind)*, 50
.checkMFClasses, 961
.decode_numeric_version *(numeric_version)*, 293
.deparseOpts, *102*, *117*, *119*, *1454*
.deparseOpts *(deparseOpts)*, 103
.doTrace *(trace)*, 469
.doTracePrint *(TraceClasses)*, 957
.dynLibs *(library.dynam)*, 235
.encode_numeric_version *(numeric_version)*, 293
.expand_R_libs_env_var *(libPaths)*, 230
.getXlevels *(.checkMFClasses)*, 961
.handleSimpleError *(conditions)*, 70
.isOpen *(srcfile)*, 412
.leap.seconds *(DateTimeClasses)*, 95
.libPaths, *133*, *234*, *237*, *500*, *1446*, *1520*, *1547*
.libPaths *(libPaths)*, 230
.makeMessage *(message)*, 272
.makeTracedFunction *(TraceClasses)*, 957
.make_numeric_version *(numeric_version)*, 293
.noGenerics *(library)*, 231
.onAttach, *289*
.onAttach *(ns-hooks)*, 287
.onLoad, *233*, *236*, *288*
.onLoad *(ns-hooks)*, 287
.onUnload, *236*, *289*
.onUnload *(ns-hooks)*, 287
.packageStartupMessage *(message)*, 272
.packages, *234*, *237*, *381*, *1550*
.packages *(zpackages)*, 499
.primTrace *(trace)*, 469
.primUntrace *(trace)*, 469
.ps.prolog *(postscript)*, 627
.row_names_info, *370*
.setOldIs *(setOldClass)*, 946
.signalSimpleWarning *(conditions)*, 70
.slotNames *(slot)*, 954
.standard_regexps *(zutils)*, 500
.untracedFunction *(TraceClasses)*, 957
.userHooksEnv *(UserHooks)*, 486
/ *(Arithmetic)*, 17

/.difftime (difftime), 110
:, 209, 384
: (Colon), 63
:: (ns-dblcolon), 286
:::, 1464
::: (ns-dblcolon), 286
< (Comparison), 67
<-, 27
<- (assignOps), 27
<-class (language-class), 909
<= (Comparison), 67
<<-, 29
<<- (assignOps), 27
= (assignOps), 27
==, 11
== (Comparison), 67
> (Comparison), 67
>= (Comparison), 67
?, 1474
? (help), 1468
??, 1471
?? (help.search), 1473
[, 119, 144, 210, 294, 429, 1132
[(Extract), 140
[.AsIs (AsIs), 25
[.Date (Dates), 94
[.POSIXct (DateTimeClasses), 95
[.POSIXlt (DateTimeClasses), 95
[.acf (acf), 962
[.data.frame, 92, 141–143, 1179
[.data.frame
(Extract.data.frame), 144
[.difftime (difftime), 110
[.factor, 141, 143, 151, 152
[.factor (Extract.factor), 146
[.formula (formula), 1090
[.getAnywhere (getAnywhere), 1462
[.hexmode (hexmode), 200
[.noquote (noquote), 283
[.numeric_version
(numeric_version), 293
[.octmode (octmode), 294
[.simple.list (Extract), 140
[.terms (delete.response), 1045
[.ts (ts), 1370
[<-, 210
[<- (Extract), 140
[<-.Date (Dates), 94
[<-.POSIXct (DateTimeClasses), 95
[<-.POSIXlt (DateTimeClasses), 95
[<-.data.frame
(Extract.data.frame), 144
[<-.factor (Extract.factor), 146
[[, 210, 1050
[[(Extract), 140
[[.Date (Dates), 94
[[.POSIXct (DateTimeClasses), 95
[[.data.frame
(Extract.data.frame), 144
[[.dendrogram (dendrogram), 1048
[[.factor (Extract.factor), 146
[[.numeric_version
(numeric_version), 293
[[<-, 210
[[<- (Extract), 140
[[<-.data.frame
(Extract.data.frame), 144
[[<-.numeric_version
(numeric_version), 293
$, 210
$ (Extract), 140
$.DLLInfo (getLoadedDLLs), 185
$.package_version
(numeric_version), 293
$<-, 210
$<- (Extract), 140
$<-.data.frame
(Extract.data.frame), 144
%*%, 19, 84, 223, 306
%*% (matmult), 262
%/% (Arithmetic), 17
%% (Arithmetic), 17
%in%, 388
%in% (match), 257
%o%, 84
%o% (outer), 305
%x% (kronecker), 222
& (Logic), 246
&& (Logic), 246
__ClassMetaData (Classes), 877
^ (Arithmetic), 17
~ (tilde), 466
` (Quotes), 330
| (Logic), 246

abbreviate, 6, 1358
ability.cov, 501, 1075
abline, 659, 697, 713, 726, 754
abs, 392
abs (MathFun), 261
absolute.size, 784, 861
acf, 962, 1228
acf2AR, 964, 988, 999
acos, 201
acos (Trig), 476

acosh (Hyperbolic), 200
adapt, 1124
add.scope (factor.scope), 1075
add1, 965, 974, 1072, 1076, 1338, 1339
addGrob, 793, 798, 812, 814
addGrob (grid.add), 797
addmargins, 256, 455, 967
addNA (factor), 150
addTaskCallback, 459, 461, 462
addTaskCallback (taskCallback), 458
Adobe_glyphs (charsets), 1400
aggregate, 16, 372, 458, 969
agnes, 1000, 1035, 1111
agrep, 7, 195, 1474
AIC, 971, 1071, 1072
airmiles, 502
AirPassengers, 503, 1345, 1376
airquality, 504
alarm, 1427
alias, 972, 983, 1027
alist, 23, 40, 165
alist (list), 237
all, 9, 11, 13, 248, 418
all.equal, 10, 68, 205, 219
all.equal.numeric, 219
all.equal.POSIXct (DateTimeClasses), 95
all.names, 11, 1018
all.vars, 1018, 1091
all.vars (all.names), 11
allGenerics (GenericFunctions), 891
anova, 434, 967, 973, 975, 977, 1104, 1105, 1136, 1144, 1163, 1201, 1245, 1336
anova-class (setOldClass), 946
anova.glm, 966, 974, 1104, 1105, 1108, 1336
anova.glm-class (setOldClass), 946
anova.glm.null-class (setOldClass), 946
anova.glmlist (anova.glm), 974
anova.lm, 976, 1146, 1150, 1336
anova.lmlist (anova.lm), 976
anova.mlm, 977, 1174, 1328
anova.mlmlist (anova.mlm), 977
ansari.test, 979, 1005, 1089, 1187, 1384
ansari_test, 981
anscombe, 505, 1146
any, 10, 12, 248
ANY-class (BasicClasses), 870
aov, 301, 742, 966, 967, 982, 1032, 1033, 1063, 1066, 1067, 1071, 1076, 1143, 1144, 1146, 1150, 1169, 1177, 1183, 1233, 1274, 1275, 1301, 1337, 1347, 1353, 1367, 1378
aov-class (setOldClass), 946
aperm, 13, 20, 453, 1294
append, 14
apply, 15, 64, 121, 226, 261, 437, 458, 970
applyEdit (gEdit), 789
applyEdits (gEdit), 789
approx, 160, 1323
approx (approxfun), 983
approxfun, 983, 1065, 1238, 1323, 1340
apropos, 195, 251, 358, 1428, 1475
ar, 985, 990, 993, 998, 1316
ar.burg (ar), 985
ar.mle (ar), 985
ar.ols, 986, 988, 988
ar.yw, 964
ar.yw (ar), 985
arcCurvature (grid.curve), 807
Arg (complex), 69
args, 16, 40, 165, 176, 280, 1463, 1482, 1502, 1536, 1538
argsAnywhere (getAnywhere), 1462
arima, 988, 990, 994, 997–1000, 1131, 1256, 1257, 1345, 1374
arima.sim, 988, 993, 993, 1083
arima0, 992, 993, 995
Arith, 18
Arith (S4groupGeneric), 930
Arithmetic, 17, 212, 246, 262, 263, 404, 439, 1244
ARMAacf, 964, 998, 1000
ARMAtoMA, 999, 999
array, 19, 113, 114, 119, 143, 285, 457, 493, 877, 1441, 1442
array-class (StructureClasses), 956
arrow, 785, 808, 821, 825, 838, 847
arrows, 661, 726, 760
arrowsGrob (grid.arrows), 798
as, 61, 93, 489, 866, 875, 880, 902, 905, 906, 928, 932, 933
as.array (array), 19
as.call (call), 45
as.character, 6, 49, 55, 56, 103, 113, 128, 167, 193, 202, 210, 238, 281, 309, 311, 370, 408, 425, 427, 431, 574, 683, 702, 775, 1358
as.character (character), 53

as.character.condition (conditions), 70
as.character.Date (format.Date), 168
as.character.error (conditions), 70
as.character.hexmode (hexmode), 200
as.character.numeric_version (numeric_version), 293
as.character.octmode (octmode), 294
as.character.person (person), 1495
as.character.personList (person), 1495
as.character.POSIXt (strptime), 419
as.character.srcref (srcfile), 412
as.complex, *210*
as.complex (complex), 69
as.data.frame, 20, *25*, *91*, *145*, *1095*, *1102*, *1143*, *1152*, *1179*
as.data.frame.Date (Dates), 94
as.data.frame.numeric_version (numeric_version), 293
as.data.frame.POSIXct (DateTimeClasses), 95
as.data.frame.POSIXlt (DateTimeClasses), 95
as.data.frame.table, *21*, *1396*
as.data.frame.table (table), 454
as.Date, *296*
as.Date (format.Date), 168
as.dendrogram, *1047*, *1112*, *1221*
as.dendrogram (dendrogram), 1048
as.difftime (difftime), 110
as.dist (dist), 1059
as.double, *210*, *291*, *353*, *408*
as.double (double), 115
as.double.difftime (difftime), 110
as.double.POSIXlt (as.POSIX*), 23
as.environment, 21, *29*, *885*
as.expression (expression), 139
as.factor, *405*, *457*
as.factor (factor), 150
as.formula, *1383*
as.formula (formula), 1090
as.function, 22
as.graphicsAnnot, 573, *665*, *708*, *709*, *720*, *736*, *769*, *778*, *842*, *1232*
as.hclust, 1000, *1035*
as.hexmode (hexmode), 200
as.integer, *82*, *140*, *144*, *210*, *366*
as.integer (integer), 207
as.list, *225*, *483*
as.list (list), 237
as.list.numeric_version (numeric_version), 293
as.logical, *210*
as.logical (logical), 248
as.matrix, *91*, *93*, *263*, *369*, *453*, *1060*, *1061*, *1513*, *1556*
as.matrix (matrix), 263
as.matrix.dist (dist), 1059
as.matrix.noquote (noquote), 283
as.matrix.POSIXlt (DateTimeClasses), 95
as.name, *216*
as.name (name), 277
as.null (NULL), 290
as.numeric, *93*, *313*
as.numeric (numeric), 290
as.numeric_version (numeric_version), 293
as.octmode, *157*
as.octmode (octmode), 294
as.ordered (factor), 150
as.package_version (numeric_version), 293
as.pairlist (list), 237
as.person (person), 1495
as.personList (person), 1495
as.POSIX*, 23
as.POSIXct, *96*, *97*
as.POSIXct (as.POSIX*), 23
as.POSIXlt, *97*, *243*, *419*, *420*, *467*, *493*
as.POSIXlt (as.POSIX*), 23
as.qr (qr), 325
as.raw, *210*
as.raw (raw), 341
as.real (real), 352
as.relistable (relist), 1517
as.roman (roman), 1521
as.single, *163*
as.single (double), 115
as.stepfun, *1128*
as.stepfun (stepfun), 1339
as.symbol, *139*
as.symbol (name), 277
as.table, *1095*
as.table (table), 454
as.ts (ts), 1370
as.vector, *15*, *44*, *54*, *69*, *82*, *116*, *208*, *210*, *238*, *249*, *277*, *387*

as.vector (vector), 488
as<- (as), 866
asin, 201
asin (Trig), 476
asinh (Hyperbolic), 200
AsIs, 25, 166
asOneSidedFormula, 1001
asp (plot.window), 748
asS4, 929
asS4 (isS4), 218
assign, 26, 28, 29, 183, 496, 577
assignInNamespace (getFromNamespace), 1463
assignOps, 27
assoc, 663
assocplot, 662, 719
atan, 201
atan (Trig), 476
atan2 (Trig), 476
atanh (Hyperbolic), 200
atomic, 1515
atomic (vector), 488
atop (plotmath), 619
attach, 26, 28, 106, 107, 234, 380, 381, 496, 1436, 1493
attachNamespace (ns-load), 288
attenu, 506
attitude, 507, 1146
attr, 30, 32, 66, 298, 299, 369, 426, 1296
attr.all.equal (all.equal), 10
attr<-, 955
attr<- (attr), 30
attributes, 10, 11, 31, 31, 66, 113, 141, 148, 275, 279, 426, 577, 935, 1046
attributes<- (attributes), 31
austres, 508
autoload, 32, 234
autoloader (autoload), 32
Autoloads (autoload), 32
available.packages, 1413, 1423, 1452, 1453, 1494
available.packages (update.packages), 1546
ave, 1001
Axis, 664, 667, 684
axis, 619, 622, 664, 665, 668, 669, 672, 678, 698, 726, 729, 730, 757
axis.Date (axis.POSIXct), 667
axis.POSIXct, 664, 667, 701
axTicks, 314, 666, 667, 669, 696, 730

backquote, 176
backquote (Quotes), 330
backsolve, 33, 59, 397
backtick, 28, 140, 363, 1469
backtick (Quotes), 330
bandwidth, 1002
bandwidth.kernel (kernel), 1132
bandwidth.nrd, 1003
bar (plotmath), 619
barplot, 670, 699, 710, 744, 756
barplot.default, 600
bartlett.test, 981, 1004, 1089, 1187, 1384
base, 573
base (base-package), 1
base-defunct, 35
base-deprecated, 105
base-deprecated, 34
base-package, 1
baseenv (environment), 130
basename, 35, 159, 310
BasicClasses, 870
BATCH, 66, 1429
bcv, 1003
beaver1 (beavers), 508
beaver2 (beavers), 508
beavers, 508
Bessel, 36, 404
bessel (Bessel), 36
besselI (Bessel), 36
besselJ (Bessel), 36
besselK (Bessel), 36
besselY (Bessel), 36
Beta, 1005
beta, 36, 1006, 1007
beta (Special), 402
bgroup (plotmath), 619
bindenv, 38
bindingIsActive (bindenv), 38
bindingIsLocked (bindenv), 38
bindtextdomain (gettext), 190
binom.test, 1007, 1278
Binomial, 1008
binomial, 1105
binomial (family), 1076
biplot, 1010, 1012, 1268
biplot.default, 1012
biplot.prcomp, 1255
biplot.prcomp (biplot.princomp), 1011
biplot.princomp, 1011, 1011, 1269
birthday, 1013
bitmap, 594, 596, 626
bitmap (dev2bitmap), 592

BJsales, 509
bmp, *48*, *595*, *649*
bmp *(png)*, 623
BOD, 510
body, 39, *165*, *176*
body<- *(body)*, 39
body<-,MethodDefinition-method *(MethodsList-class)*, 918
bold *(plotmath)*, 619
bolditalic *(plotmath)*, 619
box, 673, *677*, *682*, *684*, *692*, *726*, *747*, *754*, *756*, *765*
Box.test, 1014, *1374*
boxplot, *575*, *664*, 674, *677*, *744*, *745*, *767*
boxplot.stats, 574, *675*, *676*, *1088*, *1283*
bquote, 40, *430*, *622*
break, *362*
break *(Control)*, 82
browseEnv, *358*, 1430
browser, 41, *100*, *298*, *469–471*, *1515*, *1516*
browseURL, *301*, 1432, *1433*, *1469*, *1476*, *1527*
browseVignettes, 1433
bs, *1168*
bug.report, *298*, 1434, *1473*
build *(PkgUtils)*, 1496
buildVignettes, 1399
builtins, 42
bw.bcv *(bandwidth)*, 1002
bw.nrd, *1052*, *1053*
bw.nrd *(bandwidth)*, 1002
bw.nrd0 *(bandwidth)*, 1002
bw.SJ *(bandwidth)*, 1002
bw.ucv *(bandwidth)*, 1002
bxp, *575*, *674–676*, 677
by, 43, *271*, *458*
bzfile *(connections)*, 74

