Le paquet *luadraw* (3.7)

Patrick Fradin

5 novembre 2026

Dessin 2D et 3D avec Lua et TikZ.

https://github.com/pfradin/luadraw

Résumé

Le paquet *luadraw* définit l'environnement du même nom, celui-ci permet de créer des graphiques mathématiques en utilisant le langage Lua. Ces graphiques sont dessinés au final par TikZ (et automatiquement sauvegardés), alors pourquoi les faire en Lua ? Parce que celui-ci apporte toute la puissance d'un langage de programmation simple, efficace, capable de faire des calculs, tout en utilisant les possibilités graphiques de TikZ.

Table des matières

1 Dessin 2D **10**

I Introduction . . . 10
1) Nouveautés depuis la version 3.0 . . 10
2) Prérequis . . . 11
3) Options de l'environnement *luadraw* 12
4) La classe ld.cpx (complexes) 12
5) Afficher une variable dans le terminal 13
6) Création d'un graphe 13
7) Sauvegarde, restauration des paramètres . . . 15
8) Peut-on utiliser directement du TikZ dans l'environnement *luadraw*? . . 15

II Méthodes graphiques . . . 15
1) Lignes polygonales 15
Dpolyline . . . 15
Options de dessin d'une ligne polygonale . . . 16
Dpolyline2 . . . 18
2) Segments et droites 18
Dangle . . . 18
Dbissec . . . 18
Dhline . . . 19
Dline . . . 19
DlineEq . . . 19
Dmarkarc . . . 19
Dmarkseg . . . 19
Dmed . . . 19
Dparallel . . . 20
Dperp . . . 20
Dseg . . . 20
Dtangent . . . 20
DtangentC . . . 20
DtangentI . . . 21
Dtangent_from . . . 21
Dcommon_tangent . . . 22
Dparallel_tangent . . . 23
Dnormal . . . 23
DnormalC . . . 23
DnormalI . . . 24
3) Figures géométriques 25
Darc . . . 25
Dcircle . . . 25
Dellipse . . . 25
Dellipticarc . . . 26
Dparallelogram . . . 26
Dpolyreg . . . 26
Drectangle . . . 27
Dsequence . . . 27
Dsquare . . . 28
Dwedge . . . 28
4) Courbes . . . 29
Paramétriques : Dparametric . . . 29
Polaires : Dpolar . . . 29
Cartésiennes : Dcartesian . . . 29
Fonctions périodiques : Dperiodic . 30
Fonctions en escaliers : Dstepfunction 30
Fonctions affines par morceaux : Daffinebypiece . . . 31
Équations différentielles : Dodesolve 31
Courbes implicites : Dimplicit . . . 33
Courbes de niveau : Dcontour . . . 33
Paramétrisation d'une ligne polygonale : *curvilinear_param* . 34
5) Domaines liés à des courbes cartésiennes . . . 35
Ddomain1 . . . 35
Ddomain2 . . . 35
Ddomain3 . . . 35
Dinequalities . . . 36
Dimplicit_inequalities . . . 37
6) Points (Ddots) et labels (Dlabel) . . 38
7) Chemins : Dpath, Dspline et Dtcurve 40
Qu'est ce qu'un chemin . . . 40
Fonctions sur les chemins . . . 41
Dessiner un chemin . . . 42
8) Paramétrer un chemin . . . 43
9) Chemins et clipping : Beginclip() et Endclip() . . . 44
10) Axes et grilles . . . 45
Daxes . . . 46
DaxeX et DaxeY . . . 48
Dgradline . . . 49
Dgrid . . . 51
Dgradbox . . . 52
11) Dessins d'ensembles (diagrammes de Venn) . . . 53
Dessiner un ensemble . . . 53
Opérations sur les ensembles . . . 54
12) Importer une image . . . 55
Placer une image dans le graphique 55
Mapper une image sur un parallélogramme . . . 56

13) Les couleurs . . . 57
Calculs sur les couleurs . . . 57
III Constructions géométriques . . . 59
1) circumcircle(), incircle() . . . 59
2) cvx_hull2d() . . . 59
3) delaunay() . . . 59
4) voronoi() . . . 60
5) line2strip() . . . 61
6) parallel_polyline() . . . 62
7) sss_triangle() . . . 62
8) sas_triangle() . . . 62
9) asa_triangle() . . . 62
IV Calculs sur les listes . . . 63
1) concat . . . 63
2) cut . . . 63
3) cutpolyline . . . 63
4) cutpolyline2 . . . 64
5) getbounds . . . 65
6) getdot . . . 65
7) insert . . . 65
8) interCC . . . 66
9) interD . . . 67
10) interDC . . . 67
11) interDL . . . 67
12) interL . . . 67
13) interP . . . 67
14) linspace . . . 67
15) map . . . 67
16) merge . . . 67
17) range . . . 67
18) read_csv_file . . . 67
19) Fonctions de clipping . . . 68
20) Ajout de fonctions mathématiques . 68
Évaluation protégée : evalf . . . 68
int . . . 69
gcd . . . 69
lcm . . . 69
nth_root . . . 69
solve . . . 69
V Transformations . . . 71
1) affin . . . 71
2) ftransform . . . 71
3) hom . . . 71
4) inv . . . 71
5) proj . . . 71
6) projO . . . 71
7) rotate . . . 71
8) shift . . . 71
9) simil . . . 71
10) sym . . . 71
11) symG . . . 71
12) symO . . . 72
VI Calcul matriciel . . . 72
1) Calculs sur les matrices . . . 72
applymatrix et applyLmatrix . . . 72
composematrix . . . 73
getmatrix . . . 73
invmatrix . . . 73
matrixof . . . 73
mtransform et mLtransform . . . 73
2) Matrice associée au graphe . . . 73
g:Composematrix() . . . 73
g:Det2d() . . . 73
g:IDmatrix() . . . 73
g:Mtransform() . . . 74
g:MLtransform() . . . 74
g:Rotate() . . . 74
g:Scale() . . . 75
g:Savematrix() et g:Restorematrix() 75
g:Setmatrix() . . . 75
g:Shift() . . . 75
3) Changement de vue. Changement de repère . . . 76
VII Ajouter ses propres méthodes à la classe ld.graph . . . 77
1) Un exemple . . . 77
2) Comment importer le fichier . . . 78
3) Modifier une méthode existante . . 80

2 Dessin 3D **82**
I Introduction . . . 82
1) Prérequis . . . 82
2) Quelques rappels . . . 82
3) Création d'un graphe 3D . . . 83
4) Modes de projection affine . . . 84
5) Projection centrale . . . 85
II La classe pt3d . . . 86
1) Représentation des points et vecteurs 86
2) Opérations sur les points 3D . . . 87
3) Méthodes de la classe *pt3d* . . . 87
4) Afficher une variable dans le terminal 87
III Méthodes graphiques élémentaires . . . 87
1) Dessin aux traits . . . 88
Ligne polygonale : Dpolyline3d . . . 88
Angle droit : Dangle3d . . . 88
Segment : Dseg3d, Dmarkseg3d . . 88
Droite : Dline3d . . . 88
Arc de cercle : Darc3d . . . 89
Cercle : Dcircle3d . . . 89
Arc d'ellipse : Dellipticarc3d . . . 89
Ellipse : Dellipse3d . . . 89
Chemin 3D : Dpath3d . . . 90
Fonctions sur les chemins 3D . . . 90

Plan : Dplane 91
Courbe paramétrique : Dparametric3d 92
Paramétrisation d'une ligne polygonale : *curvilinear_param3d* 93
Le repère : Dboxaxes3d 93
Dessiner sur un plan 94
2) Points et labels 95
Points 3D : Ddots3d, Dballdots3d, Dcrossdots3d 95
Labels 3D : Dlabel3d 96
3) Solides de base (sans facette) 97
Cylindre : Dcylinder 97
Cône : Dcone 98
Tronc de cône : Dfrustum 98
Sphère : Dsphere 98
Sphère coupée : Dcut_sphere 99
Cone_outline, Cylinder_outline, Frustum_outline, Sphere_outline 100
Cone_tangency, Cylinder_tangency, Frustum_tangency, Sphere_tangency 101
IV Solides à facettes 102
1) Définition d'un solide 102
2) Dessin d'un polyèdre : Dpoly 103
3) Visualiser les numéros des facettes et/ou ceux des sommets d'un polyèdre 104
4) Fonctions de construction de polyèdres 105
5) Conversion d'un polyèdre en graphe 109
6) Lecture dans un fichier obj 109
7) Surface au format obj 110
8) Dessin d'une liste de facettes : Dfacet et Dmixfacet 110
9) Fonctions de construction de listes de facettes 112
surface() 112
cartesian3d() 112
cylindrical_surface() 113
curve2cone() 114
curve2cylinder() 115
line2tube(); section2tube() 116
read_table3d() 117
rotcurve() 120
rotline() 121
10) Arêtes d'un solide 122
11) Méthodes et fonctions s'appliquant à des facettes ou polyèdres 123
12) Découper un solide : cutpoly, cutfacet et cutfacet2 124
13) Clipper des facettes avec un polyèdre convexe : clip3d 126
14) Clipper un plan avec un polyèdre convexe : clipplane 127
V La méthode Dscene3d 127
1) Le principe, les limites 127
2) Construction d'une scène 3D 128
3) Méthodes pour ajouter un objet dans la scène 3D 129
Ajouter des facettes : g:addFacet et g:addPoly 129
Ajouter un plan : g:addPlane et g:addPlaneEq 130
Ajouter une ligne polygonale : g:addPolyline 130
Ajouter des axes : g:addAxes 131
Ajouter une droite : g:addLine . . . 131
Ajouter un angle "droit" : g:addAngle 131
Ajouter un arc de cercle : g:addArc . 131
Ajouter un cercle : g:addCircle . . . 131
Ajouter des points : g:addDots . . . 133
Ajouter des labels : g:addLabel . . . 133
Ajouter des cloisons séparatrices : g:addWall 134
VI Constructions géométriques 137
1) Cercle circonscrit, cercle inscrit : circumcircle3d(), incircle3d() 137
2) Enveloppe convexe : cvx_hull3d() . 137
3) Plans : plane(), planeEq(), orthoframe(), plane2ABC() 138
4) Sphère circonscrite, Sphère inscrite : circumsphere(), insphere() 139
5) Tétraèdre à longueurs fixées : tetra_len() 139
6) Triangles : sss_triangle3d(), sas_triangle3d(), asa_triangle3d() 140
VII Transformations calcul matriciel et quelques fonctions mathématiques 141
1) Transformations 3D 141
Appliquer une fonction de transformation : ftransform3d . . 141
Projections : proj3d, proj3dO, dproj3d 141
Projections sur les axes ou les plans liés aux axes 141
Symétries : sym3d, sym3dO, dsym3d, psym3d 141
Rotation : rotate3d, rotateaxe3d . . 141
Homothétie : scale3d 142
Inversion : inv3d 142
Stéréographie : projstereo et inv_projstereo 142