C, *152*, 1015, *1032*, *1033*, *1181*
c, 44, *52*, *97*, *210*, *239*, *283*, *483*, *489*
c.Date *(Dates)*, 94
c.noquote *(noquote)*, 283
c.numeric_version *(numeric_version)*, 293
c.POSIXct *(DateTimeClasses)*, 95
c.POSIXlt *(DateTimeClasses)*, 95
cairo, 576
cairo_pdf, *48*, *590*, *595*, *616*
cairo_pdf *(cairo)*, 576
cairo_ps, *48*, *631*
cairo_ps *(cairo)*, 576
call, *22*, 45, *115*, *134*, *139*, *142*, *214*, *259*, *260*, *275*, *278*, *353*, *1055*, *1061*, *1128*
call-class *(language-class)*, 909
callCC, 46
callGeneric, 871, *874*, *932*
callNextMethod, 873, *914*, *919–921*
canCoerce, *869*, 875
cancor, 1016
capabilities, 47, *78*, *202*, *577*, *595*, *626*, *1452*
capture.output, *394*, *465*, 1436
cars, 511, *1168*, *1243*
case.names, *371*
case.names *(case/variable.names)*, 1017
case/variable.names, 1017
casefold *(chartr)*, 56
cat, 48, *78*, *300*, *309*, *316*, *491*, *497*, *498*, *1106*, *1529*
Cauchy, 1018
cbind, 50, *210*, *271*, *876*, *1373*
cbind.ts *(ts)*, 1370
cbind2, 875
cbind2,ANY,ANY-method *(cbind2)*, 875
cbind2,ANY,missing-method *(cbind2)*, 875
cbind2-methods *(cbind2)*, 875
ccf *(acf)*, 962
cdplot, 680, *763*
ceiling *(Round)*, 365
char.expand, 52
character, 53, *151*, *252*, *283*, *318*, *332*, *450*, *775*, *777*, *1232*, *1537*
character-class *(BasicClasses)*, 870
charmatch, *53*, 55, *195*, *258*, *311*
charset_to_Unicode *(charsets)*, 1400
charsets, 1400
charToRaw, *342*
charToRaw *(rawConversion)*, 344
chartr, *54*, 56, *129*, *195*
check, *1457*
check *(PkgUtils)*, 1496
check.options, 577, *631*, *635*
check_tzones *(DateTimeClasses)*, 95
checkCRAN *(mirrorAdmin)*, 1487
checkDocFiles *(QC)*, 1414
checkDocStyle *(QC)*, 1414
checkFF, 1401
checkMD5sums, 1402, *1413*
checkNEWS *(readNEWS)*, 1418
checkReplaceFuns *(QC)*, 1414

checkS3methods (QC), 1414
checkTnF, 1403
checkVignettes, 1404
ChickWeight, 512
chickwts, 513
childNames (grid.grob), 816
chisq.test, 455, 525, 663, 1019, 1086, 1396
Chisquare, 1022, 1080, 1361
chol, 34, 57, 60, 127
chol2inv, 59, 59, 397
choose, 1442
choose (Special), 402
chooseCRANmirror, 301, 1437, 1534
chron, 24, 168
chull, 578
CIDFont, 629, 632, 634
CIDFont (Type1Font), 645
circleGrob (grid.circle), 801
CITATION, 1438
CITATION (citEntry), 1439
citation, 1438
citEntry, 1438, 1439, 1544
citFooter (citEntry), 1439
citHeader (citEntry), 1439
class, 31, 32, 60, 90, 116, 197, 214, 251, 283, 291, 315, 340, 433, 455, 486, 697, 745, 928, 973, 1144, 1255, 1487
class<- (class), 60
Classes, 865, 877, 881, 896, 914, 915, 921, 935, 955
classRepresentation, 877, 895, 911, 933, 937, 960
classRepresentation-class, 880
ClassUnionRepresentation-class (setClassUnion), 936
clearNames, 1024, 1304
clip, 682, 730, 731
clipboard (connections), 74
clipGrob (grid.clip), 802
close, 1512
close (connections), 74
close.screen (screen), 758
close.socket, 1441, 1484, 1511
close.srcfile (srcfile), 412
close.txtProgressBar (txtProgressBar), 1544
closeAllConnections (showConnections), 389
cm, 579
cm.colors (Palettes), 612
cmdscale, 1025
co.intervals (coplot), 686
CO2, 514
co2, 515
codoc, 1405, 1421
codocClasses (codoc), 1405
codocData (codoc), 1405
coef, 660, 991, 1026, 1028, 1068, 1108, 1146, 1150, 1201, 1351, 1354
coefficients, 974, 1087, 1104, 1295
coefficients (coef), 1026
coerce (as), 866
coerce,ANY,array-method (as), 866
coerce,ANY,call-method (as), 866
coerce,ANY,character-method (as), 866
coerce,ANY,complex-method (as), 866
coerce,ANY,environment-method (as), 866
coerce,ANY,expression-method (as), 866
coerce,ANY,function-method (as), 866
coerce,ANY,integer-method (as), 866
coerce,ANY,list-method (as), 866
coerce,ANY,logical-method (as), 866
coerce,ANY,matrix-method (as), 866
coerce,ANY,name-method (as), 866
coerce,ANY,NULL-method (as), 866
coerce,ANY,numeric-method (as), 866
coerce,ANY,S3-method (S3Part), 927
coerce,ANY,S4-method (S3Part), 927
coerce,ANY,single-method (as), 866
coerce,ANY,ts-method (as), 866
coerce,ANY,vector-method (as), 866
coerce,oldClass,S3-method (S3Part), 927
coerce-methods (as), 866
coerce<- (as), 866
col, 62, 368, 384, 395
col2rgb, 580, 583, 586, 612, 613, 641, 642
colMeans, 267
colMeans (colSums), 64
colnames, 114, 516
colnames (row/colnames), 370
colnames<- (row/colnames), 370
Colon, 63
colon (Colon), 63

colorConverter, *585*
colorConverter *(make.rgb)*, 608
colorRamp, 581, *612*
colorRampPalette *(colorRamp)*, 581
colors, *580*, 583, *586*, *612*, *613*, *731*, *749*, *791*, *1064*
colorspaces *(convertColor)*, 585
colours *(colors)*, 583
colSums, 64, *372*, *433*
combn, *138*, *404*, 1441
commandArgs, 65, *415*
comment, *31*, *32*, 66
comment<- *(comment)*, 66
Compare, *68*
Compare *(S4groupGeneric)*, 930
compareVersion, *293*, 1443
Comparison, 67, *151*, *204*, *205*, *303*, *342*, *399*, *439*
COMPILE, 1443, *1535*
complete.cases, *277*, 1027
Complex, *69*, *70*
Complex *(S4groupGeneric)*, 930
Complex *(groupGeneric)*, 196
complex, 69, *262*, *292*, *312*
complex-class *(BasicClasses)*, 870
computeRestarts *(conditions)*, 70
condition *(conditions)*, 70
conditionCall *(conditions)*, 70
conditionMessage *(conditions)*, 70
conditions, 70, *273*
confint, 1028, *1146*, *1201*
confint.glm, *1028*
confint.nls, *1028*
conflicts, *29*, 73, *232*
Conj, *453*
Conj *(complex)*, 69
connection, *377*, *1417*, *1418*, *1444*, *1509*, *1512*
connection *(connections)*, 74
connections, 74, *298*, *299*, *324*, *343*, *347*, *350*, *352*, *382*, *390*, *465*, *498*
Constants, 80
constrOptim, 1029, *1196*, *1198*, *1217*
contour, *298*, *584*, *603*, *605*, *664*, 683, *693*, *705*, *726*, *734*
contourLines, *298*, 584, *684*
contourplot, *684*, *693*
contr.helmert, *1033*
contr.helmert *(contrast)*, 1031
contr.poly, *1033*, *1243*
contr.poly *(contrast)*, 1031
contr.SAS *(contrast)*, 1031
contr.sum, *1016*, *1033*
contr.sum *(contrast)*, 1031
contr.treatment, *1033*, *1288*
contr.treatment *(contrast)*, 1031
contrast, 1031
contrasts, *147*, *301*, *1016*, 1032, *1032*, *1181*, *1301*
contrasts<- *(contrasts)*, 1032
contrib.url, *1437*
contrib.url *(update.packages)*, 1546
contributors, 81, *83*
Control, 82, *439*
convertColor, 585, *608*
convertHeight *(grid.convert)*, 804
convertNative, 785
convertUnit *(grid.convert)*, 804
convertWidth *(grid.convert)*, 804
convertX *(grid.convert)*, 804
convertXY, 685
convertY *(grid.convert)*, 804
convolve, 1033, *1082*, *1083*, *1131*, *1194*
cooks.distance, *1149*, *1233*
cooks.distance *(influence.measures)*, 1119
cophenetic, 1035, *1289*
coplot, *664*, 686, *724*, *759*, *1091*
copyright, 83
copyrights *(copyright)*, 83
cor, 1036, *1255*, *1268*, *1269*
cor.test, 1038, *1038*
cos, *201*
cos *(Trig)*, 476
cosh *(Hyperbolic)*, 200
count.fields, 1444, *1515*
cov, *1042*, *1166*, *1255*, *1269*
cov *(cor)*, 1036
cov.mcd, *1268*
cov.mve, *1268*
cov.wt, *1038*, 1041, *1073*, *1268*
cov2cor *(cor)*, 1036
covratio, *1149*
covratio *(influence.measures)*, 1119
coxph, *1364*, *1367*
cpgram, 1042, *1319*
CRAN.packages *(utils-deprecated)*, 1552
crimtab, 515
crossprod, 84
Cstack_info, 85
cummax *(cumsum)*, 85

cummin (cumsum), 85
cumprod, 323
cumprod (cumsum), 85
cumsum, 85, 323
current.transform (Querying the Viewport Tree), 853
current.viewport (Querying the Viewport Tree), 853
current.vpPath (Querying the Viewport Tree), 853
current.vpTree (Querying the Viewport Tree), 853
curve, 689
curveGrob (grid.curve), 807
cut, 86, 88, 89, 406, 705
cut.Date, 95
cut.Date (cut.POSIXt), 88
cut.dendrogram (dendrogram), 1048
cut.POSIXt, 88, 98
cut2, 87
cutree, 1043, 1111
cycle (time), 1368

D (deriv), 1054
daisy, 1061
data, 81, 234, 375, 1445, 1471, 1502, 1504, 1558
data.class, 89
data.entry, 1455, 1456, 1553
data.entry (dataentry), 1447
data.frame, 21, 25, 50–52, 66, 90, 93, 106, 107, 112, 114, 145, 146, 196, 252, 264, 271, 299, 318, 370, 380, 453, 475, 483, 654, 656, 738, 739, 1179, 1180, 1507–1510, 1514, 1515
data.frame-class (setOldClass), 946
data.frameRowLabels-class (setOldClass), 946
data.matrix, 92, 265, 369, 722, 739
dataentry, 301, 1447
datasets (datasets-package), 501
datasets-package, 501
dataViewport, 786, 851
Date, 94, 110, 169, 196, 291, 296, 367, 384, 385, 421, 448, 457, 493, 1176
Date (Dates), 94
date, 24, 94, 111, 168, 243, 448, 1369
Date-class (setOldClass), 946
date-time, 111
date-time (DateTimeClasses), 95
Dates, 94, 97, 668
DateTimeClasses, 24, 81, 94, 95, 95, 111, 155, 367, 386, 422, 448, 493, 668
dbeta, 1081, 1099
dbeta (Beta), 1005
dbinom, 1006, 1081, 1099, 1100, 1118, 1193, 1241, 1362
dbinom (Binomial), 1008
dcauchy (Cauchy), 1018
dcf, 98
dchisq, 1081, 1099
dchisq (Chisquare), 1022
de (dataentry), 1447
debug, 41, 42, 99, 176
debugger, 1449
decompose, 1044
default.stringsAsFactors (data.frame), 90
Defunct, 100, 105, 284, 1420, 1552
delayedAssign, 33, 101, 120, 430, 1537
delete.response, 1045
delimMatch, 1406
deltat, 1140
deltat (time), 1368
demo, 402, 605, 607, 1451, 1458
dendrapply, 341, 1046
dendrogram, 363, 1035, 1043, 1046, 1048, 1111, 1112, 1221
density, 680, 681, 699, 738, 747, 772, 1002, 1003, 1051, 1229
density-class (setOldClass), 946
deparse, 54, 102, 104, 118, 120, 281, 308, 331, 430, 773
deparseOpts, 103
Deprecated, 35, 101, 105, 284, 1337, 1420, 1552
deriv, 1054, 1195, 1196
deriv3 (deriv), 1054
derivedDefaultMethodWithTrace-class (TraceClasses), 957
det, 105, 127, 326
detach, 29, 106, 232–234, 289, 381, 487
determinant (det), 105
dev.control, 594
dev.control (dev2), 590
dev.copy, 644
dev.copy (dev2), 590
dev.copy2eps (dev2), 590
dev.copy2pdf (dev2), 590
dev.cur, 1, 591, 595
dev.cur (dev.xxx), 588
dev.interactive, 587, 595
dev.list (dev.xxx), 588

dev.new, *588*
dev.new *(dev.xxx)*, 588
dev.next *(dev.xxx)*, 588
dev.off *(dev.xxx)*, 588
dev.prev *(dev.xxx)*, 588
dev.print, *577*, *595*, *626*, *644*
dev.print *(dev2)*, 590
dev.set *(dev.xxx)*, 588
dev.size, 588
dev.xxx, 588
dev2, 590
dev2bitmap, *132*, 592, *595*
devAskNewPage, *301*, 594, *726*, *833*
deviance, 1057, *1058*, *1072*, *1108*, *1150*, *1201*
device *(Devices)*, 594
deviceIsInteractive *(dev.interactive)*, 587
Devices, *577*, *587*, *589*, 594, *596*, *616*, *619*, *626*, *631*, *638*, *650*, *653*, *759*
dexp *(Exponential)*, 1070
df, *1362*
df *(FDist)*, 1080
df.kernel *(kernel)*, 1132
df.residual, *1057*, 1058, *1108*, *1150*, *1201*
dfbeta *(influence.measures)*, 1119
dfbetas, *1149*
dfbetas *(influence.measures)*, 1119
dffits, *1149*
dffits *(influence.measures)*, 1119
dgamma, *1007*, *1023*, *1071*
dgamma *(GammaDist)*, 1097
dgeom, *1193*
dgeom *(Geometric)*, 1100
dget, *120*
dget *(dput)*, 117
dhyper *(Hypergeometric)*, 1117
diag, 108, *249*, *263*
diag<- *(diag)*, 108
diana, *1000*
diff, 109, *1058*, *1059*, *1140*, *1372*
diff.ts, *110*
diff.ts *(ts-methods)*, 1371
diffinv, *110*, 1058
difftime, *96*, *98*, 110, *196*, *296*, *385*, *386*
digamma *(Special)*, 402
dim, *19*, *20*, *31*, *32*, 112, *148*, *210*, *264*, *285*, *457*, *493*
dim<-, *210*
dim<- *(dim)*, 112
dimnames, *19*, *20*, *31*, *32*, *84*, 113, *113*, *140*, *210*, *264*, *279*, *320*, *370*, *371*, *457*, *479*, *483*, *747*
dimnames<-, *210*
dimnames<- *(dimnames)*, 113
dir *(list.files)*, 239
dir.create *(files)*, 157
dirname *(basename)*, 35
discoveries, 517
DISPLAY *(EnvVar)*, 132
displaystyle *(plotmath)*, 619
dist, *1026*, *1035*, 1059, *1061*, *1112*
dlnorm, *1210*
dlnorm *(Lognormal)*, 1159
dlogis *(Logistic)*, 1155
dmultinom *(Multinomial)*, 1187
DNase, 518
dnbinom, *1010*, *1101*, *1241*
dnbinom *(NegBinomial)*, 1192
dnorm, *1160*
dnorm *(Normal)*, 1209
do.call, *46*, 114, *353*
Documentation, 881
Documentation-class *(Documentation)*, 881
Documentation-methods *(Documentation)*, 881
dot *(plotmath)*, 619
dotchart, *672*, 690, *737*
dotsMethods, 883, *918*, *938*, *941*, *944*
double, 115, *116*, *148*, *207*, *291*, *352*, *656*
double-class *(BasicClasses)*, 870
download.file, *48*, *76*, *133*, *241*, *298*, *299*, 1452, *1494*, *1547*, *1548*, *1550*, *1552*
download.packages, *1453*
download.packages *(update.packages)*, 1546
downViewport, *796*, *860*
downViewport *(Working with Viewports)*, 861
dpois, *1010*, *1193*
dpois *(Poisson)*, 1240
dput, *78*, *104*, 117, *120*, *375*, *1454*, *1495*, *1536*
dQuote, *430*
dQuote *(sQuote)*, 410
draw.details *(drawDetails)*, 787
drawDetails, 787
drop, 118, *140*, *263*
drop.scope *(factor.scope)*, 1075
drop.terms *(delete.response)*, 1045