Translation : shift3d 142
2) Calcul matriciel 142
applymatrix3d et applyLmatrix3d . 142
composematrix3d 142
getmatrix3d 142
invmatrix3d 142
matrix3dof 143
mtransform3d et mLtransform3d . 143
planetransform 143
3) Matrice associée au graphe 3D . . . 143
g:Composematrix3d() 143
g:Det3d() 143
g:IDmatrix3d() 143
g:Mtransform3d() 143
g:MLtransform3d() 143
g:Rotate3d() 143
g:Scale3d() 144
g:Setmatrix3d() 144
g:Shift3d() 144
4) Fonctions mathématiques supplémentaires 144
clippolyline3d() 144
clipline3d() 144
cutpolyline3d() 144
getbounds3d() 144
interDP() 145
interPP() 145
interDD() 145
interCS() 145
interDS() 145
interPS() 145
interSS() 145
interSSS() 145
merge3d() 145
split_points_by_visibility() 145
VIII Exemples plus poussés 147
1) La boîte de sucre 147
2) Empilement de cubes 148
3) Illustration du théorème de Dandelin 150
4) Volume défini par une intégrale double 152
5) Volume défini sur autre chose qu'un pavé 153

3 Annexes **155**
I Extensions 155
1) Le module *luadraw_polyhedrons* . . 155
2) Le module *luadraw_spherical* . . . 157
Fonctions globales du module . . . 158
Définition de la sphère 158
Ajout d'un label aux éléments graphiques 159
Ajouter un cercle : g:DScircle 159
Ajouter un grand cercle : g:DSgreatcircle 160
Ajouter un arc de grand cercle : g:DSarc 160
Ajouter un angle : g:DSangle 160
Ajouter une facette sphérique : g:DSfacet 161
Ajouter une courbe sphérique : g:DScurve 161
Ajouter une région sphérique : g:DSregion 162
Ajouter un segment : g:DSseg . . . 162
Ajouter une droite : g:DSline 162
Ajouter une ligne polygonale : g:DSpolyline 163
Ajouter des axes : g:DSaxes 163
Ajouter un plan : g:DSplane 163
Ajouter un label : g:DSlabel 164
Ajouter des points : g:DSdots et g:DSstars 164
Stéréographie inverse : g:DSinvstereo_curve et g:DSinvstereo_polyline 164
Ajouter des chemins 164
Méthodes supplémentaires : g:Dhalf_sphere, g:Dquater_sphere, g:Deighth_sphere 165
Exemples 165
3) Le module *luadraw_palettes* 174
4) Le module *luadraw_compile_tex* . . 174
Première partie : compilation et lecture 175
Deuxième partie : exploitation du résultat 175
5) Le module *luadraw_cvx_polyhedra_nets* 178
La fonction de base 178
La méthode de dessin 179
Exemples 179
La fonction *unfold_tree()* 181
6) Le module *luadraw_fields* 183
Champs 2D 183
Champs 3D 183
7) Le module *luadraw_shadedforms* . 186
Dshadedpolyline 186
Dcolorbar 187
Dshadedrectangle 188
Dshadedregion 189
8) Le module *luadraw_povray* 190
Prérequis et introduction 190

Réglages par défaut 190
Avant la création des objets 191
Création des objets 191
Liste des objets prédéfinis 192
Sauvegarde, exécution, inclusion . . 195
Exemples 195
9) Le module *luadraw_log_axes* 198
Initialisation : g:Beginlogview() . . . 198
Méthodes de dessin et conversion . 200
Fin de l'utilisation de la grille : g:Endlogview() 200
Exemples 200
10) Le module *luadraw_decorations* . . 202
Lignes polygonales 2D : g:Dpolyline() 202
Chemins 2D : g:Dpath() 203
Droites 2D : g:Dline() 203
Segments 2D : g:Dseg() 203
Arc 2D : g:Darc() 203
Arc 3D : g:Darc3d() 205
Angle 3D : g:Dangle3d() 207
11) Le module *luadraw_coils_chains* . . 209
Les ressorts 209
Les chaînes 211
12) Le module *luadraw_pdfliteral* . . . 213
La méthode g:Dliteralpath() 214
La méthode g:Dliteralpolyline() . . 214
La méthode g:Dliteraldots() 214
La méthode g:Dliteralfacet() 215
13) Le module *luadraw_linprog* 217
En 2D 217
En 3D 220
II Historique 222
1) Version 3.7 222
2) Version 3.6 222
3) Version 3.5 223
4) Version 3.4 224
5) Version 3.3 224
6) Version 3.2 225
7) Version 3.1 226
8) Version 3.0 226
9) Version 2.8 227
10) Version 2.7 228
11) Version 2.6 228
12) Version 2.5 228
13) Version 2.4 229
14) Version 2.3 229
15) Version 2.2 229
16) Version 2.1 230
17) Version 2.0 230
18) Version 1.0 230

Table des figures

1 Un premier exemple : trois sous-figures dans un même graphique . . . 10
2 Champ de vecteurs, courbe intégrale de $y' = 1 - xy^2$ . . . 18
3 Tangentes issues d'un point . . . 22
4 Tangentes communes . . . 23
5 Symétrique de l'orthocentre . . . 25
6 polygone régulier étoilé . . . 27
7 Suite $u_{n+1} = \cos(u_n)$ . . . 28
8 Un système différentiel de Lotka-Volterra . . . 32
9 Exemple avec Dcontour . . . 34
10 Points répartis sur une ligne polygonale . . . 35
11 Partie entière, fonctions Ddomain1 et Ddomain3 . . . 36
12 Dinequalities . . . 37
13 La méthode **g:Dimplicit_inequalities** . . . 38
14 Path example . . . 41
15 Path et Spline . . . 42
16 Courbe d'interpolation avec vecteurs tangents imposés . . . 43
17 Paramétrisation d'une spline . . . 44
18 Exemple de clipping . . . 45
19 Exemple avec axes avec grille . . . 48
20 Exemples de droites graduées . . . 51
21 Exemple de repère non orthogonal . . . 52
22 Utilisation de Dgradbox . . . 53
23 Dessiner un ensemble . . . 54
24 Opérations sur les ensembles . . . 55
25 Importer une image . . . 56
26 Mapper une image . . . 57
27 Les cinq palettes par défaut . . . 59
28 Triangulation de Delaunay . . . 60
29 Diagramme de Voronoï . . . 61
30 Exemple avec *line2strip* . . . 62
31 sss_triangle, sas_triangle et asa_triangle . . . 63
32 Illustrer un exercice de programmation linéaire . . . 64
33 cutpolyline2 . . . 65
34 Tangentes à un cercle {O,2} et à une ellipse {O,3,2} issues d'un point . . . 66
35 Fonction f définie par $\int_x^{f(x)} \exp(t^2)\mathrm{d}t = 1$. . . . 70
36 Utilisation de transformations . . . 72
37 Utilisation de la matrice du graphe . . . 74
38 Utilisation de Shift, Rotate et Scale . . . 75
39 Classification des points d'une courbe paramétrée . . . 77
40 Utilisation des nouvelles méthodes . . . 79
41 Modification d'une méthode existante . . . 81

1 Point col en M(0,0,0) ($z = x^2 - y^2$) . . . 82
2 Angles de vue . . . 84
3 Modes de projection affine . . . 85
4 Modes "symétriques" des précédents . . . 85
5 Projection centrale . . . 86
6 Dplane, exemple avec mode = left+bottom . . . 92
7 Une courbe et ses projections sur trois plans . . . 93
8 Dessiner sur un plan . . . 95
9 Un tétraèdre et les centres de gravité de chaque face . . . 96
10 Cylindres, cônes et sphères . . . 99
11 Sphère coupée . . . 100
12 Points de tangence . . . 102
13 Section d'un tétraèdre par un plan . . . 104
14 Visualiser les faces et sommets d'un polyèdre . . . 105
15 Cône tronqué, pyramide tronquée, cylindre oblique . . . 107
16 Hyperbole : intersection cône - plan . . . 107
17 Section de cône avec plusieurs vues . . . 109
18 Masque de Nefertiti . . . 110
19 Exemple de courbes de niveaux sur une surface . . . 112
20 Surfaces utilisant l'option *addwall* . . . 114
21 Exemple de cône elliptique . . . 115
22 Section d'un cylindre non circulaire . . . 116
23 Exemple avec line2tube et section2tube . . . 117
24 La fonction *ld.read_table3d()* . . . 120
25 Exemple avec rotcurve . . . 121
26 Exemple avec rotline . . . 122
27 Sphère inscrite dans un octaèdre avec projection du centre sur les faces . . . 124
28 Cube coupé par un plan (cutpoly), avec *close*=false et avec *close*=true . . . 125
29 La fonction *ld.cutfacet2* . . . 126
30 Exemple avec clip3d : construction d'un dé à partir d'un cube et d'une sphère . . . 127
31 Premier exemple avec Dscene3d : intersection de deux plans . . . 129
32 Cylindre plein plongé dans de l'eau . . . 132
33 Construction d'un icosaèdre . . . 134
34 Exemple avec addWall (les deux facettes transparentes roses sont normalement invisibles) . . . 135
35 Tore et lemniscate . . . 136
36 Section de sphère sans Dscene3d() . . . 137
37 Utilisation de cvx_hull3d() . . . 138
38 Faces d'un cube trouées avec un hexagone régulier . . . 139
39 Un tétraèdre avec la longueur des arêtes fixée . . . 140
40 Une courbe sur un cylindre . . . 146
41 Boite de morceaux de sucre . . . 148
42 Empilement de cubes . . . 150
43 Illustration du théorème de Dandelin . . . 151
44 Volume correspondant à $\int_{x_1}^{x_2}\int_{y_1}^{y_2} f(x,y)\,dx\,dy$ . . . 153
45 Volume : $0 \leqslant x \leqslant 1;\ 0 \leqslant y \leqslant x^2;\ 0 \leqslant z \leqslant y^2$ . . . 154

1 Polyèdres du module *luadraw_polyhedrons* . . . 157
2 Cube dans une sphère . . . 166
3 Fenêtre de Viviani . . . 167
4 Un pavage sphérique . . . 168
5 Tangentes à la sphère issues d'un point . . . 169
6 Projection stéréographique d'un cercle et d'une facette sphériques . . . 170

7 Méthodes *DSinvstereo_curve* et *DSinvstereo_polyline* . . . 170
8 Ajouter des chemins . . . 172
9 Peindre une région sphérique . . . 173
10 Exemple avec **g:Dhalf_sphere** et compagnie . . . 174
11 Exemple avec *compile_tex* en 2D . . . 176
12 Écrire sur un cylindre . . . 178
13 Patron d'un parallélépipède . . . 180
14 Patron imposé d'un parallélépipède . . . 180
15 Parallélépipède tronqué à demi déplié . . . 181
16 Dépliage d'un dodécaèdre . . . 183
17 La méthode *g:Dsurfacefield()* . . . 185
18 Sur une sphère . . . 186
19 Shaded polyline . . . 187
20 Color bars . . . 188
21 Shaded rectangle . . . 189
22 Shaded region . . . 190
23 Intersection de deux surfaces implicites . . . 196
24 Cercles de Villarceau . . . 197
25 Trous dans une demi-sphère . . . 198
26 Axe des x logarithmique . . . 201
27 Les deux autres cas de figure. . . . 202
28 Méthode *g:Darc()* . . . 205
29 Méthode *g:Darc3d()* . . . 207
30 Décorations 2D . . . 208
31 Décorations 3D . . . 209
32 Méthode *g:Dcoil()* . . . 210
33 Méthode *g:Dcoil2()* . . . 211
34 Méthode *g:Dchain()* . . . 212
35 Méthode *g:Dchain2()* . . . 213
36 Diagramme de bifurcation de la suite $u_{u+1} = r\,u_n(1-u_n)$ . . . 215
37 Bandes sphériques . . . 216
38 La méthode *g:DlinprogRegion()* . . . 219
39 La méthode *g:DlinprogHalfPlanes()* . . . 220
40 La fonction *ld.linprogSolve3d()* . . . 222

Chapitre 1

Dessin 2D

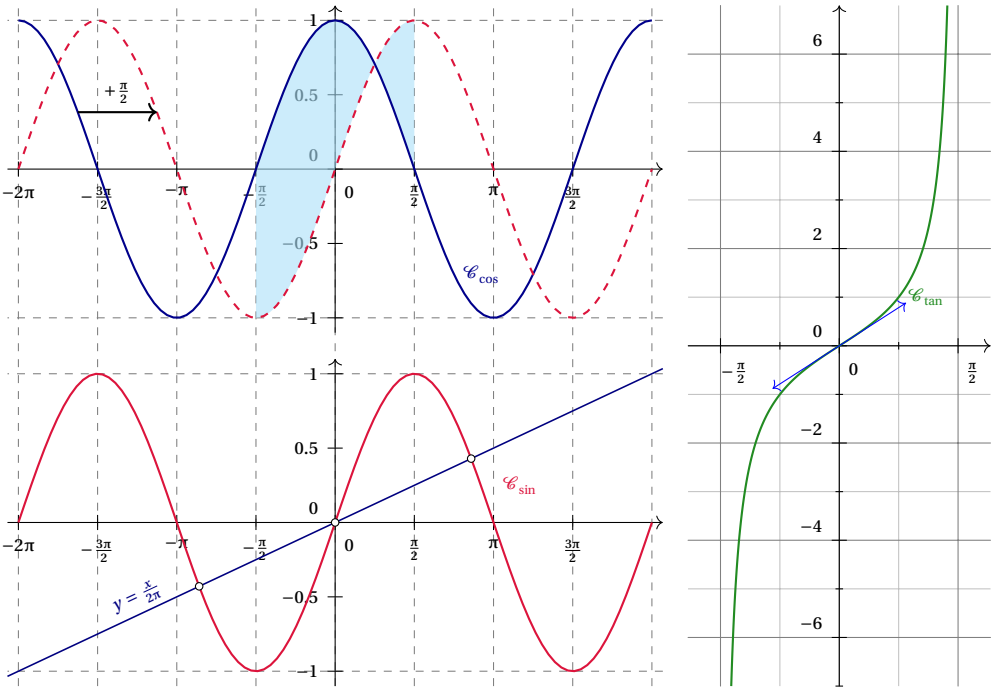


FIGURE 1 : Un premier exemple : trois sous-figures dans un même graphique

I Introduction

1) Nouveautés depuis la version 3.0

Depuis la version 3.0, **toutes** les données relatives au paquet *luadraw* sont encapsulées dans un même espace de noms, celui-ci s'appelle **luadraw** (c'est une table). Ceci a pour conséquence que les données sont accessibles uniquement avec la notation pointée, plus précisément :

1. la classe *graph* devient *luadraw.graph*,
2. la classe *cpx* (nombres complexes) devient *luadraw.cpx*,
3. la classe *graph3d* devient *luadraw.graph3d*,
4. la classe *pt3d* (points 3D) devient *luadraw.pt3d*,
5. toutes les variables globales, toutes les fonctions non graphiques sont dans l'espace de nommage *luadraw*, sauf quelques exceptions propres aux nombres complexes qui sont passées dans la classe *cpx*, et quelques exceptions propres aux points 3D qui sont passées dans la classe *pt3d*,
6. par contre il n'y a pas de changement pour les méthodes graphiques puisqu'elles étaient déjà encapsulées dans les classes *graph* et *graph3d*.

Le langage Lua n'autorise pas la factorisation de l'espace de noms, il faut donc réellement utiliser la notation pointée, mais heureusement on peut se créer des raccourcis (ou alias), par exemple :

```
local ld = luadraw -- alias sur l'espace de nommage
local cpx, pt3d = ld.cpx, ld.pt3d -- raccourcis pour les classes cpx et pt3d
local Z, M, i = cpx.Z, pt3d.M, cpx.I -- raccourcis vers les fonctions Z et M et le complexe i
local Origin, vecI, vecJ, vecK = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK
local g = ld.graph3d:new{}
...
```

Du fait de ce changement majeur, les scripts *luadraw* des versions antérieures **ne sont plus compilables directement**, ils doivent être adaptés à cette nouvelle version. Mais avec des raccourcis, les changements sont minimes.

Autre nouveauté : l'environnement *luadraw* fait appel à un environnement *luacode* (et non plus *luacode**) ce qui permet d'introduire des macros TEX dans le code *Lua*, on peut donc mettre les raccourcis les plus fréquents dans une macro et l'utiliser dans ses codes *luadraw*, par exemple :

```
\documentclass{article}
\usepackage[3d]{luadraw}

\newcommand*{\shortcuts}{%
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Z, M, i = cpx.Z, pt3d.M, cpx.I
local Origin, vecI, vecJ, vecK = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK
}%

\begin{document}

\begin{luadraw}{name=test}
\shortcuts
local g = ld.graph3d:new{}
local P = ld.polyreg(0, Z(2,3), 5)
g:Dpolyline(P, true, "red,line width=1.2pt")
local Q = ld.map(pt3d.toPoint3d, P)
P = ld.pyramid(Q, M(0,0,4))
g:Dpoly(P, {color="blue", opacity=0.6})
g:Show()
\end{luadraw}
\end{document}
```

L'environnement étoilé *luadraw** quant à lui, fait appel à un environnement *luacode**, dans ce cas il n'y a aucune ingérence de TEX dans le code Lua.

NB : dans toute la suite de cette documentation nous utiliserons les raccourcis suivants :

```
local ld = luadraw -- alias sur l'espace de nommage
local cpx = ld.cpx -- raccourci pour la classe cpx
local pt3d = ld.pt3d -- raccourci pour la classe pt3d
```

2) Prérequis

- Dans le préambule, il faut déclarer le package *luadraw* : `\usepackage[options globales]{luadraw}`
- La compilation se fait avec LuaLateX **exclusivement**.
- Les couleurs dans l'environnement *luadraw* sont des chaînes de caractères qui doivent correspondre à des couleurs connues de TikZ. Il est fortement conseillé d'utiliser le package *xcolor* avec l'option *svgnames*.

Quelque soient les options globales choisies, ce paquet charge le module *luadraw_graph2d.lua* qui définit la classe *ld.graph*, et fournit l'environnement *luadraw* qui permet de faire des graphiques en Lua. Pour pouvoir faire des graphiques en 3D il faut charger en plus la classe *ld.graph3d* grâce à l'option globale `3d` :

`\usepackage[3d]{luadraw}`.

Options globales du paquet : `noexec`, `3d` et `cachedir=`.

- `noexec` : lorsque cette option globale est mentionnée la valeur par défaut de l'option *exec* pour l'environnement *luadraw* sera false (et non plus true).
- `3d` : lorsque cette option globale est mentionnée, le module *luadraw_graph3d.lua* est également chargé. Celui-ci définit en plus la classe *ld.graph3d* (qui s'appuie sur la classe *ld.graph*) pour des dessins en 3D.
- `cachedir=<dossier>` : par défaut les fichiers créés sont enregistrés dans le dossier *_luadraw* qui est un sous-dossier du dossier courant (contenant le document maître). Ce dossier peut être changé avec l'option `cachedir`, par exemple `cachedir={tikz}`.