drop1, *119*, *974, 975*, *977*, *1072*, *1076*, *1338, 1339*
drop1 *(add1)*, 965
dsignrank, *1393*
dsignrank *(SignRank)*, 1306
dt, *1019*, *1081*
dt *(TDist)*, 1361
dummy.coef, 1062
dump, *78*, *104*, *117, 118*, 119, *375*
dump.frames, *298*, *1516*
dump.frames *(debugger)*, 1449
dump.frames-class *(setOldClass)*, 946
dumpMethod *(GenericFunctions)*, 891
dumpMethods *(GenericFunctions)*, 891
dunif *(Uniform)*, 1379
duplicated, 121, *480*
duplicated.numeric_version *(numeric_version)*, 293
duplicated.POSIXlt *(DateTimeClasses)*, 95
dweibull, *1071*
dweibull *(Weibull)*, 1386
dwilcox, *1306*
dwilcox *(Wilcoxon)*, 1392
dyn.load, *4*, 122, *162*, *164*, *184, 185*, *188*, *236, 237*, *1444*, *1535*
dyn.unload, *236*
dyn.unload *(dyn.load)*, 122

eapply, 125, *226*
ecdf, *160*, 1063, *1238*, *1283*, *1340*
edit, *104*, *301*, *469, 470*, *1449*, 1454, *1456*, *1459–1461*, *1464*, *1495*, *1554*
edit.data.frame, *1454*, 1455, *1455*, *1460, 1461*, *1553*
edit.matrix *(edit.data.frame)*, 1455
edit.vignette *(vignette)*, 1553
editDetails, 788
editGrob, *790*, *793*
editGrob *(grid.edit)*, 811
EDITOR *(EnvVar)*, 132
eff.aovlist, 1066
effects, *974*, 1067, *1105*, *1108*, *1145, 1146*, *1150*
eigen, 126, *219*, *326*, *436*, *1255*, *1268, 1269*
else, *362*
else *(Control)*, 82
emacs *(edit)*, 1454
embed, 1068
embedFonts, 595, *615, 616*, *629*, *633*
emptyenv *(environment)*, 130
encoded_text_to_latex, 1407
encodeString, *49*, 127, *167*, *282*, *319*
Encoding, *56*, 129, *194*, *202*, *344*, *424, 425*, *431, 432*, *488*
Encoding<- *(Encoding)*, 129
end, *1371*
end *(start)*, 1335
engine.display.list *(grid.display.list)*, 809
env.profile *(environment)*, 130
environment, *22*, *26–29*, *104*, 130, *134, 135*, *137*, *182*, *226*, *238*, *250*, *359*, *442*, *577*, *639*, *1340*, *1431*, *1445*, *1450*
environment variables, *440*, *446*, *1470*
environment variables *(EnvVar)*, 132
environment-class, 885
environment<- *(environment)*, 130
environmentIsLocked *(bindenv)*, 38
environmentName *(environment)*, 130
EnvVar, 132
erase.screen *(screen)*, 758
Error *(aov)*, 982
esoph, 519, *1105*
estVar *(SSD)*, 1328
euro, 520
eurodist, 521
EuStockMarkets, 521
eval, *131*, 134, *139*, *227*, *308*, *402*, *430*, *444*, *639*, *1555*
evalq, *496*
evalq *(eval)*, 134
example, *301*, 1457, *1477*, *1558*
exists, *27*, *131*, 136, *183*, *1481*
existsMethod *(getMethod)*, 896
exp, *1071*
exp *(log)*, 244
expand.grid, 138, *1263*, *1442*
expand.model.frame, 1069, *1181*
expm1 *(log)*, 244
Exponential, 1070, *1387*
expression, *40*, *46*, *82*, *102, 103*, *134, 135*, 139, *142*, *214*, *308*, *430*, *639*, *675*, *708*, *720*, *769*, *775*, *777*, *834*, *842*, *1055, 1056*
expression-class *(BasicClasses)*, 870
extendrange, *338*, 596
extends, *878*, *937*

extends (is), 903
externalptr-class (BasicClasses), 870
Extract, 140, 146, 147, 396, 439
Extract.data.frame, 144
Extract.factor, 146
extractAIC, 967, 971, 972, 1057, 1071, 1337, 1338
Extremes, 148

F (logical), 248
factanal, 1072, 1151, 1192, 1385
factor, 63, 68, 87, 116, 141, 147, 150, 192, 196, 209, 229, 249, 282, 434, 457, 499, 675, 687, 742, 744, 1103, 1181, 1288
factor-class (setOldClass), 946
factor.scope, 1075
factorial (Special), 402
faithful, 522
FALSE, 362
FALSE (logical), 248
family, 1076, 1102, 1104, 1156, 1167, 1244
family.glm (glm.summaries), 1107
family.lm (lm.summaries), 1149
fdeaths (UKLungDeaths), 563
FDist, 1080
fft, 1034, 1052, 1081, 1194, 1318
fifo (connections), 74
file, 202, 308, 343, 347, 349, 351, 352, 377, 379, 390, 401, 413, 465, 498, 1512, 1514, 1524
file (connections), 74
file.access, 152, 155, 158, 159, 240
file.append (files), 157
file.choose, 154, 240
file.copy (files), 157
file.create (files), 157
file.edit, 1459, 1553
file.exists, 1460
file.exists (files), 157
file.info, 153, 154, 159, 240, 295, 1409, 1460
file.path, 35, 155, 159, 1409, 1460
file.remove, 481
file.remove (files), 157
file.rename (files), 157
file.show, 156, 159, 298, 1459, 1469, 1495, 1550
file.symlink (files), 157
file_path_as_absolute (fileutils), 1408
file_path_sans_ext (fileutils), 1408
file_test, 159, 1459
files, 155, 157, 157, 240, 1459
fileutils, 1408
filled.contour, 567, 664, 684, 691, 705, 726
Filter (funprog), 177
filter, 999, 1034, 1082, 1131
Find (funprog), 177
find, 251, 893
find (apropos), 1428
findClass, 886
findFunction (GenericFunctions), 891
findInterval, 87, 159, 258
findMethod (getMethod), 896
findMethods, 888
findMethodSignatures (findMethods), 888
findRestart (conditions), 70
finite (is.finite), 211
fisher.test, 1084
fitted, 1086, 1108, 1146, 1150, 1192, 1201
fitted.values, 974, 1027, 1105, 1295, 1307
fivenum, 575, 1087, 1127, 1283
fix, 1454, 1455, 1459, 1460, 1464, 1495
fixInNamespace (getFromNamespace), 1463
fixPre1.8, 889
fligner.test, 981, 1005, 1088, 1187
floor (Round), 365
flush (connections), 74
flush.console, 1461
for, 362, 1498
for (Control), 82
for-class (language-class), 909
force, 135, 160
Foreign, 161, 1402
Formaldehyde, 523
formals, 17, 164, 238, 239, 280
formals<- (formals), 164
format, 49, 50, 54, 165, 170, 173, 174, 264, 316, 317, 434, 468, 1060, 1272, 1461, 1537, 1557
format.Date, 95, 166, 168
format.difftime (difftime), 110
format.dist (dist), 1059
format.ftable (read.ftable), 1285
format.hexmode (hexmode), 200
format.info, 167, 170

format.octmode (octmode), 294
format.POSIXct, 24, 166
format.POSIXct (strptime), 419
format.POSIXlt, 24
format.POSIXlt (strptime), 419
format.pval, 171, 1272
formatC, 167, 170, 172, 409, 1537
formatDL, 174, 1417, 1462
formatOL (format), 1461
formatUL (format), 1461
formula, 25, 63, 130, 331, 466, 684, 742, 745, 1001, 1055, 1090, 1092, 1102, 1143, 1144, 1152, 1179–1181, 1200, 1201, 1365, 1367, 1396
formula-class (setOldClass), 946
formula.lm (lm.summaries), 1149
formula.nls, 1092
forwardsolve (backsolve), 33
fourfoldplot, 694
frac (plotmath), 619
frame, 695, 1495
frameGrob (grid.frame), 812
freeny, 524, 1146
frequency, 1320, 1371
frequency (time), 1368
friedman.test, 1092, 1281
ftable, 455, 968, 1094, 1097, 1286, 1467
ftable.default, 1096, 1097
ftable.formula, 1095, 1096, 1096
function, 23, 40, 46, 130, 139, 165, 176, 211, 362, 687, 738
function-class (BasicClasses), 870
functionWithTrace-class (TraceClasses), 957
funprog, 177
fuzzy matching, 1473
fuzzy matching (agrep), 7

Gamma, 1156
Gamma (family), 1076
gamma, 36, 1098, 1099
gamma (Special), 402
gammaCody, 404
GammaDist, 1097
gaussian, 1156
gaussian (family), 1076
gc, 179, 181, 268, 270, 354, 452
gc.time, 180, 322
gcinfo, 268
gcinfo (gc), 179
gctorture, 180, 181
gEdit, 789
gEditList (gEdit), 789
genericFunction, 916
genericFunction-class, 890
GenericFunctions, 865, 891, 898, 912, 953
genericFunctionWithTrace-class (TraceClasses), 957
Geometric, 1100
get, 27, 131, 137, 182, 261, 286, 313, 577, 887, 1463–1465, 1481
get.gpar (gpar), 790
get_all_vars (model.frame), 1179
getAllConnections (showConnections), 389
getAnywhere, 1462, 1486
getCConverterDescriptions (getNumCConverters), 188
getCConverterStatus (getNumCConverters), 188
getClass, 877, 880, 895, 927, 955
getClassDef, 880, 959
getClassDef (getClass), 895
getClasses (findClass), 886
getConnection (showConnections), 389
getCRANmirrors (chooseCRANmirror), 1437
getDepList, 1410
getDLLRegisteredRoutines, 183, 185, 188
geterrmessage, 477, 1449
geterrmessage (stop), 416
getFromNamespace, 1463, 1463
getGeneric, 892, 918, 941
getGenerics, 922, 923
getGenerics (GenericFunctions), 891
getGraphicsEvent, 597
getGrob, 793, 798, 812, 814, 837
getGrob (grid.get), 813
getGroupMembers, 931
getHook (UserHooks), 486
getInitial, 1101
getLoadedDLLs, 124, 184, 185, 237
getMethod, 894, 896, 1470
getMethods (findMethods), 888
getNames, 790
getNativeSymbolInfo, 185, 186
getNumCConverters, 188
getOption, 91, 166, 170, 318, 421, 490, 587, 589, 625, 1469, 1504
getOption (options), 297
getPackageName, 898, 911

getpid, 189
getRversion, *332*
getRversion *(numeric_version)*, 293
getS3method, *486*, *1464*, 1465, *1486, 1487*
getSlots *(slot)*, 954
getSrcLines *(srcfile)*, 412
getTaskCallbackNames, *459*, *461*
getTaskCallbackNames *(taskCallbackNames)*, 462
gettext, 190, *272, 273*, *407*, *409*, *416, 417*, *490, 491*, *1424*
gettextf, *1424*
gettextf *(sprintf)*, 407
getTxtProgressBar *(txtProgressBar)*, 1544
getValidity *(validObject)*, 958
getwd, 191, *240*, *377*, *440*, *1512*
gl, *152*, 192, *387*
gList *(grid.grob)*, 816
glm, *434*, *966*, *974, 975*, *1027*, *1032, 1033*, *1057, 1058*, *1076*, *1078*, *1087*, *1090, 1091*, 1102, *1106–1108*, *1120, 1121*, *1127*, *1146*, *1148*, *1150*, *1159*, *1167*, *1178*, *1190*, *1212*, *1232*, *1258*, *1295*, *1307*, *1337, 1338*, *1348*, *1350*, *1364, 1365*, *1386*, *1388*
glm-class *(setOldClass)*, 946
glm.control, *1103*, 1106
glm.fit, *1106, 1107*
glm.null-class *(setOldClass)*, 946
glm.summaries, 1107
glob2rx, *195*, *240*, *250, 251*, *358*, *1429*, 1465
globalenv, *22*
globalenv *(environment)*, 130
gpar, *622*, *629*, *632*, *649*, *651*, 790, *826*
gPath, 792, *829, 830*
graphics *(graphics-package)*, 659
graphics-package, 659
graphics.off, *595*
graphics.off *(dev.xxx)*, 588
gray, *583*, 599, *600*, *606*, *612, 613*, *641*, *731*
gray.colors, 600, *680*, *762*
grconvertX *(convertXY)*, 685
grconvertY *(convertXY)*, 685
grDevices *(grDevices-package)*, 573
grDevices-package, 573
gregexpr *(grep)*, 193
grep, *8*, *54, 55*, 193, *251*, *311*, *355*, *358*, *424*, *1474*, *1531*
grey, *581*
grey *(gray)*, 599
grey.colors *(gray.colors)*, 600
Grid, 793, *796*, *799*, *802, 803*, *808*, *816*, *819*, *821*, *825–827*, *831, 832*, *836*, *838*, *840*, *842, 843*, *845*, *847*, *849*
grid, *573*, *659*, 696, *1231*
Grid Viewports, 794
grid-package, 783
grid.add, 797
grid.arrows, 798
grid.circle, 801
grid.clip, 802
grid.collection, 803
grid.convert, *786*, 804
grid.convertHeight *(grid.convert)*, 804
grid.convertWidth *(grid.convert)*, 804
grid.convertX *(grid.convert)*, 804
grid.convertY *(grid.convert)*, 804
grid.copy, 806
grid.curve, 807
grid.display.list, 809
grid.draw, *788*, 810, *817*
grid.edit, *789*, 811, *817*, *829, 830*, *860*
grid.frame, 812, *829, 830*
grid.gedit *(grid.edit)*, 811
grid.get, 813, *817*
grid.gget *(grid.get)*, 813
grid.grab, 814
grid.grabExpr *(grid.grab)*, 814
grid.gremove *(grid.remove)*, 836
grid.grill, 815
grid.grob, *804*, *806*, 816, *839*
grid.layout, *793*, *796*, 818, *840*, *855*
grid.line.to, *799*
grid.line.to *(grid.move.to)*, 824
grid.lines, *799*, 819
grid.locator, 821
grid.ls, 822
grid.move.to, 824
grid.newpage, *594*, 826, *834*
grid.null, 826
grid.pack, *813*, 827, *830*
grid.place, 829, *829*
grid.plot.and.legend, 830
grid.points, 830
grid.polygon, 831
grid.polyline *(grid.lines)*, 819
grid.pretty, 833
grid.prompt, 833, *1457*
grid.record, 834
grid.rect, 835

grid.refresh, 836
grid.remove, 836
grid.segments, *799*, 837
grid.set, 839
grid.show.layout, *796*, *819*, 840
grid.show.viewport, 841
grid.text, 842
grid.xaxis, 844, *849*
grid.xspline, *808*, 845
grid.yaxis, *845*, 848
grob, *790*, *793*, *798*, *810*, *812*, *814*, *824*, *837*
grob *(grid.grob)*, 816
grobHeight *(grobWidth)*, 849
grobName, 849
grobPathListing *(grid.ls)*, 822
grobTree *(grid.grob)*, 816
grobWidth, 849, *850*, *854*
grobX, 850, *864*
grobY, *864*
grobY *(grobX)*, 850
group *(plotmath)*, 619
group generic, *61*, *151*, *210*, *484*
group generic *(groupGeneric)*, 196
groupGeneric, 196
groupGenericFunction-class *(genericFunction-class)*, 890
GroupGenericFunctions, *872*, *914*
GroupGenericFunctions *(S4groupGeneric)*, 930
groupGenericFunctionWithTrace-class *(TraceClasses)*, 957
gsub, *56*, *129*
gsub *(grep)*, 193
gTree, *815*
gTree *(grid.grob)*, 816
gzcon, *76*, *78*, 198
gzfile, *199*
gzfile *(connections)*, 74

HairEyeColor, 525
Harman23.cor, 526, *1075*
Harman74.cor, 526, *1075*, *1385*
hasArg, 899
hasMethod *(getMethod)*, 896
hasMethods *(findMethods)*, 888
hasTsp *(tsp)*, 1375
hat, *1149*, *1163*, *1233*
hat *(influence.measures)*, 1119
hat *(plotmath)*, 619
hatvalues, *1233*
hatvalues *(influence.measures)*, 1119
hcl, *583*, *599*, 601, *606*, *613*, *641*, *731*
hclust, *1000*, *1035*, *1043*, *1061*, 1108, *1112*, *1113*, *1119*, *1287*
head, 1466
heat.colors, *582*, *583*, *704*, *705*
heat.colors *(Palettes)*, 612
heatmap, *705*, 1111, *1289*
heightDetails, *784*
heightDetails *(widthDetails)*, 861
help, *17*, *132*, *133*, *157*, *301*, *881*, *1447*, *1457*, 1468, *1475–1477*, *1499*, *1558*
help.request, 1471
help.search, *358*, *1429*, *1471*, 1473, *1527*
help.start, *301*, *1469*, *1471*, 1475, *1475*, *1483*, *1527*
Hershey, 603, *607*, *684*, *727*, *776*, *792*
hexmode, 200, *295*
hist, *611*, *672*, 697, *700*, *701*, *746–748*, *756*, *761–763*, *1053*
hist.Date, *95*
hist.Date *(hist.POSIXt)*, 700
hist.default, *701*
hist.POSIXt, 700
history *(savehistory)*, 1530
HoltWinters, 1114, *1230*, *1260*
HOME *(EnvVar)*, 132
hsearch-class *(setOldClass)*, 946
hsv, *583*, *599*, *601*, *602*, 606, *612*, *613*, *641*, *642*, *647*, *705*, *731*
Hyperbolic, 200
Hypergeometric, 1117