NB : dans ce chapitre nous ne parlerons pas de l'option `3d`. Celle-ci fait l'objet du chapitre suivant. Nous ne parlerons donc que de la version 2D.

Lorsqu'un graphique est terminé il est exporté au format TikZ, donc ce paquet charge également le paquet TikZ ainsi que les librairies :

- *patterns, patterns.meta*
- *plotmarks*
- *arrows.meta, bending*
- *decorations.markings*
- *backgrounds*

Les graphiques sont créés dans un environnement *luadraw*, celui-ci appelle *luacode*, c'est donc du **langage Lua** qu'il faut utiliser dans cet environnement :

```
\begin{luadraw}{ name = <filename>, exec = true/false, auto = true/false }
local ld = luadraw
-- création d'un nouveau graphique en lui donnant un nom local (cette ligne est un commentaire)
local g = ld.graph:new{ window={x1,x2,y1,y2 [,xscale,yscale]}, margin={top,right,bottom,left},
                    size={width,height,ratio}, bg="color", border=true/false }
-- construction du graphique g
    instructions graphiques en langage Lua ...
-- affichage du graphique g et sauvegarde dans le fichier <filename>.tkz
g:Show()
-- ou bien sauvegarde uniquement dans le fichier <filename>.tkz
g:Save()
\end{luadraw}
```

Sauvegarde du fichier **.tkz* : le graphique est exporté au format TikZ dans un fichier (avec l'extension *tkz*), par défaut celui-ci est sauvegardé dans le dossier *_luadraw* qui est un sous-dossier du dossier courant (contenant le document maître), mais il est possible d'imposer un chemin vers un autre sous-dossier avec l'option globale `cachedir=...`.

3) Options de l'environnement *luadraw*

Celles-ci sont :

- `name=...` : permet de donner un nom au fichier TikZ produit, on donne un nom sans extension (celle-ci sera automatiquement ajoutée, c'est *.tkz*). Si cette option est omise, alors il y a un nom par défaut, qui est le nom du fichier maître suivi d'un numéro.
- `exec=true/false` : permet d'exécuter ou non le code Lua compris dans l'environnement. Par défaut cette option vaut true, **SAUF** si l'option globale `noexec` a été mentionnée dans le préambule avec la déclaration du paquet. Lorsqu'un graphique complexe qui demande beaucoup de calculs est au point, il peut être intéressant de lui ajouter l'option `exec=false`, cela évitera les recalculs de ce même graphique pour les compilations à venir.
- `auto=true/false` : permet d'inclure ou non automatiquement le fichier TikZ en lieu et place de l'environnement *luadraw* lorsque l'option `exec` est à `false`. Par défaut l'option `auto` vaut `true`.

4) La classe ld.cpx (complexes)

Rappel : nous utilisons l'alias `local cpx = ld.cpx`.

Elle est automatiquement chargée par le module *luadraw_graph2d* et donc au chargement du paquet *luadraw*. Cette classe permet de manipuler les nombres complexes et de faire les calculs habituels. On crée un complexe avec la fonction **cpx.Z(a, b)** pour $a + i \times b$, ou bien avec la fonction **cpx.Zp(r, theta)** pour $r \times e^{i\theta}$ en coordonnées polaires.

- Exemple : `local z = cpx.Z(a,b)` va créer le complexe correspondant à $a + i \times b$ dans la variable z. On accède alors aux parties réelle et imaginaire de z comme ceci : `z.re` et `z.im`. On peut bien sûr créer un raccourci vers cette fonction avec l'instruction `local Z = cpx.Z` en début de code.
- **Attention** : un nombre réel x n'est pas considéré comme complexe par Lua. Cependant, les fonctions proposées pour les constructions graphiques font la vérification et la conversion réel vers complexe. On peut néanmoins, utiliser $Z(x,0)$ à la place de x.
- Les opérateurs habituels ont été surchargés ce qui permet l'utilisation des symboles habituels, à savoir : +, x, -, /, ainsi que le test d'égalité avec = (ce test d'égalité est fait à `ld.epsilon` près où `ld.epsilon` et une variable globale qui vaut $1e-16$ par défaut). Lorsqu'un calcul échoue le résultat renvoyé en principe doit être égal à `nil`.
- À cela s'ajoutent les fonctions suivantes :
  - le module : **cpx.abs(z)**,
  - le module au carré : **cpx.abs2(z)**,
  - la normalisation : **cpx.normalize(z)** (renvoie *nil* si z est nul),
  - la norme 1 : **cpx.N1(z)**,
  - l'argument principal : **cpx.arg(z)**,
  - le conjugué : **cpx.bar(z)**,
  - l'égalité parfaite : **cpx.equal(z1,z2)**,
  - l'exponentielle complexe : **cpx.exp(z)**,
  - le cosinus et sinus hyperboliques complexes : **cpx.cosh(z)** et **cpx.sinh(z)**,
  - la fonction puissance a : **cpx.pow(z,a)** (le calcul utilise l'argument principal de z),
  - le produit scalaire : **cpx.dot(z1, z2)**, où les complexes représentent des affixes de vecteurs,
  - le déterminant : **cpx.det(z1, z2)**,
  - l'angle orienté (en radians) entre deux vecteurs non nuls : **cpx.angle(z1, z2)**
  - l'arrondi : **cpx.round(z, nb decimales)**,
  - la fonction : **cpx.isNul(z)** teste si les parties réelle et imaginaire de z sont en valeur absolue inférieures à la variable globale `ld.epsilon` qui vaut $1e-16$ par défaut,
  - la fonction **cpx.isComplex(u)** teste si $\langle u \rangle$ est un complexe ou non et renvoie un booléen (attention : si $\langle u \rangle$ est réel, la fonction renvoie `false`),
  - la fonction **cpx.toComplex(x)** teste si $\langle x \rangle$ est un réel, si c'est le cas, elle renvoie le nombre complexe `cpx.Z(x,0)`, si $\langle x \rangle$ est un nombre complexe la fonction renvoie $\langle x \rangle$, et dans les autres cas, elle renvoie `nil`,
  - la fonction **cpx.isobar(L)** où $\langle L \rangle$ est une liste de nombres complexes, renvoie l'isobarycentre des points de $\langle L \rangle$.

  On dispose également de la constante `cpx.I` qui représente l'imaginaire pur $\imath$.

  Exemple :

```
local cpx = luadraw.cpx
local i = cpx.I
local A = 2+3*i
```

Le symbole de multiplication est obligatoire.

5) Afficher une variable dans le terminal

L'instruction **ld.whatis(variable [, msg])** affiche dans le terminal lors de la compilation, le type de la $\langle$*variable*$\rangle$ ainsi que son contenu. Les types reconnus sont, les types prédéfinis plus : *complex number*, *list of (complex) numbers*, *list of lists of (complex) numbers*. L'argument *msg* est une chaîne optionnelle (vide par défaut) qui est affichée avec le type pour repérer la variable dans le terminal.

6) Création d'un graphe

Comme cela a été vu plus haut, la création se fait dans un environnement *luadraw*, cette création se fait en nommant le graphique :

```
local ld = luadraw
local g = ld.graph:new{ window = {x1,x2,y1,y2 [,xscale,yscale]}, margin = {left,right,top,bottom},
                        size = {width,height,ratio}, bg = "color", border =true/false,
                        bbox = true/false, pictureoptions = "" }
```

La classe *ld.graph* est définie dans le paquet *luadraw*. On instancie cette classe en invoquant son constructeur et en donnant un nom (ici c'est *g*), on le fait en local de sorte que le graphique *g* ainsi créé, n'existera plus une fois sorti de l'environnement (sinon *g* resterait en mémoire jusqu'à la fin du document). Voici les options du constructeur **ld.graph:new** :

- `window={x1,x2,y1,y2 [,xscale,yscale]}`. Cette option (facultative) définit le pavé de $\mathbf{R}^2$ dans lequel se fait au graphique (c'est $[x_1;x_2]\times[y_1;y_2]$), ainsi que les échelles sur les axes, qui sont les arguments ⟨*xscale*⟩ et ⟨*yscale*⟩. Ces deux derniers paramètres sont facultatifs et valent 1 par défaut, ils représentent l'échelle (cm par unité) sur les axes. Par défaut on a `window={-5,5,-5,5,1,1}`.
- `margin={left,right,top,bottom}`. Cette option (facultative) définit des marges autour du graphique en cm. Par défaut on a `margin={0.5,0.5,0.5,0.5}`.
- `size={width,height [,ratio]}`. Cette option (facultative) permet d'imposer une taille (en cm, marges incluses) pour le graphique, l'argument ⟨*ratio*⟩ est facultatif et correspond au rapport d'échelle souhaité (⟨*xscale*⟩/⟨*yscale*⟩), un ratio de 1 donnera un repère orthonormé, et si le ratio n'est pas précisé alors le ratio par défaut est conservé (celui défini par l'option `window`). Un ratio égal à 0 détermine les échelles de telle sorte que le graphique ait exactement la taille demandée. L'utilisation de ce paramètre va modifier les valeurs de ⟨*xscale*⟩ et ⟨*yscale*⟩. La taille de la fenêtre par défaut est de 11 × 11 (en cm) avec les marges par défaut (10 × 10 sans les marges).
- `bg="color"`. Cette option (facultative) permet de définir une couleur de fond pour le graphique, cette couleur est une chaîne de caractères représentant une couleur pour TikZ. Par défaut cette chaîne est vide ce qui signifie que le fond ne sera pas peint.
- `border=true/false`. Cette option (facultative) indique si un cadre doit être dessiné ou non autour du graphique (en incluant les marges). Par défaut ce paramètre vaut `false`.
- `bbox=true/false`. Cette option (facultative) indique si une boundingbox doit être ajoutée au graphique de telle sorte que celui-ci ait la taille souhaitée, tout ce qui en sort est clippé par TikZ. Par défaut ce paramètre vaut `true`. Avec la valeur `false` il n'y a pas de boundingbox ajoutée, mais tout ce qui sort de la fenêtre 2D, sauf les path, est clippé par *luadraw*, la taille du graphique peut être plus petite que celle demandée.
- `pictureoptions=""`. Cette option (facultative) est une chaîne de caractères destinée à contenir des options qui seront passées à l'environnement *tikzpicture* comme ceci :

```
\begin{tikzpicture}[line join=round <,pictureoptions>]
```

Construction du graphique.

- L'objet instancié (*g* ici dans l'exemple) possède un certain nombre de méthodes permettant de faire du dessin (segments, droites, courbes,…). Les instructions de dessins ne sont pas directement envoyées à TEX, elles sont enregistrées sous forme de chaînes dans une table qui est une propriété de l'objet *g*. C'est la méthode **g:Show()** qui va envoyer ces instructions à TEX tout en les sauvegardant dans un fichier texte[1]. La méthode **g:Save()** enregistre le graphique dans le fichier désigné par l'option (de l'environnement) `name` mais sans envoyer les instructions à TEX.
- On peut faire une sauvegarde du graphique en cours dans un autre fichier avec la méthode :

  g:Savetofile(<nom de fichier avec extension>)
- On peut réinitialiser un graphique en cours, c'est-à-dire supprimer tous les éléments déjà créés et remettre tous les attributs à leur valeur par défaut, avec la méthode :

  g:Cleargraph()
- Le paquet *luadraw* fournit aussi un certain nombre de fonctions mathématiques, ainsi que des fonctions permettant des calculs sur les listes (tables) de complexes, des transformations géométriques, … etc. Ces fonctions sont dans l'espace de noms *luadraw*.

Système de coordonnées. Repérage

- L'objet instancié (*g* ici dans l'exemple) possède :
  1. Une vue originelle : c'est le pavé de $\mathbf{R}^2$ défini par l'option `window` à la création. Celui-ci **ne doit pas être modifié** par la suite.
  2. Une vue courante : c'est un pavé de $\mathbf{R}^2$ qui doit être inclus dans la vue originelle, ce qui sort de ce pavé sera clippé. Par défaut la vue courante est la vue originelle. Pour retrouver la vue courante on peut utiliser la méthode **g:Getview()** qui renvoie une table `{x1,x2,y1,y2}`, celle-ci représente le pavé $[x_1;x_2]\times[y_1;y_2]$.

1. Ce fichier contiendra un environnement *tikzpicture*.

3. Une matrice de transformation : celle-ci est initialisée à la matrice identité. Lors d'une instruction de dessin les points sont automatiquement transformés par cette matrice avant d'être envoyés à TikZ.
4. Un système de coordonnées (repère cartésien) lié à la vue courante, c'est le repère de l'utilisateur. Par défaut c'est le repère canonique de $\mathbf{R}^2$, mais il est possible d'en changer. Admettons que la vue courante soit le pavé $[-5;5] \times [-5;5]$, il est possible par exemple, de décider que ce pavé représente l'intervalle $[-1;12]$ pour les abscisses et l'intervalle $[0;8]$ pour les ordonnées, la méthode qui fait ce changement va modifier la matrice de transformation du graphe, de telle sorte que pour l'utilisateur tout se passe comme s'il était dans le pavé $[-1;12] \times [0,8]$. On peut retrouver les intervalles du repère de l'utilisateur avec les méthodes :

   g:Xinf(), **g:Xsup()**, **g:Yinf()** et **g:Ysup()**.

- On utilise les nombres complexes pour représenter les points ou les vecteurs dans le repère cartésien de l'utilisateur.
- Dans l'export TikZ les coordonnées seront différentes car le coin inférieur gauche (hors marges) aura pour coordonnées $(0,0)$, et le coin supérieur droit (hors marges) aura des coordonnées correspondant à la taille (hors marges) du graphique, et avec 1 cm par unité sur les deux axes. Ce qui fait que normalement, TikZ ne devrait manipuler que de « petits » nombres.
- La conversion se fait automatiquement avec la méthode **g:strCoord(x, y)** qui renvoie une chaîne de la forme *(a,b)*, où *a* et *b* sont les coordonnées pour TikZ, ou bien avec la méthode **g:Coord(z)** qui renvoie aussi une chaîne de la forme *(a,b)* représentant les coordonnées TikZ du point d'affixe *z* dans le repère de l'utilisateur.

7) Sauvegarde, restauration des paramètres

La méthode **g:Saveattr([scope_options])** sauvegarde les paramètres de *luadraw* dans une pile, et ouvre un environnement *scope*. L'argument ⟨*scope_options*⟩ est une chaîne (vide par défaut) qui sera transmise comme option à l'environnement *scope*.

La méthode **g:Restoreattr()** referme l'environnement *scope* et restaure les paramètres de *luadraw*.

8) Peut-on utiliser directement du TikZ dans l'environnement *luadraw* ?

Supposons que l'on soit en train de créer un graphique nommé *g* dans un environnement *luadraw*. Il est possible d'écrire une instruction TikZ lors de cette création, mais pas en utilisant `tex.sprint("<instruction TikZ>")`, car alors cette instruction ne ferait pas partie du graphique *g*. Il faut pour cela utiliser la méthode **g:Writeln("<instruction TikZ>")**, avec la contrainte que **les antislash doivent être doublés**, et sans oublier que les coordonnées graphiques d'un point dans *g* ne sont pas les mêmes pour TikZ. Par exemple :

```
g:Writeln("\\draw"..g:Coord(Z(1,-1))..." node[red] {Texte};")
```

Ou encore pour changer des styles :

```
g:Writeln("\\tikzset{every node/.style={fill=white}}")
```

Dans une présentation beamer, cela peut aussi être utilisé pour insérer des pauses dans un graphique :

```
g:Writeln("\\pause")
```

II Méthodes graphiques

On peut créer des lignes polygonales, des courbes, des chemins, des points, des labels.

1) Lignes polygonales

Dpolyline

Une ligne polygonale est une liste (table) de composantes connexes, et une composante connexe est une liste (table) de complexes qui représentent les affixes des points. Par exemple l'instruction :

```
local Z = luadraw.cpx.Z -- alias
local L = { {Z(-4,0), Z(0,2), Z(1,3)}, {Z(0,0), Z(4,-2), Z(1,-1)} }
```

crée une ligne polygonale à deux composantes dans une variable *L*.

Dessin d'une ligne polygonale. C'est la méthode

g:Dpolyline(L [, close, draw_options, clip])

où *g* désigne le graphique en cours de création. L'argument ⟨*L*⟩ est une ligne polygonale, ⟨*close*⟩ un argument facultatif qui vaut `true` ou `false` indiquant si la ligne doit être refermée ou non (`false` par défaut), ⟨*draw_options*⟩ est une chaîne de caractères qui sera passée directement à l'instruction *\draw* dans l'export (vide par défaut). L'argument ⟨*clip*⟩ doit contenir ou bien `nil` (valeur par défaut) ou bien une table de la forme `{x1,x2,y1,y2}`, dans le premier cas la ligne est clippée par la fenêtre 2D courante **après** sa transformation par la matrice 2D du graphe, dans le second cas la ligne est clippée par la fenêtre $[x_1;x_2]\times[y_1;y_2]$ **avant** d'être transformée par la matrice du graphe.

Options de dessin d'une ligne polygonale

On peut passer des options de dessin directement à l'instruction *\draw* dans l'export, mais elles auront un effet local uniquement. Il est possible de modifier ces options de manière globale avec la méthode

g:Lineoptions(style [, color, width, arrows])

lorsqu'un des arguments est égal à `nil`, c'est sa valeur par défaut qui est utilisée.

- ⟨*color*⟩ est une chaîne de caractères représentant une couleur connue de TikZ (`"black"` par défaut),
- ⟨*style*⟩ est une chaîne de caractères représentant le type de ligne à dessiner, ce style peut être :
  - `"noline"` : trait non dessiné,
  - `"solid"` : trait plein, valeur par défaut,
  - `"dotted"` : trait pointillé,
  - `"dashed"` : tirets,
  - style personnalisé : l'argument ⟨*style*⟩ peut être une chaîne de la forme (exemple) `"{2.5pt}{2pt}"` ce qui signifie : un trait de 2.5pt suivi d'un espace de 2pt, le nombre de valeurs peut être supérieur à 2, ex : `"{2.5pt}{2pt}{1pt}{2pt}"` (succession de on, off).
- ⟨*width*⟩ est un nombre représentant l'épaisseur de ligne exprimée en dixième de points, par exemple 8 pour une épaisseur réelle de 0.8pt (valeur de 4 par défaut),
- ⟨*arrows*⟩ est une chaîne qui précise le type de flèche qui sera dessiné, cela peut être :
  - `"-"` qui signifie pas de flèche (valeur par défaut),
  - `"->"` qui signifie une flèche à la fin,
  - `"<-"` qui signifie une flèche au début,
  - `"<->"` qui signifie une flèche à chaque bout.

  ATTENTION : la flèche n'est pas dessinée lorsque l'argument ⟨*close*⟩ vaut `true`.

On peut modifier les options individuellement avec les méthodes :

- **g:Linecolor(color)**,
- **g:Linestyle(style)**,
- **g:Linewidth(width)**,
- **g:Arrows(arrows)**,
- plus les méthodes suivantes :
  - **g:Lineopacity(opacity)** qui règle l'opacité du tracé de la ligne, l'argument ⟨*opacity*⟩ doit être une valeur entre 0 (totalement transparent) et 1 (totalement opaque), par défaut la valeur est de 1.
  - **g:Linecap(style)**, pour jouer sur les extrémités de la ligne, l'argument ⟨*style*⟩ est une chaîne qui peut valoir :
    - `"butt"` (bout droit au point d'arrêt, valeur par défaut),
    - `"round"` (bout arrondi en demi-cercle),
    - `"square"` (bout « arrondi » en carré).
  - **g:Linejoin(style)**, pour jouer sur la jointure entre segments, l'argument ⟨*style*⟩ est une chaîne qui peut valoir :
    - `"miter"` (coin pointu, valeur par défaut),
    - `"round"` (ou coin arrondi),
    - `"bevel"` (coin coupé).