I, *21*, *91*, *92*, *1090*, *1091*, *1556*
I *(AsIs)*, 25
iconv, *48*, *77*, *129*, 201, *652*, *1407*, *1419*, *1481*, *1492*
iconvlist, *413*
iconvlist *(iconv)*, 201
icuSetCollate, 203
identical, *10*, *11*, *68*, 204, *212*
identify, 702, *714*, *1119*
identify.hclust, *1110*, *1111*, 1118, *1287*
identity, 206
if, *207*, *247*, *307*, *362*
if *(Control)*, 82
if-class *(language-class)*, 909
ifelse, *83*, 206
Im *(complex)*, 69
image, *593*, *595*, *650*, *684*, *693*, 704, *726*, *734*, *748*, *1111–1113*, *1358*
implicit generic, *234*

implicit generic (implicitGeneric), 900
implicitGeneric, 900, 917, 941
in (Control), 82
index.search, 1476
Indometh, 527
Inf, 18, 160, 162, 292, 362, 1087
Inf (is.finite), 211
inf (plotmath), 619
infert, 528, 1105
influence, 1121, 1122, 1150, 1233, 1389
influence (lm.influence), 1148
influence.measures, 1108, 1119, 1148, 1149, 1151, 1295
inherits, 879
inherits (class), 60
initialize, 878, 881, 903, 907, 939, 958
initialize (new), 920
initialize,ANY-method (initialize-methods), 902
initialize,array-method (StructureClasses), 956
initialize,data.frame-method (setOldClass), 946
initialize,environment-method (initialize-methods), 902
initialize,factor-method (setOldClass), 946
initialize,matrix-method (StructureClasses), 956
initialize,mts-method (StructureClasses), 956
initialize,ordered-method (setOldClass), 946
initialize,signature-method (initialize-methods), 902
initialize,summary.table-method (setOldClass), 946
initialize,table-method (setOldClass), 946
initialize,traceable-method (initialize-methods), 902
initialize,ts-method (StructureClasses), 956
initialize-methods, 921
initialize-methods, 902
InsectSprays, 529
INSTALL, 233, 234, 899, 1477, 1479, 1492, 1497, 1519, 1534, 1549, 1550
install.packages, 234, 301, 1411, 1412, 1492, 1494, 1520, 1534
install.packages (update.packages), 1546
installed.packages, 233, 234, 1410, 1478, 1494, 1504, 1548, 1550
installFoundDepends, 1411, 1411
Insurance, 1212
integer, 63, 90, 112, 117, 148, 170, 207, 228, 262, 285, 291, 292, 334, 494, 516
integer-class (BasicClasses), 870
integral (plotmath), 619
integrate, 1122
integrate-class (setOldClass), 946
interaction, 63, 208
interaction.plot, 743, 1124
interactive, 209, 298
Internal, 210
internal generic, 61, 198, 218, 484, 485
internal generic (InternalMethods), 210
InternalGenerics, 484
InternalGenerics (InternalMethods), 210
InternalMethods, 20, 54, 141, 151, 210, 212, 228, 264, 276, 291, 482, 1371, 1415
interpSpline, 1323
intersect (sets), 387
intToBits (rawConversion), 344
intToUtf8, 129
intToUtf8 (utf8Conversion), 487
inverse.gaussian, 1156
inverse.gaussian (family), 1076
inverse.rle (rle), 364
invisible, 176, 211, 315, 611, 952, 1119, 1555
invokeRestart (conditions), 70
invokeRestartInteractively (conditions), 70
IQR, 610, 1088, 1126, 1165
iris, 529
iris3 (iris), 529
is, 61, 489, 877, 880, 903, 933
is.array, 210
is.array (array), 19
is.atomic, 379
is.atomic (is.recursive), 216
is.call (call), 45
is.character (character), 53
is.complex (complex), 69
is.data.frame (as.data.frame), 20

is.double, *291*
is.double (*double*), 115
is.element, *258*
is.element (*sets*), 387
is.empty.model, 1127
is.environment (*environment*), 130
is.expression (*expression*), 139
is.factor (*factor*), 150
is.finite, 211
is.function, 213
is.infinite (*is.finite*), 211
is.integer (*integer*), 207
is.language, *46*, 214, *216*, *278*
is.leaf (*dendrogram*), 1048
is.list, *216*, *489*
is.list (*list*), 237
is.loaded, *186*, *188*
is.loaded (*dyn.load*), 122
is.logical (*logical*), 248
is.matrix, *210*
is.matrix (*matrix*), 263
is.mts (*ts*), 1370
is.na, *151*, *210*, *1027*
is.na (*NA*), 276
is.na.POSIXlt (*DateTimeClasses*), 95
is.na<- (*NA*), 276
is.na<-.factor (*factor*), 150
is.name (*name*), 277
is.nan, *210*, *277*
is.nan (*is.finite*), 211
is.null (*NULL*), 290
is.numeric, *93*, *210*, *372*, *489*
is.numeric (*numeric*), 290
is.numeric_version (*numeric_version*), 293
is.object, *61*, *210*, 214, *218*, *484*, *486*, *574*, *956*
is.ordered (*factor*), 150
is.package_version (*numeric_version*), 293
is.pairlist (*list*), 237
is.primitive (*is.function*), 213
is.qr (*qr*), 325
is.R, 215
is.raw (*raw*), 341
is.real (*real*), 352
is.recursive, *141*, 216
is.relistable (*relist*), 1517
is.single, 217
is.stepfun (*stepfun*), 1339
is.symbol, *51*
is.symbol (*name*), 277
is.table (*table*), 454
is.ts (*ts*), 1370
is.tskernel (*kernel*), 1132
is.unsorted, 217, *400*
is.vector (*vector*), 488
isClass, *875*, *895*, *896*
isClass (*findClass*), 886
isClassUnion (*setClassUnion*), 936
isGeneric, *918*, *941*
isGeneric (*GenericFunctions*), 891
isGroup (*GenericFunctions*), 891
isIncomplete, *464*
isIncomplete (*connections*), 74
islands, 531
ISOdate (*strptime*), 419
ISOdatetime (*strptime*), 419
isoMDS, *1026*, *1129*
isOpen (*connections*), 74
isoreg, 1128, *1230*, *1231*
isRestart (*conditions*), 70
isS4, *214*, 218, *396*, *479*, *929*, *956*
isSealedClass (*isSealedMethod*), 908
isSealedMethod, 908
isSeekable (*seek*), 381
isSymmetric, 219
isTRUE, *11*, *205*
isTRUE (*Logic*), 246
isXS3Class (*S3Part*), 927
italic (*plotmath*), 619

Japanese, *605*, 607
jitter, 220, *757*, *771*
JohnsonJohnson, 531, *1376*
jpeg, *47*, *48*, *593*, *595*
jpeg (*png*), 623
julian (*weekdays*), 492

KalmanForecast, *1257*
KalmanForecast (*KalmanLike*), 1129
KalmanLike, *992*, 1129, *1345*, *1346*
KalmanRun (*KalmanLike*), 1129
KalmanSmooth, *1376*
KalmanSmooth (*KalmanLike*), 1129
kappa, 221
kernapply, 1131, *1133*
kernel, *1131*, 1132
kmeans, *1111*, 1133
knots, *1237*, *1340*
knots (*stepfun*), 1339
kronecker, 222, *306*
kruskal.test, 1135, *1213*, *1391*

ks.test, 1137
ksmooth, 1139

l10n_info, 224, *244*
La.svd *(svd)*, 435
labels, 224, *1050*, *1150*, *1365*
labels.dendrogram *(order.dendrogram)*, 1221
labels.dist *(dist)*, 1059
labels.lm *(lm.summaries)*, 1149
labels.terms *(terms)*, 1365
lag, 1140
lag.plot, 1141
LakeHuron, 532
LANGUAGE *(EnvVar)*, 132
language-class, 909
lapply, *16*, *125*, 225, *261*, *340*, *341*, *458*, *970*, *1047*
Last.value, 227
last.warning *(warnings)*, 491
layout, *589*, *610*, 706, *729*, *731*, *759*, *819*, *1113*
lbeta *(Special)*, 402
lchoose *(Special)*, 402
lcm *(layout)*, 706
ldeaths *(UKLungDeaths)*, 563
legend, *139*, *619*, 708, *756*, *1125*
length, *210*, 227
length<-, *210*
length<- *(length)*, 227
LETTERS *(Constants)*, 80
letters *(Constants)*, 80
levelplot, *684*, *693*, *705*
levels, *31*, *32*, *152*, 228, *249*, *282*, *1288*, *1290*
levels<-, *210*
levels<- *(levels)*, 228
lfactorial *(Special)*, 402
lgamma *(Special)*, 402
lh, 532
libPaths, 230
library, *29*, *33*, *107*, 231, *231*, *237*, *287*, *288*, *298*, *380*, *447*, *487*, *500*, *893*, *899*, *1443*, *1471*, *1478*, *1502*, *1504*, *1550*
library.dynam, *124*, *125*, *184*, *233*, *234*, 235, *1535*
library.dynam.unload, *107*, *124*
libraryIQR-class *(setOldClass)*, 946
licence *(license)*, 237
license, *83*, 237
LifeCycleSavings, 533, *1146*
limitedLabels *(recover)*, 1515
line, 1142
linearizeMlist, *910*, *911*, *919*
LinearMethodsList-class, 910
lines, *654*, *660*, *682*, *690*, *697*, 712, *716*, *724*, *726–729*, *734*, *738*, *745*, *746*, *749*, *750*, *752*, *754*, *760*, *780*, *1230*, *1239*, *1364*
lines.formula *(plot.formula)*, 744
lines.histogram *(plot.histogram)*, 746
lines.isoreg *(plot.isoreg)*, 1230
lines.stepfun *(plot.stepfun)*, 1237
lines.ts *(plot.ts)*, 1239
linesGrob *(grid.lines)*, 819
lineToGrob *(grid.move.to)*, 824
LINK, 1479
link-glm, *1077*
list, *107*, *143*, *182*, 237, *276*, *297*, *326*, *331*, *377*, *457*, *475*, *1230*, *1232*, *1311*, *1418*, *1442*, *1488*, *1518*, *1534*
list-class *(BasicClasses)*, 870
list.files, *154*, *155*, *157*, *159*, *192*, 239, *358*, *451*, *1409*
list_files_with_exts *(fileutils)*, 1408
list_files_with_type *(fileutils)*, 1408
listFromMlist, *919*
lm, *301*, *434*, *482*, *966*, *967*, *976*, *977*, *983*, *1018*, *1027*, *1028*, *1032*, *1033*, *1057*, *1058*, *1067*, *1071*, *1076*, *1087*, *1090*, *1091*, *1105*, *1120*, *1127*, *1136*, 1143, *1143*, *1146–1150*, *1163*, *1165*, *1190*, *1232*, *1245*, *1262*, *1275*, *1295*, *1307*, *1337*, *1350*, *1351*, *1364*, *1365*, *1388*, *1389*, *1536*
lm-class *(setOldClass)*, 946
lm.fit, *326*, *1144*, 1146, *1146*
lm.influence, *1121*, *1122*, *1146*, 1148, *1163*, *1233*, *1389*
lm.summaries, 1149
lm.wfit, *1146*
lm.wfit *(lm.fit)*, 1146
lme, *982*
load, *29*, *78*, 241, *375*, *1446*
loadedNamespaces, *381*
loadedNamespaces *(ns-load)*, 288
loadhistory *(savehistory)*, 1530
loadings, 1151, *1269*
loadNamespace, *287*, *487*
loadNamespace *(ns-load)*, 288

Loblolly, 534
local, *353*, *414*
local *(eval)*, 134
localeconv, *224*, 242
locales, *67*, *169*, 243, *356*, *422*
localeToCharset, *202*, *402*, 1480
locator, *703*, *708*, 714
lockBinding, *26*, *464*
lockBinding *(bindenv)*, 38
lockEnvironment, *359*
lockEnvironment *(bindenv)*, 38
loess, 1152, *1154*, *1162*, *1263*, *1298*, *1309*, *1343*
loess.control, *1152*, *1153*, 1154
loess.smooth *(scatter.smooth)*, 1297
log, 244, *262*, *931*
log10 *(log)*, 244
log1p *(log)*, 244
log2 *(log)*, 244
logb *(log)*, 244
Logic, 246, *247*, *342*, *439*, *494*
Logic *(S4groupGeneric)*, 930
logical, 248, *248*, *262*, *418*, *493*
logical-class *(BasicClasses)*, 870
Logistic, 1155
logLik, *971*, *972*, *1071*, *1078*, 1156, *1201*
logLik-class *(setOldClass)*, 946
logLik.gls, *1157*
logLik.lme, *1157*
loglin, *525*, *718*, *719*, *1105*, 1157
loglm, *1105*, *1159*
Lognormal, 1159
longley, 534, *1146*
lower.tri, *109*, 249
lowess, *654*, *724*, *1153*, 1161, *1309*
lqs, *1180*
ls, *131*, 250, *313*, *358*, *359*, *1431*, *1481*
ls.diag, 1162, *1163*, *1165*
ls.print, 1163, *1163*, *1165*
ls.str, *131*, *251*, 1481, *1538*
lsf.str *(ls.str)*, 1481
lsfit, *326*, *327*, *1162*, *1163*, 1164
lynx, 535

mad, *610*, *1127*, 1165, *1300*
mahalanobis, 1166
make.link, *1077*, *1078*, 1167, *1244*
make.names, *20*, *26*, *91*, *92*, 251, *253*, *363*, *1507*, *1513*
make.packages.html, *1476*, 1482
make.rgb, *585*, *586*, 608
make.socket, *48*, *1441*, 1483, *1511*
make.unique, *145*, *251*, 252, *252*, *1491*
makeActiveBinding *(bindenv)*, 38
makeARIMA *(KalmanLike)*, 1129
makeClassRepresentation, *887*, 911, *935*
makeLazyLoading, 1412
makepredictcall, 1168
makepredictcall.poly *(poly)*, 1242
manglePackageName, 253
manova, 1169, *1353*
mantelhaen.test, 1170
maov-class *(setOldClass)*, 946
Map *(funprog)*, 177
mapply, *178*, *226*, 254, *458*
margin.table, 255, *323*, *455*, *968*
mat.or.vec, 256
match, *55*, *145*, *195*, 257, *271*, *311*, *494*
match.arg, 258, *258*, *260*, *261*, *311*, *1061*
match.call, *226*, 259, *259*, *311*, *897*
match.fun, *15*, *125*, *225*, *254*, *259*, 260, *260*, *305*, *311*, *436*
Math, *86*, *151*, *201*, *245*, *262*, *366*, *392*, *404*, *476*, *1415*
Math *(S4groupGeneric)*, 930
Math *(groupGeneric)*, 196
Math,structure-method *(StructureClasses)*, 956
Math.data.frame, *92*
Math.Date *(Dates)*, 94
Math.difftime *(difftime)*, 110
Math.factor *(factor)*, 150
Math.POSIXlt *(DateTimeClasses)*, 95
Math.POSIXt *(DateTimeClasses)*, 95
Math2, *366*
Math2 *(S4groupGeneric)*, 930
MathFun, 261
matlines *(matplot)*, 715
matmult, 262
matplot, *530*, 715
matpoints *(matplot)*, 715
matrix, *20*, *93*, *109*, *113*, *114*, *143*, *219*, *249*, 263, *263*, *285*, *688*, *716*, *877*, *1441*, *1442*
matrix-class *(StructureClasses)*, 956
mauchly.test, 1172, *1328*
max, *338*, *494*, *495*, *984*
max *(Extremes)*, 148
max.col, *495*
max.col *(maxCol)*, 265
maxCol, 265
mcnemar.test, 1174