Options de remplissage d'une ligne polygonale. C'est la méthode

g:Filloptions(style [, color, opacity, evenodd])

(qui utilise la librairie *patterns* de TikZ, celle-ci est chargée avec le paquet). Lorsqu'un des arguments vaut `nil`, c'est sa valeur par défaut qui est utilisée :

- ⟨*color*⟩ est une chaîne de caractères représentant une couleur connue de TikZ (`"black"` par défaut).
- ⟨*style*⟩ est une chaîne de caractères représentant le type de remplissage, ce style peut être :
  - `"none"` : pas de remplissage, c'est la valeur par défaut,
  - `"full"` : remplissage plein,
  - `"bdiag"` : hachures descendantes de gauche à droite,
  - `"fdiag"` : hachures montantes de gauche à droite,
  - `"horizontal"` : hachures horizontales,
  - `"vertical"` : hachures verticales,
  - `"hvcross"` : hachures horizontales et verticales,
  - `"diagcross"` : diagonales descendantes et montantes,
  - `"gradient"` : dans ce cas le remplissage se fait avec un gradient défini avec la méthode **g:Gradstyle(chaîne)**, ce style est passé tel quel à l'instruction *\draw*. Par défaut la chaîne définissant le style de gradient est :
    `"left color = white,right color = red"`
  - tout autre style connu de la librairie *patterns* est également possible.
- ⟨*opacity*⟩ : nombre entre 0 et 1 (1 par défaut).
- ⟨*evenodd*⟩ : booléen indiquant si l'option *even odd rule* doit être utilisée pour le remplissage (`false` par défaut).

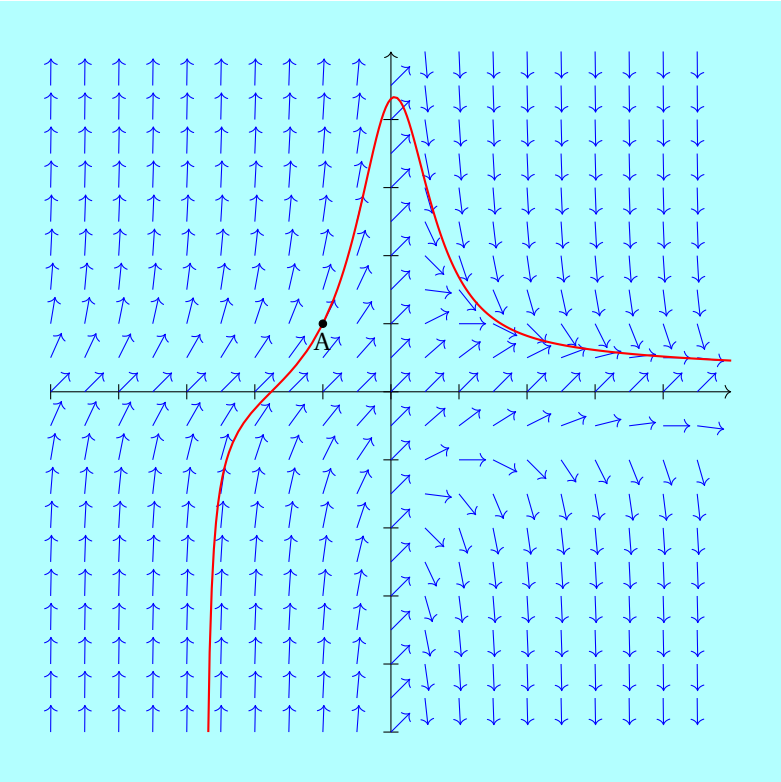
On peut modifier certaines options individuellement avec les méthodes :

- **g:Fillopacity(opacity)**,
- **g:Filleo(evenodd)**.

```
\begin{luadraw}{name=champ}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z
local g = ld.graph:new{window={-5,5,-5,5},bg="Cyan!30",size={10,10}}
local f = function(x,y) -- éq. diff. y'= 1-x*y^2=f(x,y)
    return 1-x*y^2
end
local A = Z(-1,1) -- A = -1+i
local deltaX, deltaY, long = 0.5, 0.5, 0.4
local champ = function(f)
    local vecteurs, v = {}
    for y = g:Yinf(), g:Ysup(), deltaY do
        for x = g:Xinf(), g:Xsup(), deltaX do
            v = Z(1,f(x,y)) -- coordonnées 1 et f(x,y)
            v = v/cpx.abs(v)*long -- normalisation de v
            table.insert(vecteurs, {Z(x,y), Z(x,y)+v} )
        end
    end
    return vecteurs -- vecteurs est une ligne polygonale
end

g:Daxes( {0,1,1}, {labelpos={"none","none"}, arrows="->"} )
g:Dpolyline( champ(f), "->,blue")
g:Dodesolve(f, A.re, A.im, {t={-2.75,5},draw_options="red,line width=0.8pt"})
g:Dlabeldot("$A$", A, {pos="S"})
g:Show()
\end{luadraw}
```

FIGURE 2 : Champ de vecteurs, courbe intégrale de $y' = 1 - xy^2$



Dpolyline2

La méthode **g:Dpolyline2(L [, close, draw_options])**, fait la même chose que la méthode **g:Dpolyline()** à deux différences près :

1. La liste ⟨*L*⟩, en plus des nombres complexes, peut contenir des nœuds TikZ sous forme de chaînes, comme par exemple : `"(A.south)"`. Cependant, ces nœuds ne seront réellement déterminés qu'à la compilation du code TikZ, donc après l'exécution de la méthode **g:Show()**, par conséquent *luadraw* ne connaît pas les coordonnées numériques de ces nœuds, ce qui rend l'intérêt de cette méthode assez limité.
2. Pour les raisons évoquées ci-dessus, la liste ⟨*L*⟩ ne peut pas être clippée par *luadraw*.

2) Segments et droites

Un segment est une liste (table) de deux complexes représentant les extrémités. Une droite est une liste (table) de deux complexes, le premier représente un point de la droite, et le second un vecteur directeur de la droite (celui-ci doit être non nul).

Dangle

- La méthode **g:Dangle(B, A, C [, r, draw_options])** dessine l'angle BAC avec un parallélogramme (deux côtés seulement sont dessinés), l'argument facultatif ⟨*r*⟩ précise la longueur d'un côté (0.25 par défaut). L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.angleD(B, A, C, r)** renvoie la liste des points de cet angle.

Dbissec

- La méthode **g:Dbissec(B, A, C [, interior, draw_options, scale])** dessine une bissectrice de l'angle géométrique BAC, intérieure si l'argument facultatif ⟨*interior*⟩ vaut `true` (valeur par défaut), extérieure sinon. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*. L'argument ⟨*scale*⟩ (qui vaut 1 par défaut) est soit un nombre (pourcentage) permettant de faire varier la longueur du segment affiché (à partir de son milieu), soit une table de deux nombres (pourcentages) {scaleA, scaleB} permettant de faire varier la longueur du segment affiché à chacune de ses extrémités.

- La fonction **ld.bissec(B, A, C, interior)** renvoie cette bissectrice sous forme d'une liste {A, u} où u est un vecteur directeur de la droite.

Dhline

La méthode **g:Dhline(d [, draw_options, scale])** dessine une demi-droite, l'argument ⟨d⟩ doit être une liste de deux nombres complexes {A, B}, c'est la demi-droite [A, B) qui est dessinée.

Variante : **g:Dhline(A, B [, draw_options, scale])**. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *draw*. L'argument ⟨*scale*⟩ (qui vaut 1 par défaut) est soit un nombre (pourcentage) permettant de faire varier la longueur du segment affiché (à partir de son milieu), soit une table de deux nombres (pourcentages) {scaleA, scaleB} permettant de faire varier la longueur du segment affiché à chacune de ses extrémités.

Dline

La méthode **g:Dline(d [, draw_options, scale])** trace la droite ⟨d⟩, celle-ci est une liste du type {A, u} où A représente un point de la droite (un nombre complexe) et u un vecteur directeur (un nombre complexe non nul).

Variante : la méthode **g:Dline(A, B [, draw_options, scale])** trace la droite passant par les points ⟨A⟩ et ⟨B⟩ (deux nombres complexes). L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *draw*. L'argument ⟨*scale*⟩ (qui vaut 1 par défaut) est soit un nombre (pourcentage) permettant de faire varier la longueur du segment affiché (à partir de son milieu), soit une table de deux nombres (pourcentages) {scaleA, scaleB} permettant de faire varier la longueur du segment affiché à chacune de ses extrémités.

DlineEq

- La méthode **g:DlineEq(a, b, c [, draw_options, scale])** dessine la droite d'équation $ax + by + c = 0$. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *draw*. L'argument ⟨*scale*⟩ (qui vaut 1 par défaut) est soit un nombre (pourcentage) permettant de faire varier la longueur du segment affiché (à partir de son milieu), soit une table de deux nombres (pourcentages) {scaleA, scaleB} permettant de faire varier la longueur du segment affiché à chacune de ses extrémités.
- La fonction **ld.lineEq(a, b, c [, inequality])** renvoie la droite d'équation $ax + by + c = 0$ sous la forme d'une liste {A, u} où A est un point de la droite et u un vecteur directeur. L'argument optionnel ⟨*inequality*⟩ (`nil` par défaut) peut aussi prendre comme valeur `'<'` ou `'>'`, le vecteur directeur u est alors choisi de telle sorte que le demi-plan vérifiant l'inégalité $ax + by + c < 0$ (ou $ax + by + c > 0$ dans l'autre cas) soit à gauche de u (dans le sens trigonométrique).

Dmarkarc

La méthode **g:Dmarkarc(b, a, c, r, n [,long,espace,draw_options])** permet de marquer l'arc de cercle de centre ⟨a⟩, de rayon ⟨r⟩, allant de ⟨b⟩ à ⟨c⟩, avec ⟨n⟩ petits segments. Par défaut la longueur (argument ⟨*long*⟩) est de 0.25, et l'espacement (argument ⟨*espace*⟩) et de 0.0625. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *draw*.

Dmarkseg

La méthode **g:Dmarkseg(a, b, n, [long, espace, angle, draw_options])** permet de marquer le segment défini par ⟨a⟩ et ⟨b⟩ avec ⟨n⟩ petits segments penchés de ⟨*angle*⟩ degrés (45° par défaut). Par défaut la longueur (argument ⟨*long*⟩) est de 0.25, et l'espacement (argument ⟨*espace*⟩) et de 0.125. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *draw*.

Dmed

- La méthode **g:Dmed(A, B [, draw_options, scale])** trace la médiatrice du segment [A; B].

  Variante : **g:Dmed(seg [, draw_options, scale])** où segment est une liste de deux points représentant le segment. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *draw*. L'argument ⟨*scale*⟩ (qui vaut 1 par défaut) est soit un nombre (pourcentage) permettant de faire varier la longueur du

segment affiché (à partir de son milieu), soit une table de deux nombres (pourcentages) {scaleA, scaleB} permettant de faire varier la longueur du segment affiché à chacune de ses extrémités.

- La fonction **ld.med(A, B)** (ou **ld.med(seg)**) renvoie la médiatrice du segment [A;B] sous la forme d'une liste {C, u} où C est un point de la droite et u un vecteur directeur.

Dparallel

- La méthode **g:Dparallel(d, A [, draw_options, scale])** trace la parallèle à ⟨d⟩ passant par ⟨A⟩. L'argument ⟨d⟩ peut être soit une droite (une liste constituée d'un point et un vecteur directeur) soit un vecteur non nul. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *draw*. L'argument ⟨*scale*⟩ (qui vaut 1 par défaut) est soit un nombre (pourcentage) permettant de faire varier la longueur du segment affiché (à partir de son milieu), soit une table de deux nombres (pourcentages) {scaleA, scaleB} permettant de faire varier la longueur du segment affiché à chacune de ses extrémités.
- La fonction **ld.parallel(d, A)** renvoie la parallèle à ⟨d⟩ passant par ⟨A⟩ sous la forme d'une liste {B, u} où B est un point de la droite et u un vecteur directeur.

Dperp

- La méthode **g:Dperp(d, A [, draw_options, scale])** trace la perpendiculaire à ⟨d⟩ passant par ⟨A⟩. L'argument ⟨d⟩ peut être soit une droite (une liste constituée d'un point et un vecteur directeur) soit un vecteur non nul. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *draw*. L'argument ⟨*scale*⟩ (qui vaut 1 par défaut) est soit un nombre (pourcentage) permettant de faire varier la longueur du segment affiché (à partir de son milieu), soit une table de deux nombres (pourcentages) {scaleA, scaleB} permettant de faire varier la longueur du segment affiché à chacune de ses extrémités.
- La fonction **ld.perp(d, A)** renvoie la perpendiculaire à ⟨d⟩ passant par ⟨A⟩ sous la forme d'une liste {B, u} où B est un point de la droite et u un vecteur directeur.

Dseg

La méthode **g:Dseg(seg [, scale, draw_options])** dessine le segment défini par l'argument ⟨*seg*⟩ qui doit être une liste de deux complexes. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *draw*. L'argument ⟨*scale*⟩ (qui vaut 1 par défaut) est soit un nombre (pourcentage) permettant de faire varier la longueur du segment affiché (à partir de son milieu), soit une table de deux nombres (pourcentages) {scaleA, scaleB} permettant de faire varier la longueur du segment affiché à chacune de ses extrémités.

Dtangent

- La méthode **g:Dtangent(p, t0 [, long, draw_options])** dessine la tangente à la courbe paramétrée par ⟨p⟩ : $t \mapsto p(t)$ (à valeurs complexes), au point de paramètre ⟨*t0*⟩. Si l'argument ⟨*long*⟩ vaut `nil` (valeur par défaut) alors c'est la droite entière qui est dessinée. Si c'est un nombre alors c'est un segment de longueur ⟨*long*⟩ centré sur le point où est effectué le calcul, noté A. Si c'est une table de deux nombres $\{l_1, l_2\}$, alors c'est un segment dont la première partie (jusqu'au point A) a pour longueur l_1, et la deuxième partie (à partir du point A) a pour longueur l_2. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *draw*.
- La fonction **ld.tangent(p, t0 [, long])** renvoie soit la droite, soit un segment (suivant que ⟨*long*⟩ vaut `nil` ou pas).

DtangentC

- La méthode **g:DtangentC(f, x0 [, long, draw_options])** dessine la tangente à la courbe cartésienne d'équation $y = f(x)$, au point d'abscisse ⟨*x0*⟩. Si l'argument ⟨*long*⟩ vaut `nil` (valeur par défaut) alors c'est la droite entière qui est dessinée. Si c'est un nombre alors c'est un segment de longueur ⟨*long*⟩ centré sur le point où est effectué le calcul, noté A. Si c'est une table de deux nombres $\{l_1, l_2\}$, alors c'est un segment dont la première partie (jusqu'au point A) a pour longueur l_1, et la deuxième partie (à partir du point A) a pour longueur l_2. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *draw*.
- La fonction **ld.tangentC(f, x0 [, long])** renvoie soit la droite, soit un segment (suivant que ⟨*long*⟩ vaut `nil` ou pas).

DtangentI

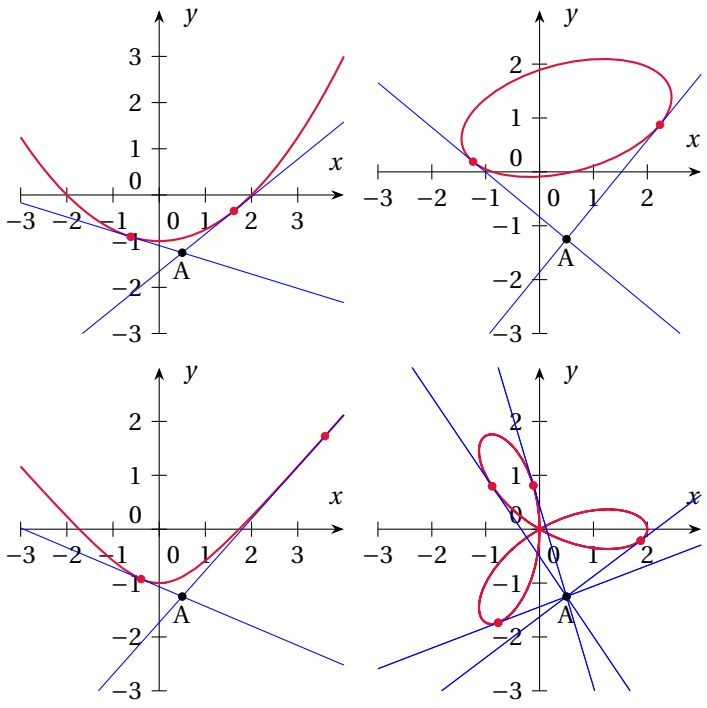
- La méthode **g:DtangentI(f, x0, y0 [, long, draw_options])** dessine la tangente à la courbe implicite d'équation $f(x,y)=0$, au point ⟨*(x0,y0)*⟩ supposé sur la courbe. Si l'argument ⟨*long*⟩ vaut `nil` (valeur par défaut) alors c'est la droite entière qui est dessinée. Si c'est un nombre alors c'est un segment de longueur ⟨*long*⟩ centré sur le point où est effectué le calcul, noté A. Si c'est une table de deux nombres $\{l_1, l_2\}$, alors c'est un segment dont la première partie (jusqu'au point A) a pour longueur l_1, et la deuxième partie (à partir du point A) a pour longueur l_2. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.tangentI(f, x0, y0 [, long])** renvoie soit la droite, soit un segment (suivant que ⟨*long*⟩ vaut `nil` ou pas).

Dtangent_from

- La méthode **g:Dtangent_from(A, p, t1, t2 [, dp, draw_options, out, scale])** dessine la ou les tangentes à la courbe paramétrée par $\langle p\rangle : t \mapsto p(t)$ (à valeurs complexes) issue(s) du point ⟨*A*⟩ (nombre complexe). Les arguments ⟨*t1*⟩ et ⟨*t2*⟩ représentent les bornes de l'intervalle de recherche. L'argument optionnel ⟨*dp*⟩ est une fonction représentant la dérivée de la fonction ⟨*p*⟩, par défaut cet argument vaut `nil` et la dérivée de ⟨*p*⟩ est remplacée par une approximation. L'argument optionnel ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*. L'argument optionnel ⟨*out*⟩, s'il est utilisé, doit être le nom d'une variable, celle-ci doit être une table, elle contiendra à la fin de l'exécution les points de la courbe pour lesquels la tangente passe par ⟨*A*⟩. L'argument ⟨*scale*⟩ (qui vaut 1 par défaut) est soit un nombre (pourcentage) permettant de faire varier la longueur du segment affiché (à partir de son milieu), soit une table de deux nombres (pourcentages) {scaleA, scaleB} permettant de faire varier la longueur du segment affiché à chacune de ses extrémités.
- La fonction **ld.tangent_from(A, p, t1, t2 [, dp])** renvoie la liste les points de la courbe paramétrée par ⟨*p*⟩, sur l'intervalle $[t_1;t_2]$ pour lesquels la tangente passe par ⟨*A*⟩ (nombre complexe). S'il n'y en a pas, la fonction renvoie une liste vide.

```
\begin{luadraw}{name=tangent_from}
local ld = luadraw
local cpx = ld.cpx
local i, cos, sin, pi = cpx.I, math.cos, math.sin, math.pi
local g = ld.graph:new{window={-5,5,-5,5}, size={10,10}}
local p1 = function(t) return t+i*(t^2/4-1) end -- parabole
local p2 = function(t) return 1/2+i+ld.rotate(2*cos(t)+i*sin(t),15) end -- ellipse
local p3 = function(t) return math.sinh(t)+i*math.cosh(t)-i*2 end -- branche d'hyperbole
local p4 = function(t) return 2*cos(3*t)*cpx.exp(i*t) end -- autre
local P = 0.5-1.25*i
local draw = function(p,t1,t2)
    local axis_style = { arrows='-Stealth', legend={'$x$','$y$'} }
    local S = {}
    g:Daxes({0, 1, 1}, axis_style)
    g:Dparametric(p,{t={t1,t2}, draw_options="line width=0.8pt, Crimson"})
    g:Dtangent_from(P,p,t1,t2,"blue",S)
    g:Ddots(S,"Crimson"); g:Ddots(P); g:Dlabel("$A$",P,{pos="S"})
end
g:Saveattr(); g:Viewport(-5,-0.25,0.25,5); g:Coordsystem(-3,4,-3,4); draw(p1,-4,4); g:Restoreattr()
g:Saveattr(); g:Viewport(0.25,5,0.25,5); g:Coordsystem(-3,3,-3,3); draw(p2,-pi,pi); g:Restoreattr()
g:Saveattr(); g:Viewport(-5,-0.25,-5,-0.25); g:Coordsystem(-3,4,-3,3); draw(p3,-4,4); g:Restoreattr()
g:Saveattr(); g:Viewport(0.25,5,-5,-0.25); g:Coordsystem(-3,3,-3,3); draw(p4,-pi,pi); g:Restoreattr()
g:Show()
\end{luadraw}
```