md5sum, *1402*, 1413
mdeaths (*UKLungDeaths*), 563
mean, *64*, 266, *984*, *1002*, *1362*, *1388*
mean.Date (*Dates*), 94
mean.difftime (*difftime*), 110
mean.POSIXct, *267*
mean.POSIXct (*DateTimeClasses*), 95
mean.POSIXlt (*DateTimeClasses*), 95
median, *1002*, *1088*, *1165*, 1175, *1177*, *1309*
medpolish, 1176
mem.limits, *1474*
mem.limits (*Memory*), 267
Memory, *180*, 267, *269*, *298*, *354*, *415*
Memory-limits, *268*, *1490*
Memory-limits, 269, *1484*
memory.limit (*memory.size*), 1484
memory.profile, *268*, 269
memory.size, 1484
menu, *1437*, 1485, *1532*
merge, 270
message, 272, *393*, *491*, *1424*
method.skeleton, 912, *943, 944*
MethodDefinition, *897, 898*, *918*, *920*, *953, 954*
MethodDefinition-class, 913
MethodDefinitionWithTrace-class (*TraceClasses*), 957
Methods, *100*, *338*, *865*, *872*, *874*, *880*, *883, 884*, *887*, *897, 898*, *903*, 914, *919*, *931*, *934, 935*, *938*, *941*, *944*, *947*, *955*, *1487*
methods, *11*, *198*, *210*, *214*, *251*, *283*, *316*, *433*, *484*, *486*, *1107*, *1149*, *1348*, *1465*, *1471*, 1486
methods-package, 865
MethodsList, *911*, *914*
MethodsList-class, *889*
MethodsList-class, 918
MethodWithNext, *914*
MethodWithNext-class, 919
MethodWithNextWithTrace-class (*TraceClasses*), 957
mget (*get*), 182
min, *338*, *494*, *984*
min (*Extremes*), 148
mirror2html (*mirrorAdmin*), 1487
mirrorAdmin, 1487
missing, 273, *430*, *900*
missing-class (*BasicClasses*), 870
mlm-class (*setOldClass*), 946
Mod, *11*, *262*
Mod (*complex*), 69
mode, *11*, *61*, *116*, *137*, *182, 183*, 274, *291*, *388*, *430*, *479*, *484*, *1428*, *1481*
mode<- (*mode*), 274
model.extract, 1178, *1182*
model.frame, *948*, *980*, *1004*, *1039*, *1069*, *1073*, *1088*, *1091*, *1093*, *1096*, *1105*, *1135*, *1145*, *1168*, *1178*, 1179, *1181, 1182*, *1186*, *1189*, *1212, 1213*, *1250*, *1253*, *1268*, *1280*, *1360*, *1384*, *1390*, *1396*
model.frame.default, *1168*
model.matrix, *21*, *91*, *1144*, 1181, *1181*, *1366*, *1383*
model.matrix.default, *1144*
model.offset, *1103*, *1144*, *1212*
model.offset (*model.extract*), 1178
model.response (*model.extract*), 1178
model.tables, *982, 983*, *1063*, 1182, *1275*, *1291*, *1301*, *1347*, *1378*
model.tables.aovlist, *1066*
model.weights (*model.extract*), 1178
modifyList, 1488
month.abb (*Constants*), 80
month.name (*Constants*), 80
monthplot, 1184
months (*weekdays*), 492
mood.test, *981*, *1005*, *1089*, 1186, *1384*
morley, 536
mosaic, *719*
mosaicplot, *525*, *663*, *695*, 717, *747*, *763*
mostattributes<- (*attributes*), 31
moveToGrob (*grid.move.to*), 824
mtable-class (*setOldClass*), 946
mtcars, 537
mtext, *619*, *622*, *682*, *688*, 720, *726*, *728*, *776*, *778*
mts-class (*setOldClass*), 946
Multinomial, 1187
mvfft (*fft*), 1081

n2mfrow, 609, *1141*
NA, *35*, *67*, *110*, *150, 151*, *160*, *162*, *212*, *228*, *247*, *249*, *274*, 276, *301*, *316*, *318*, *338, 339*, *362*, *378*, *493*, *574*, *580*, *653*, *656*, *674*, *696*, *1037*, *1087*, *1112*, *1189, 1190*, *1222*, *1250*, *1272*, *1279*, *1282*, *1357*, *1507*, *1513*, *1544, 1545*
na.action, *277*, *1148*, 1188, *1190*, *1192*
na.contiguous, 1189, *1190*, *1372*

na.exclude, *1102*, *1143*, *1148*, *1149*, *1192*, *1200*
na.exclude *(na.fail)*, 1190
na.fail, *277*, *1027*, *1069*, *1102*, *1143*, *1179*, *1189*, 1190, *1200*, *1253*, *1268*, *1372*
na.omit, *277*, *1027*, *1069*, *1102*, *1143*, *1179*, *1189*, *1192*, *1200*, *1253*, *1268*, *1372*
na.omit *(na.fail)*, 1190
na.omit.ts, *1189*
na.omit.ts *(ts-methods)*, 1371
na.pass, *1180*
na.pass *(na.fail)*, 1190
NA_character_, *309*, *362*
NA_character_ *(NA)*, 276
NA_complex_, *362*
NA_complex_ *(NA)*, 276
NA_integer_, *18*, *292*, *362*
NA_integer_ *(NA)*, 276
NA_real_, *205*, *292*, *362*
NA_real_ *(NA)*, 276
name, *49*, *134*, *140*, *214*, *232*, 277, *1451*, *1468*
name-class *(language-class)*, 909
names, *10*, *19*, *31*, *32*, *84*, *92*, *108*, *113*, *140*, *143*, *148*, *210*, *225*, *252*, 278, *370*, *479*, *483*, *589*, *1282*
names<-, *210*
names<- *(names)*, 278
NaN, *18*, *67*, *162*, *205*, *276*, *277*, *292*, *362*, *574*, *1087*, *1282*
NaN *(is.finite)*, 211
napredict, *1074*, *1087*, *1190*, *1258*, *1261*, *1268*
napredict *(naresid)*, 1191
naprint, 1191
naresid, *1108*, *1148*, *1150*, *1190*, 1191, *1295*
nargs, 280
NativeSymbol, *161*
NativeSymbol *(getNativeSymbolInfo)*, 186
NativeSymbolInfo, *161*, *1401*
NativeSymbolInfo *(getNativeSymbolInfo)*, 186
nchar, *166*, 281, *309*, *424*, *432*, *468*, *770*
nclass, 610
nclass.FD, *699*
nclass.scott, *699*
nclass.Sturges, *699*
NCOL, *370*
NCOL *(nrow)*, 285
ncol, *113*
ncol *(nrow)*, 285
Negate *(funprog)*, 177
NegBinomial, 1192
nestedListing *(grid.ls)*, 822
new, *878*, *880*, *902*, 920, *927*, *929*, *946*, *959*
new.env, *886*
new.env *(environment)*, 130
new.packages *(update.packages)*, 1546
next, *362*
next *(Control)*, 82
NextMethod, *61*
NextMethod *(UseMethod)*, 484
nextn, *1034*, *1082*, 1194
ngettext, *1424*
ngettext *(gettext)*, 190
nhtemp, 537
Nile, 538, *1376*
nlevels, *152*, *229*, 282
nlm, *1056*, 1195, *1198*, *1217*, *1220*, *1381*, *1517*
nlminb, *1196*, 1197, *1217*
nls, *187*, *1087*, *1092*, *1101*, *1196*, 1199, *1205*, *1235*, *1265*, *1274*, *1295*, *1303*, *1325–1327*, *1330–1333*, *1335*, *1354*
nls.control, *1200*, *1201*, 1204
NLSstAsymptotic, 1206
NLSstClosestX, 1206, *1207*, *1208*, *1315*
NLSstLfAsymptote, 1207, *1207*, *1315*
NLSstRtAsymptote, *1207*, 1208, *1208*, *1315*
nonstandardGenericWithTrace-class *(TraceClasses)*, 957
noquote, 283, *316*, *319*, *1358*, *1400*
Normal, 1209
normalizePath, 1488
nottem, 539
NotYet, 284
NotYetImplemented *(NotYet)*, 284
NotYetUsed *(NotYet)*, 284
NROW, *370*
NROW *(nrow)*, 285
nrow, *113*, 285
ns, *1168*
ns-dblcolon, 286
ns-hooks, 287
ns-load, 288
ns-topenv, 289
nsl, 1489
NULL, *285*, 290, *362*, *418*, *642*, *1112*, *1113*, *1534*
NULL-class *(BasicClasses)*, 870
nullGrob *(grid.null)*, 826

numeric, *63*, *116, 117*, *208*, 290, *291*, *338*
numeric-class *(BasicClasses)*, 870
numeric_version, 293
NumericConstants, *81*, 292, *379*, *439*
numericDeriv, 1211
nzchar *(nchar)*, 281

object.size, *268, 269*, 1490
objects, *29*, *107*, *234*, *359*, *381*, *1429*
objects *(ls)*, 250
ObjectsWithPackage-class, 922
occupationalStatus, 540
octmode, *200*, 294
offset, *1091*, *1103*, *1144*, *1178*, 1212, *1367*
old.packages *(update.packages)*, 1546
oldClass, *197*, *947*
oldClass *(class)*, 60
oldClass-class *(setOldClass)*, 946
oldClass<- *(class)*, 60
on.exit, *176*, 295, *392*, *443*, *1483*
oneway.test, 1212
open *(connections)*, 74
open.srcfile *(srcfile)*, 412
open.srcfilecopy *(srcfile)*, 412
Ops, *18*, *67*, *111*, *151*, *247*, *296*, *485*
Ops *(S4groupGeneric)*, 930
Ops *(groupGeneric)*, 196
Ops,array,array-method *(StructureClasses)*, 956
Ops,array,structure-method *(StructureClasses)*, 956
Ops,structure,array-method *(StructureClasses)*, 956
Ops,structure,structure-method *(StructureClasses)*, 956
Ops,structure,vector-method *(StructureClasses)*, 956
Ops,vector,structure-method *(StructureClasses)*, 956
Ops.Date, *95*, 296
Ops.difftime *(difftime)*, 110
Ops.factor *(factor)*, 150
Ops.numeric_version *(numeric_version)*, 293
Ops.ordered *(factor)*, 150
Ops.POSIXt *(DateTimeClasses)*, 95
Ops.ts *(ts)*, 1370
optim, *991, 992*, *996*, *1029, 1030*, *1056*, *1073*, *1116*, *1196*, *1198*, 1214, *1344, 1345*, *1517, 1518*
optimise *(optimize)*, 1219
optimize, *1196*, *1198*, *1216, 1217*, 1219, *1381*
options, *4*, *30*, *42*, *49*, *91*, *97*, *124*, *132, 133*, *143*, *166*, *232*, *244*, 297, *308*, *316*, *319*, *351*, *379*, *402*, *411*, *414*, *417*, *419*, *447*, *477*, *490, 491*, *584*, *591*, *594, 595*, *684*, *703*, *714*, *725*, *731*, *833*, *1033*, *1050*, *1102*, *1106*, *1143*, *1179*, *1189, 1190*, *1200*, *1253*, *1268*, *1271, 1272*, *1434*, *1437*, *1450*, *1452, 1453*, *1472*, *1502*, *1504*, *1516*, *1536, 1537*, *1556, 1557*
Orange, 541
OrchardSprays, 542
order, *217*, 303, *340*, *398–400*, *499*, *1128*
order.dendrogram, *1050*, *1113*, 1221
ordered, *196*, *316*
ordered *(factor)*, 150
ordered-class *(setOldClass)*, 946
outer, *223*, *254*, *261*, 305
over *(plotmath)*, 619

p.adjust, 1222, *1224–1227*
pacf *(acf)*, 962
package.dependencies, 1413
package.skeleton, *899*, *913*, 1491, *1499*
package_version, *196*, *1443*
package_version *(numeric_version)*, 293
packageDescription, *1439*, 1492
packageEvent *(UserHooks)*, 486
packageInfo-class *(setOldClass)*, 946
packageIQR-class *(setOldClass)*, 946
packageSlot, *886*, *935*
packageSlot *(getPackageName)*, 898
packageSlot<- *(getPackageName)*, 898
packageStartupMessage *(message)*, 272
packageStatus, *1443*, 1493
packBits *(rawConversion)*, 344
packGrob *(grid.pack)*, 827
page, *156*, 1495
PAGER *(EnvVar)*, 132
pairlist, *1*, *82*, *165*, *290*
pairlist *(list)*, 237
pairs, *664*, *688*, 722, *724*, *739*
pairwise.prop.test, 1224
pairwise.t.test, *1223*, 1224, *1226*
pairwise.table, 1225
pairwise.wilcox.test, 1226

palette, *580*, *583*, *600*, 611, *613*, *693*, *731*, *749*, *791*, *1010*
Palettes, 612
panel.identify, *822*
panel.smooth, *688*, 724, *1232*, *1364*
par, *376*, *588*, *599*, *603*, *605*, *609*, *610*, *622*, *628*, *629*, *632*, *641*, *649*, *651*, *660*, *661*, *665–667*, *669*, *672*, *673*, *678*, *682*, *688*, *691*, *696*, *702*, *705*, *707*, *713–716*, *718*, *721*, *724*, 725, *734*, *737*, *738*, *740–742*, *744*, *745*, *748*, *750*, *751*, *754*, *756*, *759*, *760*, *764*, *765*, *768*, *769*, *771*, *775–777*, *779*, *780*, *1112*, *1113*, *1125*, *1141*, *1228*, *1230*, *1232*, *1234*, *1239*, *1343*, *1364*, *1457*
Paren, *83*, 306, *439*
parent.env *(environment)*, 130
parent.env<- *(environment)*, 130
parent.frame, *131*, *134*, *135*
parent.frame *(sys.parent)*, 442
parse, *78*, *102*, *103*, *129*, *139*, 307, *400*, *402*
paste, *50*, *54*, *129*, *167*, *282*, 309, *409*, *424*, *432*, *1439*
path.expand, *35*, *153*, *154*, *156*, *158*, *159*, *230*, 310, *441*, *1459*
pathListing *(grid.ls)*, 822
pbeta, *1009*, *1081*, *1193*, *1362*
pbeta *(Beta)*, 1005
pbinom *(Binomial)*, 1008
pbirthday *(birthday)*, 1013
pcauchy *(Cauchy)*, 1018
pch *(points)*, 750
pchisq, *1081*, *1376*
pchisq *(Chisquare)*, 1022
pdf, *133*, *576*, *577*, *589–595*, 613, *617*, *622*, *632*, *634*, *641*, *645*, *646*, *686*, *750*, *751*, *776*, *1530*
pdf.options, *578*, *595*, *614*, *616*, 617, *635*
pdfFonts, *615*, *616*, *646*
pdfFonts *(postscriptFonts)*, 632
periodicSpline, *1323*
person, 1495
personList, *1440*
personList *(person)*, 1495
persp, *487*, *645*, 732
pexp *(Exponential)*, 1070
pf *(FDist)*, 1080
pgamma, *404*, *1193*
pgamma *(GammaDist)*, 1097
pgeom *(Geometric)*, 1100
phantom *(plotmath)*, 619
phyper *(Hypergeometric)*, 1117
pi *(Constants)*, 80
pico *(edit)*, 1454
pictex, *594*, 618
pie, 736
pipe *(connections)*, 74
pkgDepends, *1412*, *1422*
pkgDepends *(getDepList)*, 1410
PkgUtils, 1496
pkgVignettes *(buildVignettes)*, 1399
placeGrob *(grid.place)*, 829
plain *(plotmath)*, 619
PlantGrowth, 543
plclust *(hclust)*, 1108
plnorm *(Lognormal)*, 1159
plogis, *201*
plogis *(Logistic)*, 1155
plot, *653*, *672*, *697*, *699*, *705*, *710*, *713*, *715*, *716*, *721*, 737, *739–742*, *744*, *747*, *748*, *750*, *752*, *771*, *1064*, *1110*, *1132*, *1177*, *1230*, *1237–1239*, *1268*
plot.acf, *964*, 1227
plot.data.frame, *92*, 739
plot.Date, *95*
plot.Date *(axis.POSIXct)*, 667
plot.decomposed.ts *(decompose)*, 1044
plot.default, *654*, *664*, *668*, *671*, *677*, *684*, *689*, *692*, *696*, *713*, *715*, *726*, *727*, *730*, *731*, *738*, *739*, 740, *742*, *744*, *745*, *747–750*, *765*, *768*, *771*, *1049*, *1109*, *1132*, *1141*, *1239*
plot.dendrogram, *682*
plot.dendrogram *(dendrogram)*, 1048
plot.density, *1053*, 1228
plot.design, 742
plot.ecdf *(ecdf)*, 1063
plot.factor, 744, *745*, *747*
plot.formula, *738*, 744, *744*
plot.function *(curve)*, 689
plot.hclust, *1048*
plot.hclust *(hclust)*, 1108
plot.histogram, *697*, *698*, 746
plot.HoltWinters, 1229
plot.isoreg, *1129*, 1230
plot.lm, *297*, 1231, *1364*
plot.mlm *(plot.lm)*, 1231
plot.new, *487*, *594*, *682*, *729*, *748*, *769*
plot.new *(frame)*, 695
plot.POSIXct *(axis.POSIXct)*, 667
plot.POSIXlt *(axis.POSIXct)*, 667

plot.ppr, 1234, *1252*
plot.prcomp *(prcomp)*, 1253
plot.princomp *(princomp)*, 1267
plot.profile.nls, 1235, *1274*
plot.spec, *301*, 1236, *1316*, *1318*, *1321*
plot.stepfun, *1064*, 1237, *1340*
plot.stl, *1342, 1343*
plot.stl *(stlmethods)*, 1343
plot.table, 747
plot.ts, *1141, 1142*, 1239, *1344*, *1371–1373*
plot.tskernel *(kernel)*, 1132
plot.TukeyHSD *(TukeyHSD)*, 1377
plot.window, *666*, *672*, *678*, *684*, *691, 692*, *696*, *705*, *726*, *730*, *738*, *740, 741*, 748, *768*, *773*
plot.xy, *713*, *715*, 749, *749*, *752*
plotmath, *149*, *239*, *262*, *278*, *309*, *323*, *388*, *433*, *574*, *603*, *618*, 619, *646*, *675*, *709*, *721*, *776–778*, *842*, *1122*
plotViewport, *787*, 851
pmatch, *53*, *55*, *143*, *195*, *258–260*, 310
pmax *(Extremes)*, 148
pmin *(Extremes)*, 148
pnbinom *(NegBinomial)*, 1192
png, *47, 48*, *301*, *591*, *593*, *595*, 623, *638*, *648*
pnorm, *1377*
pnorm *(Normal)*, 1209
points, *622*, *654*, *688*, *697*, *703*, *709*, *713*, *715, 716*, *724*, *726, 727*, *729*, *734*, *738*, *741*, *745*, *749*, 750, *750*, *831*, *1049, 1050*, *1230*, *1232*, *1364*
points.default, *749*
points.formula *(plot.formula)*, 744
pointsGrob *(grid.points)*, 830
Poisson, 1240
poisson *(family)*, 1076
poly, *1168*, 1242
polygon, *579*, *726*, *736*, 753, *756*, *760*, *780*, *1049*
polygonGrob *(grid.polygon)*, 831
polylineGrob *(grid.lines)*, 819
polym *(poly)*, 1242
polyroot, 312, *1381*
pop.viewport, 851, *852*
popViewport, *796*, *860*
popViewport *(Working with Viewports)*, 861
pos.to.env, 313
Position *(funprog)*, 177
POSIXct, *21*, *168*, *421*
POSIXct *(DateTimeClasses)*, 95
POSIXct-class *(setOldClass)*, 946
POSIXlt, *21*, *168*, *421*
POSIXlt *(DateTimeClasses)*, 95
POSIXlt-class *(setOldClass)*, 946
POSIXt, *110*, *196*, *291*
POSIXt *(DateTimeClasses)*, 95
POSIXt-class *(setOldClass)*, 946
possibleExtends, *867*
postDrawDetails *(drawDetails)*, 787
postscript, *133*, *299*, *576, 577*, *589–594*, *614–616*, *619*, *622*, *624*, 627, *632–637*, *645, 646*, *648*, *652, 653*, *686*, *731*, *750*, *776*
postscriptFonts, *596*, *628, 629*, *631*, 632, *646*
power, *1077, 1078*, *1167*, 1243
power.anova.test, 1244
power.prop.test, 1245, *1270*
power.t.test, *1244*, 1247, *1270*
PP.test, 1248
ppoints, 1249, *1280*
ppois *(Poisson)*, 1240
ppr, *1234*, 1250, *1357*
prcomp, *1011*, *1192*, 1253, *1268, 1269*, *1299*
precip, 543
predict, *138*, *1069*, *1146*, *1192*, *1201*, 1255, *1262*, *1265*
predict.ar, *1256*
predict.ar *(ar)*, 985
predict.Arima, *993*, 1256, *1256*
predict.arima0, *1256*
predict.arima0 *(arima0)*, 995
predict.glm, *1105*, *1256*, 1257, *1364*
predict.HoltWinters, *1116*, *1230*, *1256*, 1259
predict.lm, *1145, 1146*, *1256*, 1260
predict.loess, *1153*, *1256*, 1262
predict.mlm *(predict.lm)*, 1260
predict.nls, *1202*, *1256*, 1264
predict.poly, *1256*
predict.poly *(poly)*, 1242
predict.prcomp *(prcomp)*, 1253
predict.princomp, *1256*
predict.princomp *(princomp)*, 1267
predict.smooth.spline, *1256*, 1265, *1312*
predict.StructTS, *1256*
predict.StructTS *(StructTS)*, 1344
preDrawDetails *(drawDetails)*, 787
preplot, 1267
presidents, 544
pressure, 545

pretty, 313, *597*, *667*, *669*
prettyNum, *166, 167*
prettyNum *(formatC)*, 172
Primitive, 315
primitive, *485*
primitive *(Primitive)*, 315
princomp, *1011, 1012*, *1075*, *1151*, *1192*, *1255*, 1267, *1299*, *1355*
print, *48*, *50*, *66*, *167*, *283*, *298*, 315, *318–320*, *413*, *950*, *973*, *982*, *1050*, *1064*, *1110*, *1132*, *1177*, *1201*, *1268*, *1271, 1272*, *1340*, *1365*, *1495*, *1529*, *1537*
print.anova, *1271*
print.anova *(anova)*, 973
print.aov *(aov)*, 982
print.aovlist *(aov)*, 982
print.ar *(ar)*, 985
print.arima0 *(arima0)*, 995
print.AsIs *(AsIs)*, 25
print.Bibtex *(toLatex)*, 1543
print.browseVignettes *(browseVignettes)*, 1433
print.by *(by)*, 43
print.checkDocFiles *(QC)*, 1414
print.checkDocStyle *(QC)*, 1414
print.checkFF *(checkFF)*, 1401
print.checkReplaceFuns *(QC)*, 1414
print.checkS3methods *(QC)*, 1414
print.checkTnF *(checkTnF)*, 1403
print.checkVignettes *(checkVignettes)*, 1404
print.codoc *(codoc)*, 1405
print.codocClasses *(codoc)*, 1405
print.codocData *(codoc)*, 1405
print.condition *(conditions)*, 70
print.connection *(connections)*, 74
print.data.frame, *92*, 317
print.Date *(Dates)*, 94
print.default, *49*, *66*, *128*, *166*, *298*, *316, 317*, 318, *320*, *330*, *413*, *666*, *1049*
print.dendrogram *(dendrogram)*, 1048
print.density *(density)*, 1051
print.difftime *(difftime)*, 110
print.dist *(dist)*, 1059
print.DLLInfo *(getLoadedDLLs)*, 185
print.DLLInfoList *(getLoadedDLLs)*, 185
print.DLLRegisteredRoutines *(getDLLRegisteredRoutines)*, 183
print.ecdf *(ecdf)*, 1063
print.factanal *(loadings)*, 1151
print.family *(family)*, 1076
print.formula *(formula)*, 1090
print.ftable *(ftable)*, 1094
print.getAnywhere *(getAnywhere)*, 1462
print.glm *(glm)*, 1102
print.hclust *(hclust)*, 1108
print.hexmode *(hexmode)*, 200
print.HoltWinters *(HoltWinters)*, 1114
print.hsearch *(help.search)*, 1473
print.infl *(influence.measures)*, 1119
print.integrate *(integrate)*, 1122
print.kmeans *(kmeans)*, 1133
print.Latex *(toLatex)*, 1543
print.libraryIQR *(library)*, 231
print.lm *(lm)*, 1143
print.loadings, *1075*
print.loadings *(loadings)*, 1151
print.logLik *(logLik)*, 1156
print.ls_str *(ls.str)*, 1481
print.MethodsFunction *(methods)*, 1486
print.NativeRoutineList *(getDLLRegisteredRoutines)*, 183
print.noquote *(noquote)*, 283
print.numeric_version *(numeric_version)*, 293
print.octmode *(octmode)*, 294
print.packageDescription *(packageDescription)*, 1492
print.packageInfo *(library)*, 231
print.packageIQR *(data)*, 1445
print.packageStatus *(packageStatus)*, 1493
print.POSIXct *(DateTimeClasses)*, 95
print.POSIXlt *(DateTimeClasses)*, 95
print.power.htest, 1270
print.prcomp *(prcomp)*, 1253
print.princomp *(princomp)*, 1267
print.proc_time *(proc.time)*, 321
print.recordedplot *(recordPlot)*, 640
print.restart *(conditions)*, 70
print.rle *(rle)*, 364

print.sessionInfo (sessionInfo), 1533
print.simple.list (print), 315
print.socket (make.socket), 1483
print.srcfile (srcfile), 412
print.srcref (srcfile), 412
print.stepfun (stepfun), 1339
print.StructTS (StructTS), 1344
print.summary.aov (summary.aov), 1346
print.summary.aovlist (summary.aov), 1346
print.summary.glm, *1271*
print.summary.glm (summary.glm), 1348
print.summary.lm, *171*, *1271*, *1272*
print.summary.lm (summary.lm), 1350
print.summary.manova (summary.manova), 1352
print.summary.nls (summary.nls), 1353
print.summary.prcomp (prcomp), 1253
print.summary.princomp (summary.princomp), 1355
print.summary.table (table), 454
print.table (print), 315
print.terms (terms), 1365
print.ts, 1270, *1371*
print.TukeyHSD (TukeyHSD), 1377
print.undoc (undoc), 1421
print.vignette (vignette), 1553
print.warnings (warnings), 491
print.xtabs (xtabs), 1396
printCoefmat, *301*, 1271
prmatrix, 320
proc.time, *181*, 321, *389*, *448*, *452*
prod, 322
profile, *1201*, *1235*, 1273, *1274*
profile.glm, *1273*
profile.nls, *1202*, *1235*, 1273, *1273*
prohibitGeneric, *941*
prohibitGeneric (implicitGeneric), 900
proj, *983*, *1183*, 1274
promax (varimax), 1385
promise, *134*
promise (delayedAssign), 101
promises, *161*, *374*
promises (delayedAssign), 101
prompt, *924*, *925*, *1406*, *1471*, *1491*, *1492*, 1497, *1500*, *1501*
promptClass, 923, *925*, *1406*, *1491*, *1492*
promptData, *1498*, 1499, *1499*
promptMethods, 924, *924*, *1491*, *1492*
promptPackage, 1500
prop.table, *256*, 323, *455*
prop.test, *1008*, *1224*, *1246*, 1276, *1278*, *1360*
prop.trend.test, 1278
prototype, *911*, *933*
prototype (representation), 925
ps.options, *578*, *595*, *617*, *628*, *631*, 635, *653*
psigamma (Special), 402
psignrank, *1391*
psignrank (SignRank), 1306
pt (TDist), 1361
ptukey (Tukey), 1376
punif (Uniform), 1379
Puromycin, 545
push.viewport, 852, *852*
pushBack, *76*, *78*, 324, *465*
pushBackLength (pushBack), 324
pushViewport, *796*, *860*
pushViewport (Working with Viewports), 861
pweibull (Weibull), 1386
pwilcox, *1391*
pwilcox (Wilcoxon), 1392

q, *374*, *417*, *1430*
q (quit), 328
qbeta, *1081*
qbeta (Beta), 1005
qbinom (Binomial), 1008
qbirthday (birthday), 1013
QC, *1406*, 1414, *1421*
qcauchy (Cauchy), 1018
qchisq, *1081*
qchisq (Chisquare), 1022
qexp (Exponential), 1070
qf (FDist), 1080
qgamma (GammaDist), 1097
qgeom (Geometric), 1100
qhyper (Hypergeometric), 1117
qlnorm (Lognormal), 1159
qlogis (Logistic), 1155
qnbinom (NegBinomial), 1192
qnorm, *334*, *1377*
qnorm (Normal), 1209
qpois (Poisson), 1240
qqline, *1232*

qqline (qqnorm), 1279
qqnorm, 1249, 1279, 1305
qqplot, 1249
qqplot (qqnorm), 1279
qr, 34, 59, 127, 221, 222, 325, 327, 328, 436, 978, 1017, 1147, 1352
QR.Auxiliaries, 327
qr.Q, 326
qr.Q (QR.Auxiliaries), 327
qr.qy, 328
qr.R, 326
qr.R (QR.Auxiliaries), 327
qr.solve, 397
qr.X, 326
qr.X (QR.Auxiliaries), 327
qsignrank (SignRank), 1306
qt (TDist), 1361
qtukey, 1378
qtukey (Tukey), 1376
quade.test, 1093, 1280
quakes, 547
quantile, 87, 574, 1088, 1127, 1176, 1249, 1282
quarters (weekdays), 492
quartz, 48, 415, 595, 616, 625, 626, 636, 638, 703, 714
quartz.options, 595
quartzFont (quartzFonts), 638
quartzFonts, 638, 638
quasi, 1103
quasi (family), 1076
quasibinomial (family), 1076
quasipoisson (family), 1076
Querying the Viewport Tree, 853
quit, 328
qunif (Uniform), 1379
quote, 41, 114, 135, 470, 471, 622, 909
quote (substitute), 429
Quotes, 81, 292, 319, 330, 439, 441
quotes, 363
qweibull (Weibull), 1386
qwilcox (Wilcoxon), 1392

R.home, 133, 451
R.home (Rhome), 364
R.Version, 133, 331
R.version, 4, 215, 293, 442, 1533
R.version (R.Version), 331
r2dtable, 1284
R_BATCH (EnvVar), 132
R_BROWSER (EnvVar), 132
R_COMPLETION (EnvVar), 132
R_DEFAULT_PACKAGES (Startup), 414
R_DOC_DIR (EnvVar), 132
R_DVIPSCMD (EnvVar), 132
R_ENVIRON (Startup), 414
R_ENVIRON_USER (Startup), 414
R_GSCMD (EnvVar), 132
R_HISTFILE (EnvVar), 132
R_HISTSIZE (EnvVar), 132
R_HOME, 5, 83, 237, 298, 414, 415, 584, 646, 1400, 1420, 1425, 1432, 1437, 1475, 1525, 1534
R_HOME (Rhome), 364
R_INCLUDE_DIR (EnvVar), 132
R_LATEXCMD (EnvVar), 132
R_LIBS (libPaths), 230
R_LIBS_SITE (libPaths), 230
R_LIBS_USER (libPaths), 230
R_MAKEINDEXCMD (EnvVar), 132
R_PAPERSIZE (EnvVar), 132
R_PDFLATEXCMD (EnvVar), 132
R_PDFVIEWER (EnvVar), 132
R_PLATFORM (EnvVar), 132
R_PRINTCMD (EnvVar), 132
R_PROFILE (Startup), 414
R_PROFILE_USER (Startup), 414
R_RD4DVI (EnvVar), 132
R_RD4PDF (EnvVar), 132
R_SHARE_DIR (EnvVar), 132
R_system_version (numeric_version), 293
R_TEXI2DVICMD (EnvVar), 132
R_UNZIPCMD (EnvVar), 132
R_ZIPCMD (EnvVar), 132
rainbow, 583, 599, 600, 606, 612, 641, 704, 705, 731
rainbow (Palettes), 612
Random, 332
Random.user, 334, 336
randu, 547
range, 149, 338, 597, 688, 1088, 1127
rank, 304, 339, 400, 499
rapply, 226, 340, 1047
raw, 341
raw-class (BasicClasses), 870
rawConnection, 343
rawConnectionValue (rawConnection), 343
rawConversion, 344
rawShift, 342
rawShift (rawConversion), 344
rawToBits (rawConversion), 344
rawToChar, 342
rawToChar (rawConversion), 344

rbeta (Beta), 1005
rbind, *210*, *876*
rbind (cbind), 50
rbind2 (cbind2), 875
rbind2,ANY,ANY-method (cbind2), 875
rbind2,ANY,missing-method (cbind2), 875
rbind2-methods (cbind2), 875
rbinom, *1188*
rbinom (Binomial), 1008
rc.getOption (rcompgen), 1501
rc.options (rcompgen), 1501
rc.settings (rcompgen), 1501
rc.status (rcompgen), 1501
rcauchy (Cauchy), 1018
rchisq (Chisquare), 1022
rcompgen, 1501
rcond (kappa), 221
Rd2dvi (RdUtils), 345
Rd2txt (RdUtils), 345
Rd_db (Rdutils), 1416
Rd_parse (Rdutils), 1416
Rdconv, *924*
Rdconv (RdUtils), 345
Rdindex, 1415
RdUtils, 345
Rdutils, 1416
Re (complex), 69
read.00Index, 1417
read.csv, *1509*
read.csv (read.table), 1511
read.csv2 (read.table), 1511
read.dcf, *78*, *1424*, *1493*, *1550*
read.dcf (dcf), 98
read.delim (read.table), 1511
read.delim2 (read.table), 1511
read.DIF, 1506
read.fortran, 1508
read.ftable, *1096*, 1285
read.fwf, *1508*, 1509, *1509*, *1515*
read.socket, *1441*, *1484*, 1510
read.table, *92*, *129*, *299*, *330*, *380*, *1418*, *1445*, *1446*, *1508–1510*, 1511, *1545*, *1546*, *1557*
readBin, *4*, *78*, 346, *349*, *350*, *352*, *380*
readChar, *78*, *347*, 349, *380*
readCitationFile (citEntry), 1439
readline, 350, *1499*
readLines, *77*, *78*, *129*, *324*, *347*, *350*, 351, *351*, *379*, *380*, *498*
readNEWS, 1418
real, *116*, *291*, 352
Recall, *46*, 353
recordedplot-class (setOldClass), 946
recordGraphics, 639, *834*
recordGrob (grid.record), 834
recordPlot, 640
recover, *100*, *298*, *469–471*, *1450*, 1515
rect, *674*, *726*, *746*, *754*, 755, *762*
rect.hclust, *1110*, *1111*, *1119*, 1287
rectGrob (grid.rect), 835
Reduce (funprog), 177
reformulate (delete.response), 1045
reg.finalizer, *180*, 354
regex, 355
regexp, *195*, *1465*, *1466*
regexp (regex), 355
regexpr, *55*, *1407*
regexpr (grep), 193
registerImplicitGenerics (implicitGeneric), 900
regular expression, *193*, *195*, *240*, *250*, *423*, *424*, *627*, *1428*, *1431*, *1473*, *1474*, *1481*
regular expression (regex), 355
relevel, *229*, 1288, *1290*
relist, *482*, *483*, *1294*, 1517
REMOVE, *234*, *1478*, *1479*, 1519, *1520*, *1550*
remove, 359
remove.packages, *1519*, 1520, *1550*
removeCConverter (getNumCConverters), 188
removeClass (findClass), 886
removeGeneric (GenericFunctions), 891
removeGrob, *793*, *798*, *812*, *814*, *837*
removeGrob (grid.remove), 836
removeMethod (setMethod), 942
removeMethods (GenericFunctions), 891
removeTaskCallback, *461*, *462*
removeTaskCallback (taskCallback), 458
Renviron (Startup), 414
reorder, *229*, *1113*, *1221*, *1288*, *1289*
reorder (reorder.factor), 1289
reorder.dendrogram, *1050*, *1112*, 1288, *1290*
reorder.factor, 1289
rep, *145*, *210*, *306*, *315*, 360, *384*, *387*, *653*, *656*, *858*, *859*

rep.numeric_version (numeric_version), 293
repeat, 362
repeat (Control), 82
repeat-class (language-class), 909
replace, 362
replayPlot (recordPlot), 640
replicate, 361
replicate (lapply), 225
replications, 983, 1183, 1291
representation, 925, 933
require, 218, 298, 414, 1502, 1504
require (library), 231
Reserved, 331, 362, 439
reserved, 82, 211, 248, 251, 276, 290, 1468, 1469
reserved (Reserved), 362
resetClass (findClass), 886
resetGeneric, 916
reshape, 1292, 1536
resid (residuals), 1295
residuals, 974, 1027, 1067, 1087, 1105, 1108, 1146, 1150, 1192, 1201, 1295, 1307, 1364, 1389
residuals.glm, 1150, 1233, 1349
residuals.glm (glm.summaries), 1107
residuals.HoltWinters (HoltWinters), 1114
residuals.lm (lm.summaries), 1149
residuals.tukeyline (line), 1142
restartDescription (conditions), 70
restartFormals (conditions), 70
retracemem (tracemem), 473
return, 211, 307
return (function), 176
rev, 363, 1112
rev.dendrogram, 1289
rev.dendrogram (dendrogram), 1048
rexp (Exponential), 1070
rf (FDist), 1080
rgamma (GammaDist), 1097
rgb, 580, 583, 599, 601, 602, 606, 613, 641, 642, 731, 791
rgb2hsv, 606, 642
rgeom (Geometric), 1100
RHOME, 364, 1520
Rhome, 364
rhyper (Hypergeometric), 1117
ring (plotmath), 619
rivers, 548
rle, 364, 480
rle-class (setOldClass), 946
rlnorm (Lognormal), 1159
rlogis (Logistic), 1155
rm (remove), 359
rmultinom (Multinomial), 1187
rnbinom (NegBinomial), 1192
RNG, 1210
RNG (Random), 332
RNGkind, 336, 1307
RNGkind (Random), 332
RNGversion (Random), 332
rnorm, 335, 1380
rnorm (Normal), 1209
rock, 549
roman, 1521
Round, 365
round, 111, 208, 367
round (Round), 365
round.Date, 95
round.Date (round.POSIXt), 367
round.difftime (difftime), 110
round.POSIXt, 98, 367
row, 62, 368, 384, 395
row.names, 31, 32, 92, 113, 369, 370
row.names<- (row.names), 369
row/colnames, 370
rowMeans (colSums), 64
rownames, 93, 114, 264, 369, 370, 516
rownames (row/colnames), 370
rownames<- (row/colnames), 370
rowsum, 64, 371
rowSums, 372
rowSums (colSums), 64
rpart, 962
rpois (Poisson), 1240
Rprof, 415, 1273, 1521, 1523, 1539, 1540
Rprofile (Startup), 414
Rprofmem, 270, 474, 1522, 1523, 1540
Rscript, 1524
RShowDoc, 1478, 1525
rsignrank (SignRank), 1306
RSiteSearch, 1475, 1476, 1526
rstandard, 1108, 1151, 1295
rstandard (influence.measures), 1119
rstudent, 1108, 1150, 1151, 1295
rstudent (influence.measures), 1119
rt (TDist), 1361
Rtangle, 1527, 1530, 1542, 1543
RtangleSetup (Rtangle), 1527

rug, *220*, *664*, 757, *1363*
runif, *335*, *1210*
runif *(Uniform)*, 1379
runmed, 1295, *1314*
RweaveLatex, 1528, *1528*, *1542*, *1543*
RweaveLatexSetup *(RweaveLatex)*, 1528
rweibull *(Weibull)*, 1386
rwilcox *(Wilcoxon)*, 1392

S3 *(S3Part)*, 927
S3-class *(S3Part)*, 927
S3Class, *879*
S3Class *(S3Part)*, 927
S3Class<- *(S3Part)*, 927
S3groupGeneric, *931*, *932*
S3groupGeneric *(groupGeneric)*, 196
S3Methods, *1487*
S3Methods *(UseMethod)*, 484
S3Part, 927
S3Part<- *(S3Part)*, 927
S4 *(S3Part)*, 927
S4-class *(BasicClasses)*, 870
S4groupGeneric, *198*, 930, *957*
SafePrediction, *1258*, *1262*
SafePrediction *(makepredictcall)*, 1168
sammon, *1026*
sample, 372
sapply, *254*, *458*
sapply *(lapply)*, 225
save, *29*, *78*, *120*, *241*, *244*, *299*, 373, *497*, *1447*, *1449*
savehistory, *133*, 1530
savePlot, *593*, 644, *650*
scale, 376, *437*, *1168*, *1254*
scan, *78*, *129*, *152*, *308*, *324*, *330*, *352*, 377, *402*, *497*, *1445*, *1508*, *1510*, *1512–1515*
scatter.smooth, 1297
SClassExtension, *878*, *880*, *881*, *905*, *915*
SClassExtension-class, 932
screen, 758
screeplot, *1255*, *1268*, *1269*, 1299
scriptscriptstyle *(plotmath)*, 619
scriptstyle *(plotmath)*, 619
sd, *1038*, 1300, *1362*
Sd2Rd *(RdUtils)*, 345
se.contrast, *1183*, 1300
se.contrast.aovlist, *1066*
sealClass *(findClass)*, 886
SealedMethodDefinition-class *(MethodDefinition-class)*, 913
search, *22*, *26*, *29*, *73*, *106*, *107*, *137*, *182*, *232*, *234*, *250*, *359*, 380, *577*, *899*, *1429*, *1481*, *1503*
searchpaths *(search)*, 380
Seatbelts *(UKDriverDeaths)*, 561
seek, *78*, 381
seekViewport, *796*, *860*
seekViewport *(Working with Viewports)*, 861
segments, *660*, *662*, *726*, *754*, *756*, 760, *1049*, *1050*, *1238*
segmentsGrob *(grid.segments)*, 837
select.list, *1485*, 1532, *1547*
selectMethod, *873*, *875*, *882*, *916*, *952*, *953*
selectMethod *(getMethod)*, 896
selfStart, *1101*, *1207*, *1208*, 1302, *1315*, *1325–1327*, *1330–1333*, *1335*
selfStart.default, *1101*
selfStart.formula, *1101*
seq, *63*, *361*, *363*, 382, *384*, *385*, *387*, *1442*
seq.Date, *89*, *95*, 384, *384*
seq.int, *210*
seq.POSIXt, *89*, *98*, *384*, 385, *385*, *701*
seq_along *(seq)*, 382
seq_len, *386*
seq_len *(seq)*, 382
sequence, *361*, *384*, 386
sessionInfo, *332*, *442*, *1436*, *1473*, 1533, *1544*
set.seed *(Random)*, 332
setAs, *867*, *875*, *905–907*
setAs *(as)*, 866
setCConverterStatus *(getNumCConverters)*, 188
setChildren *(grid.add)*, 797
setClass, *865*, *868*, *875*, *878–881*, *894–896*, *903–906*, *909*, *911*, *912*, *915*, *916*, *918*, *925*, *926*, *928*, *932*, 933, *939*, *947–950*, *959*, *960*, *1478*
setClassUnion, *878*, *882*, *883*, *887*, *905*, *915*, *916*, *932*, 936
setdiff *(sets)*, 387
setEPS, *628*
setEPS *(ps.options)*, 635
setequal *(sets)*, 387
setGeneric, *890*, *894*, *901*, *907*, *918*, *921*, 937, *943*, *944*, *950*, *956*

setGenericImplicit (implicitGeneric), 900
setGrob, 793
setGrob (grid.set), 839
setGroupGeneric, 890, 930
setGroupGeneric (setGeneric), 937
setHook, 233, 288, 696, 734, 826
setHook (UserHooks), 486
setIs, 868, 878, 880, 895, 905, 915, 916, 921, 932, 948, 950
setIs (is), 903
setMethod, 471, 865, 866, 883, 890, 903, 908, 909, 912–914, 918, 931, 938, 941, 942, 947, 949, 953, 1478
setNames, 1024, 1304
setOldClass, 879, 921, 927–929, 944, 946, 956, 957
setPackageName (getPackageName), 898
setPS (ps.options), 635
setReplaceMethod (GenericFunctions), 891
setRepositories, 301, 302, 1437, 1533
sets, 387
setSessionTimeLimit (setTimeLimit), 388
setTimeLimit, 388
setTxtProgressBar (txtProgressBar), 1544
setValidity (validObject), 958
setwd, 446
setwd (getwd), 191
shapiro.test, 1138, 1305
SHLIB, 125, 237, 1444, 1534
show, 298, 319, 939, 950
show,ANY-method (show), 950
show,classRepresentation-method (show), 950
show,genericFunction-method (show), 950
show,MethodDefinition-method (show), 950
show,MethodWithNext-method (show), 950
show,ObjectsWithPackage-method (show), 950
show,signature-method (signature-class), 953
show,traceable-method (show), 950
show,ts-method (StructureClasses), 956
show-methods (show), 950
showClass, 951
showConnections, 78, 343, 389, 465
showDefault, 950
showMethods, 884, 889, 894, 918, 919, 951, 951, 1486, 1487
showMlist, 951
showNonASCII, 1419
shQuote, 331, 390, 411, 450
sign, 392
signalCondition, 417
signalCondition (conditions), 70
Signals, 392
signature (GenericFunctions), 891
signature-class, 953
signif, 166, 172, 318, 434, 655
signif (Round), 365
SignRank, 1306
simint, 1378
simpleCondition (conditions), 70
simpleError (conditions), 70
simpleMessage (conditions), 70
simpleWarning (conditions), 70
simulate, 1307
sin, 201, 262
sin (Trig), 476
single, 163
single (double), 115
single-class (BasicClasses), 870
sinh (Hyperbolic), 200
sink, 49, 78, 389, 390, 393, 1436
sleep, 549
slice.index, 395
slot, 396, 877, 880, 954
slot<- (slot), 954
slotNames (slot), 954
slotOp, 395
slotsFromS3 (S3Part), 927
smooth, 1297, 1308
smooth.spline, 1252, 1266, 1295, 1309, 1310, 1323
smoothEnds, 1296, 1297, 1313
socket-class (setOldClass), 946
socketConnection (connections), 74
socketSelect, 396
solve, 34, 60, 326, 397, 1166
solve.qr, 397
solve.qr (qr), 325
sort, 204, 217, 243, 304, 340, 363, 398, 499
sort.list (order), 303
sortedXyData, 1207, 1208, 1315
source, 119, 120, 209, 308, 400, 447, 867, 943, 1404, 1446, 1451, 1541

spec (spectrum), 1320
spec.ar, 1315, *1320, 1321*
spec.pgram, 1317, *1319–1321*
spec.taper, *1318*, 1319
Special, *19*, *262*, 402
spectrum, *1131*, *1237*, *1316*, *1318*, 1320
spineplot, *681*, *744*, 761
spline, *984*
spline (splinefun), 1321
splinefun, *690*, *984*, *1065*, *1238*, 1321, *1340*
splinefunH (splinefun), 1321
split, *87*, 405
split.Date (Dates), 94
split.POSIXct (DateTimeClasses), 95
split.screen, *707*, *729*, *731*
split.screen (screen), 758
split<- (split), 405
sprintf, *129*, *167*, *174*, *190*, *309*, 407, *615*, *628*, *652*
sqrt, *19*, *246*, *404*
sqrt (MathFun), 261
sQuote, *299*, *331*, *391*, 410, *430*
srcfile, *308*, 412
srcfile-class (srcfile), 412
srcfilecopy, *308*
srcfilecopy (srcfile), 412
srcfilecopy-class (srcfile), 412
srcref, *308*
srcref (srcfile), 412
srcref-class (srcfile), 412
SSasymp, *1206*, 1324, *1334, 1335*
SSasympOff, 1325
SSasympOrig, 1326
SSbiexp, 1327
SSD, *1173, 1174*, 1328
SSfol, *556*, 1329
SSfpl, 1330
SSgompertz, 1331
SSlogis, 1332
SSmicmen, 1333
SSweibull, 1334
stack, *1294*, 1535
stack.loss (stackloss), 550
stack.x (stackloss), 550
stackloss, 550, *1146*
standardGeneric, *890*, *893*
Stangle, *1404*, *1527*, *1554*
Stangle (Sweave), 1540
stars, 763, *774*
start, 1335, *1369*, *1371*, *1375*
Startup, *66*, *132, 133*, *298*, 414, *446*, *1432*
stat.anova, *974*, 1336
state, 551, *564*
stats (stats-package), 961
stats-deprecated, 1337
stats-package, 961
stderr, *272*, *393*
stderr (showConnections), 389
stdin, *324*, *352*, *377*, *389*, *401*, *1512*, *1524*
stdin (showConnections), 389
stdout, *76*, *393*
stdout (showConnections), 389
stem, *699*, *747*, 766
step, *967*, *1071, 1072*, 1337
stepAIC, *1338, 1339*
stepfun, *1064, 1065*, *1128*, *1237*, 1339
stl, *1045*, *1185*, 1341, *1343, 1344*, *1346*
stlmethods, 1343
stop, *191*, *273*, *298*, *393*, 416, *418*, *491*, *1424*
stopifnot, *10*, *417*, 418
storage.mode, *116, 117*, *208*, *291*, *479*
storage.mode (mode), 274
storage.mode<- (mode), 274
str, *251*, *1050*, *1431*, *1481, 1482*, 1536
str.default, *1049*
str.dendrogram (dendrogram), 1048
str.logLik (logLik), 1156
str.POSIXt (DateTimeClasses), 95
strftime, *97*, *168*, *298*
strftime (strptime), 419
strheight, *728*
strheight (strwidth), 769
stringHeight (stringWidth), 854
stringWidth, *850*, 854
stripchart, *664*, *676*, *739*, 767
strOptions (str), 1536
strptime, *23, 24*, *97*, *111*, *243, 244*, 419, *668*, *701*
strsplit, *54*, *129*, *282*, *309*, *355*, *358*, 423, *432*
strtrim, *129*, 425, *432*
StructTS, *1131*, *1185*, *1343*, 1344, *1374*, *1376*
structure, 426, *879*, *948*
structure-class (StructureClasses), 956
StructureClasses, 956
strwidth, *282*, *709*, *727*, 769
strwrap, 427, *1537*
sub, *54*, *56*, *129*, *424*, *1466*
sub (grep), 193
Subscript (Extract), 140

subset, *146*, 428, *475*
substitute, *41*, *101*, *103*, *114*, *274*, *295*, 429, *470, 471*, *622*, *915*
substr, *7*, *54*, *129*, *282*, *309*, *424, 425*, 431
substr<- *(substr)*, 431
substring *(substr)*, 431
substring<- *(substr)*, 431
sum, *64*, *323*, 432
Summary, *9*, *12, 13*, *148, 149*, *151*, *322*, *338*, *433*
Summary *(S4groupGeneric)*, 930
Summary *(groupGeneric)*, 196
summary, 433, *974*, *982*, *1053*, *1104, 1105*, *1149*, *1201*, *1347*, *1350, 1351*, *1354, 1355*, *1482*, *1536*, *1538*
summary.aov, *983*, 1346
summary.aovlist *(summary.aov)*, 1346
summary.connection *(connections)*, 74
Summary.Date *(Dates)*, 94
summary.Date *(Dates)*, 94
Summary.difftime *(difftime)*, 110
summary.ecdf *(ecdf)*, 1063
Summary.factor *(factor)*, 150
summary.glm, *434*, *975*, *1104, 1105*, *1108*, 1348
summary.infl *(influence.measures)*, 1119
summary.lm, *434*, *1146*, *1149, 1150*, *1163*, *1349*, 1350
summary.manova, *978, 979*, *1169*, 1352
summary.mlm *(summary.lm)*, 1350
summary.nls, *1202*, 1353
Summary.numeric_version *(numeric_version)*, 293
summary.packageStatus *(packageStatus)*, 1493
Summary.POSIXct *(DateTimeClasses)*, 95
summary.POSIXct *(DateTimeClasses)*, 95
Summary.POSIXlt *(DateTimeClasses)*, 95
summary.POSIXlt *(DateTimeClasses)*, 95
summary.prcomp *(prcomp)*, 1253
summary.princomp, *1269*, 1355
summary.stepfun *(stepfun)*, 1339
summary.table *(table)*, 454
summary.table-class *(setOldClass)*, 946
summaryRprof, *1522*, 1539
sunflowerplot, *655*, 770, *774*
sunspot.month, 552, *554*
sunspot.year, 553
sunspots, *552*, 553
sup *(plotmath)*, 619
suppressMessages *(message)*, 272
suppressPackageStartupMessages, *233*
suppressPackageStartupMessages *(message)*, 272
suppressWarnings *(warning)*, 490
supsmu, *1252*, *1309*, 1356
Surv, *499*
survival, *24*, *168*
survreg, *1364*
svd, *59*, *127*, *222*, *326*, 435, *1017*, *1255*, *1268*
svg, *48*
svg *(cairo)*, 576
Sweave, *302*, *1399, 1400*, *1404*, *1528*, *1530*, 1540, *1542*
SweaveSyntaxLatex *(Sweave)*, 1540
SweaveSyntaxNoweb *(Sweave)*, 1540
SweaveSyntConv, 1542
sweep, *16*, *261*, *376*, 436, *1037*
swiss, 554, *1146*
switch, *83*, 437
symbol *(plotmath)*, 619
symbols, *726*, *766*, 772
symnum, *1348*, *1350*, *1354*, 1357
Syntax, *19*, *68*, *83*, *143*, *247, 248*, *292*, *307*, *331*, 438
sys.call, *134*, *226*, *260*, *280*
sys.call *(sys.parent)*, 442
sys.calls *(sys.parent)*, 442
Sys.chmod *(files)*, 157
Sys.Date, *94, 95*
Sys.Date *(Sys.time)*, 448
sys.frame, *26*, *135*, *137*, *182*, *250*, *359*
sys.frame *(sys.parent)*, 442
sys.frames *(sys.parent)*, 442
sys.function *(sys.parent)*, 442
Sys.getenv, *133*, 439, *445, 446*
Sys.getlocale, *132*, *224*, *250*, *440*, *1481*
Sys.getlocale *(locales)*, 243
Sys.getpid *(getpid)*, 189
Sys.glob, *230*, *240*, 440, *481*
Sys.info, *4*, 441
Sys.localeconv, *243, 244*
Sys.localeconv *(localeconv)*, 242
sys.nframe *(sys.parent)*, 442
sys.on.exit, *296*

sys.on.exit (sys.parent), 442
sys.parent, 442, 1450
sys.parents (sys.parent), 442
Sys.putenv (base-deprecated), 34
Sys.setenv, 35, 133, 440, 445, 1453, 1531
Sys.setlocale, 242
Sys.setlocale (locales), 243
Sys.sleep, 389, 446
sys.source, 29, 131, 289, 299, 402, 447
sys.status (sys.parent), 442
Sys.time, 94, 97, 448, 467
Sys.timezone, 24, 133, 448
Sys.timezone (timezones), 466
Sys.umask, 76
Sys.umask (files), 157
Sys.unsetenv (Sys.setenv), 445
Sys.which, 449
system, 4, 5, 215, 449
system.file, 451
system.time, 322, 452, 1369

T (logical), 248
t, 14, 453, 1370
t.test, 1136, 1137, 1213, 1225, 1248, 1359, 1391
t.ts (ts), 1370
table, 87, 316, 454, 456, 457, 516, 540, 747, 968, 1095–1097, 1159, 1396
table-class (setOldClass), 946
tabulate, 87, 455, 456
tail (head), 1466
tan, 201
tan (Trig), 476
tanh, 1155
tanh (Hyperbolic), 200
tapply, 16, 43, 226, 372, 457, 970
taskCallback, 458
taskCallbackManager, 458, 459, 460, 462
taskCallbackNames, 462
tcrossprod (crossprod), 84
TDist, 1361
tempdir, 133, 1482
tempdir (tempfile), 463
tempfile, 463
termplot, 1233, 1363
terms, 742, 1046, 1091, 1104, 1145, 1179, 1180, 1182, 1365, 1366, 1367, 1383
terms.formula, 1091, 1365, 1366, 1367, 1383
terms.object, 1365, 1366, 1367
terrain.colors, 582, 612, 704, 705
terrain.colors (Palettes), 612
texi2dvi, 299, 1399, 1400, 1420
text, 139, 603, 605, 607, 619, 622, 629, 682–684, 703, 710, 721, 726–728, 730, 769, 770, 775, 778, 1239
textConnection, 78, 464, 1436
textConnectionValue (textConnection), 464
textGrob (grid.text), 842
textstyle (plotmath), 619
Theoph, 555
tiff, 47, 48, 595
tiff (png), 623
tilde, 466
time, 452, 654, 1335, 1368, 1371, 1375, 1395
time zone (timezones), 466
time zones, 23
time zones (timezones), 466
timestamp (savehistory), 1530
timezone (timezones), 466
timezones, 466
Titanic, 556
title, 622, 672, 678, 682, 684, 688, 691, 692, 698, 715, 721, 726, 729, 733, 738, 740–742, 768, 776, 777, 1109, 1230, 1232
tk_select.list, 1532
tkProgressBar, 1545
toBibtex (toLatex), 1543
toBibtex.citation (citation), 1438
toBibtex.citationList (citation), 1438
toBibtex.person (person), 1495
toBibtex.personList (person), 1495
toeplitz, 1369
toLatex, 1543
toLatex.sessionInfo (sessionInfo), 1533
tolower, 129, 195
tolower (chartr), 56
tools (tools-package), 1399
tools-deprecated, 1420
tools-package, 1399
ToothGrowth, 557
topenv, 299, 447
topenv (ns-topenv), 289
topo.colors, 581, 583, 704, 705
topo.colors (Palettes), 612
toString, 166, 167, 468
toupper, 129, 195
toupper (chartr), 56
trace, 100, 469, 474, 902, 957, 958
traceable, 878, 902

traceable-class (TraceClasses), 957
traceback, 42, 100, 298, 401, 417, 472
TraceClasses, 957
tracemem, 270, 473, 1522, 1523, 1540
tracingState, 474
tracingState (trace), 469
trans3d, 645, 734
transform, 429, 475, 496
treering, 558
trees, 559
trellis.focus, 822
Trig, 246, 476
trigamma (Special), 402
TRUE, 248, 362, 418, 1239
TRUE (logical), 248
truehist, 611, 699
trunc, 208, 931
trunc (Round), 365
trunc.Date (round.POSIXt), 367
trunc.POSIXt, 98
trunc.POSIXt (round.POSIXt), 367
truncate (seek), 381
try, 153, 233, 299, 417, 473, 477, 487
tryCatch, 473
tryCatch (conditions), 70
ts, 110, 196, 301, 1185, 1240, 1271, 1335, 1368, 1369, 1370, 1371, 1375, 1395
ts-class (StructureClasses), 956
ts-methods, 1371
ts.intersect, 1145
ts.intersect (ts.union), 1373
ts.plot, 1372
ts.union, 1373
tsdiag, 993, 998, 1374
tsp, 18, 31, 32, 247, 1335, 1368, 1369, 1371, 1375, 1395
tsp<- (tsp), 1375
tsSmooth, 1130, 1131, 1346, 1375
Tukey, 1376
TukeyHSD, 983, 1183, 1347, 1377
txtProgressBar, 1544
type, 116, 208, 291, 295
type (typeof), 478
type.convert, 1507, 1508, 1513, 1515, 1545
Type1Font, 629, 632, 634, 645
typeof, 18, 183, 213, 270, 275, 278, 292, 478, 877, 920
TZ (timezones), 466

UCBAdmissions, 560
ucv, 1003
UKDriverDeaths, 561
UKgas, 562
UKLungDeaths, 563
unclass, 152
unclass (class), 60
undebug (debug), 99
underline (plotmath), 619
undoc, 1406, 1421
Uniform, 1379
union (sets), 387
unique, 122, 479
unique.numeric_version (numeric_version), 293
unique.POSIXlt (DateTimeClasses), 95
uniroot, 312, 1003, 1196, 1220, 1245, 1246, 1248, 1380
unit, 786, 793, 796, 804, 805, 821, 822, 850, 854, 854, 857
unit.c, 856, 856
unit.length, 857
unit.pmax (unit.pmin), 858
unit.pmin, 858
unit.rep, 858
units, 778
units (difftime), 110
units<- (difftime), 110
unix.time (system.time), 452
unlink, 159, 463, 481
unlist, 44, 167, 210, 239, 482, 1294, 1517–1519
unlist.relistable (relist), 1517
unloadNamespace, 107, 233, 287
unloadNamespace (ns-load), 288
unlockBinding (bindenv), 38
unname, 265, 483
unsplit (split), 405
unstack (stack), 1535
untrace, 957
untrace (trace), 469
untracemem (tracemem), 473
unz, 1423
unz (connections), 74
update, 1381
update.formula, 1091, 1338, 1382, 1382
update.packages, 302, 1414, 1478, 1479, 1494, 1546
update.packageStatus (packageStatus), 1493
upgrade (packageStatus), 1493
upper.tri, 109
upper.tri (lower.tri), 249

upViewport, *796*, *860*
upViewport *(Working with Viewports)*, 861
url, *48*, *241*, *1453*, *1549*, *1550*
url *(connections)*, 74
url.show, *1453*, 1550
URLdecode, *77*
URLdecode *(URLencode)*, 1551
URLencode, *1432*, 1551
USAccDeaths, 563
USArrests, 564
UseMethod, *51*, *61*, *198*, 484
UserHooks, 486
USJudgeRatings, 564
USPersonalExpenditure, 565
uspop, 566
utf8Conversion, 487
utf8ToInt *(utf8Conversion)*, 487
utils *(utils-package)*, 1427
utils-deprecated, 1552
utils-package, 1427

VADeaths, 566
validDetails, 859
validObject, *880*, *912*, *934*, *948*, 958
var, *1042*, *1165*, *1166*, *1300*
var *(cor)*, 1036
var.test, *981*, *1005*, *1089*, *1187*, 1383
variable.names, *371*
variable.names *(case/variable.names)*, 1017
varimax, *1075*, 1385
vcov, *991*, *1028*, *1146*, *1201*, 1386
vector, *50*, *122*, *170*, *239*, *480*, 488
vector-class *(BasicClasses)*, 870
Vectorize, *306*, *1123*
Vectorize *(mapply)*, 254
version, *231*
version *(R.Version)*, 331
vi, *1449*
vi *(edit)*, 1454
View, 1552
viewport, *787*, *793*, *799*, *802*, *803*, *808*, *816*, *819*, *821*, *822*, *824*, *825*, *827*, *831*, *832*, *836*, *838*, *840*, *842*, *843*, *845*, *847*, *849*, *851*, *853*, *860*, *862*
viewport *(Grid Viewports)*, 794
vignette, *1433*, 1553
vignetteDepends, 1422
vignettes, *783*
vignettes *(vignette)*, 1553
VIRTUAL-class *(BasicClasses)*, 870
volcano, 567

vpList *(Grid Viewports)*, 794
vpPath, 860, *862*
vpStack *(Grid Viewports)*, 794
vpTree *(Grid Viewports)*, 794

warning, *18*, *51*, *191*, *247*, *273*, *297*, *393*, *417*, *418*, 490, *491*, *1424*
warnings, *299*, *490*, 491, *491*
warpbreaks, 568
weekdays, *95*, 492
weekdays.POSIXt, *98*
Weibull, 1386
weighted.mean, *267*, 1388
weighted.residuals, *1150*, 1388
weights, *1201*, *1388*
weights *(lm.summaries)*, 1149
weights.glm *(glm)*, 1102
which, 493, *495*
which.is.max, *495*
which.max, *266*
which.max *(which.min)*, 494
which.min, *149*, 494, *494*
while, *362*
while *(Control)*, 82
while-class *(language-class)*, 909
widehat *(plotmath)*, 619
widetilde *(plotmath)*, 619
width.SJ, *1003*
widthDetails, *784*, 861
wilcox.exact, *1391*
wilcox.test, *1136*, *1137*, *1226*, *1227*, *1306*, 1389, *1393*
wilcox_test, *1391*
Wilcoxon, 1392
window, *1369–1371*, 1394
window<- *(window)*, 1394
with, *26*, *29*, 495, *917*, *1436*
withCallingHandlers *(conditions)*, 70
within *(with)*, 495
withRestarts *(conditions)*, 70
withVisible, *211*, 1554
women, 569
Working with Viewports, 861
WorldPhones, 569
write, *118*, *120*, *316*, *380*, 497, *1557*
write.csv *(write.table)*, 1555
write.csv2 *(write.table)*, 1555
write.dcf, *1424*
write.dcf *(dcf)*, 98
write.ftable *(read.ftable)*, 1285
write.matrix, *1557*
write.socket *(read.socket)*, 1510

write.table, *99*, *497*, *1515*, 1555
write_PACKAGES, 1423
writeBin, *78*, *349*, *498*
writeBin *(readBin)*, 346
writeChar, *78*, *347*, *498*
writeChar *(readChar)*, 349
writeLines, *78*, *347*, *350*, *352*, *393*, 498
wsbrowser *(browseEnv)*, 1430
WWWusage, 570

X11, *48*, *133*, *415*, *576*, *595*, *622*, *624–626*, *644*, *651*, *703*, *714*, *776*
X11 *(x11)*, 647
x11, 647, *731*
X11.options, *625*
X11.options *(x11)*, 647
X11Font *(X11Fonts)*, 650
X11Fonts, 650, *650*
xaxisGrob *(grid.xaxis)*, 844
xDetails, 864
xedit *(edit)*, 1454
xemacs *(edit)*, 1454
xfig, *594*, 651, *750*
xgettext, *191*, 1424
xgettext2pot *(xgettext)*, 1424
xinch *(units)*, 778
xngettext *(xgettext)*, 1424
xor *(Logic)*, 246
xspline, 779, *847*
xsplineGrob *(grid.xspline)*, 845
xtabs, *455*, *568*, *1096*, *1286*, 1396
xtfrm, *304*, *339*, *399*, 499
xtfrm.numeric_version *(numeric_version)*, 293
xy.coords, *578, 579*, 653, *655*, *657*, *702*, *708*, *710*, *713*, *740, 741*, *749, 750*, *754*, *771*, *773*, *775*, *779*, *984*, *1128*, *1161*, *1298*, *1322*
xyinch *(units)*, 778
xyTable, 655, *771, 772*
xyz.coords, 656

yaxisGrob *(grid.yaxis)*, 848
yDetails *(xDetails)*, 864
yinch *(units)*, 778

zapsmall, *1272*
zapsmall *(Round)*, 365
zip.file.extract, 1558
zpackages, 499
zutils, 500