FIGURE 3 : Tangentes issues d'un point



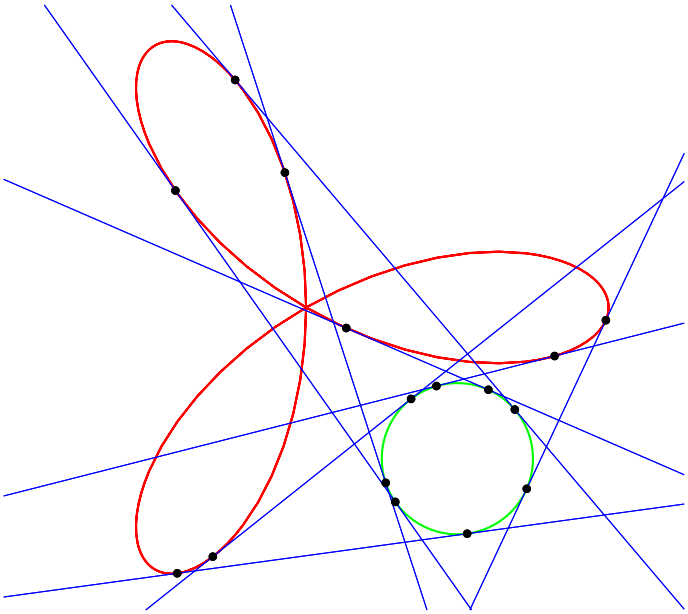
Dcommon_tangent

- La méthode **g:Dcommon_tangent(p1, p2, options)** dessine la ou les tangentes communes aux courbes paramétrées par $\langle p1\rangle : t \mapsto p(t)$ (à valeurs complexes) et $\langle p2\rangle : t \mapsto p(t)$ (à valeurs complexes). L'argument $\langle options\rangle$ est une table dont les champs sont :
  - `t1={-5,5}` : intervalle de recherche pour le paramétrage $\langle p1\rangle$.
  - `t2={-5,5}` : intervalle de recherche pour le paramétrage $\langle p2\rangle$.
  - `grid={50,50}` : table représentant le nombre de subdivisions pour $\langle p1\rangle$ et pour $\langle p2\rangle$.
  - `draw_options=""` : options de dessin pour les droites (chaîne transmise directement à l'instruction *\draw*).
  - `scale=1` : L'argument $\langle scale\rangle$ (qui vaut 1 par défaut) est soit un nombre (pourcentage) permettant de faire varier la longueur du segment affiché (à partir de son milieu), soit une table de deux nombres (pourcentages) {scaleA, scaleB} permettant de faire varier la longueur du segment affiché à chacune de ses extrémités.
- La fonction **ld.common_tangent(p1, u1,u2, p2, v1, v2, grid)** renvoie une liste de valeurs de la forme $\{t_1, t_2\}$ telle que la droite joignant les points $p_1(t_1)$ et $p_2(t_2)$ est une tangente commune aux deux courbes. S'il n'y en a pas, la fonction ne renvoie rien.

Attention : pour cette méthode les calculs peuvent être parfois un peu longs.

```
\begin{luadraw}{name=common_tangent}
local ld = luadraw
local cpx = ld.cpx
local i, cos, sin, pi = cpx.I, math.cos, math.sin, math.pi
local g = ld.graph:new{window={-4,5,-4,4}, size={10,10}}
local p1 = function(t) return  2-2*i+ cos(t)+i*sin(t) end -- ellipse
local p2 = function(t) return 4*cos(3*t)*cpx.exp(i*t) end -- autre
local dots = {}
g:Dparametric(p1,{t={-pi,pi}, draw_options="green,thick"})
g:Dparametric(p2,{t={-pi,pi}, draw_options="red,thick"})
g:Dcommon_tangent(p1,p2,{t1={-pi,pi}, t2={-pi,pi}, draw_options='blue', out=dots})
g:Ddots(dots)
g:Show()
\end{luadraw}
```

FIGURE 4 : Tangentes communes



Dparallel_tangent

- La méthode **g:Dparallel_tangent(p, t0, options)** dessine la ou les tangentes à la courbe paramétrées par $\langle p\rangle : t \mapsto p(t)$ (à valeurs complexes), parallèle à la tangente au point de paramètre $\langle t0\rangle$ **si celui-ci est un nombre réel**. Lorsque $\langle t0\rangle$ est un nombre complexe représentant un vecteur non nul, la méthode dessine les tangentes ayant pour direction ce vecteur. L'argument $\langle options\rangle$ est une table dont les champs sont :
  - `t={-5,5}` : intervalle de recherche.
  - `draw_options=""` : options de dessin pour les droites (chaîne transmise directement à l'instruction *\draw*).
  - `len=nil` : identique à l'argument $\langle long\rangle$ de la méthode **g:Dtangent()**.
  - `dp=nil` : permet de fournir éventuellement la fonction dérivée de $\langle p\rangle$. Par défaut, la dérivée est une approximation.
    `out=nil` : lorsque cette option et une variable représentant une table, alors les solutions pour le paramètres t seront ajoutées à cette table.
- La fonction **ld.parallel_tangent(p, t0, t1, t2 [,dp])** renvoie une liste de valeurs de t de l'intervalle $[t_1; t_2]$ telles que les tangentes à la courbe paramétrée par $\langle p\rangle$ aux points de paramètre t, soient parallèles à la tangente au point de paramètre $\langle t0\rangle$, ou de direction la vecteur $\langle t0\rangle$. S'il n'y en a pas, la fonction ne renvoie rien.

Dnormal

- La méthode **g:Dnormal(p, t0 [, long, draw_options])** dessine la normale à la courbe paramétrée par $\langle p\rangle : t \mapsto p(t)$ (à valeurs complexes), au point de paramètre $\langle t0\rangle$. Si l'argument $\langle long\rangle$ vaut `nil` (valeur par défaut) alors c'est la droite entière qui est dessinée. Si c'est un nombre alors c'est un segment de longueur $\langle long\rangle$ centré sur le point où est effectué le calcul, noté A. Si c'est une table de deux nombres $\{l_1, l_2\}$, alors c'est un segment dont la première partie (jusqu'au point A) a pour longueur l_1, et la deuxième partie (à partir du point A) a pour longueur l_2. L'argument $\langle draw_options\rangle$ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.normal(p, t0 [, long])** renvoie soit la droite, soit un segment (suivant que *long* vaut `nil` ou pas).

DnormalC

- La méthode **g:DnormalC(f, x0 [, long, draw_options])** dessine la normale à la courbe cartésienne d'équation $y = f(x)$, au point d'abscisse $\langle x0\rangle$. Si l'argument $\langle long\rangle$ vaut `nil` (valeur par défaut) alors c'est la droite entière qui est dessinée. Si c'est un nombre alors c'est un segment de longueur $\langle long\rangle$ centré sur le point où est effectué le calcul, noté A. Si c'est une table de deux nombres $\{l_1, l_2\}$, alors c'est un segment dont la première partie (jusqu'au point A) a pour longueur l_1, et la deuxième partie (à partir du point A) a pour longueur l_2. L'argument $\langle draw_options\rangle$ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

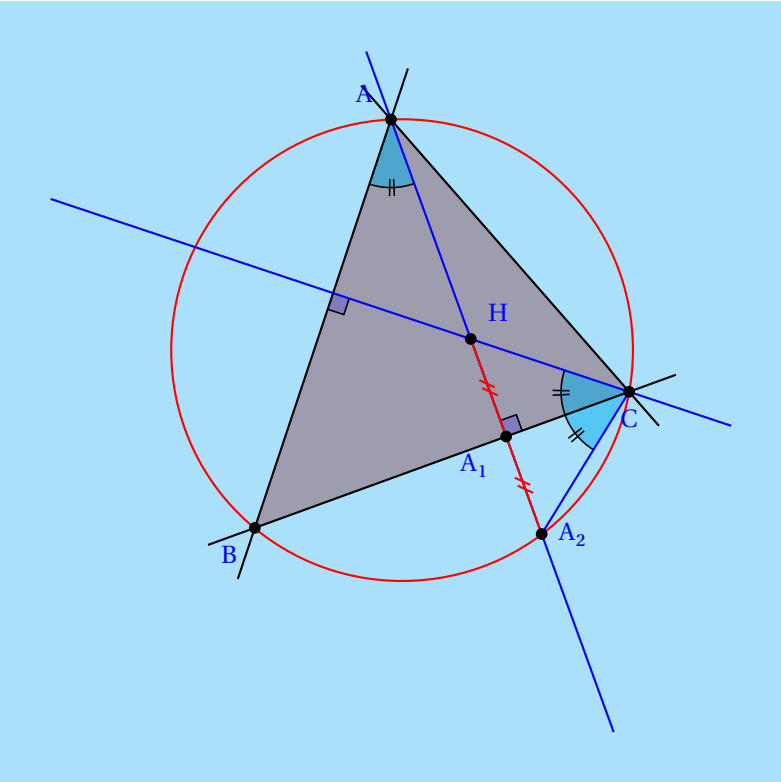
- La fonction **ld.normalC(f, x0 [, long])** renvoie soit la droite, soit un segment (suivant que ⟨*long*⟩ vaut `nil` ou pas).

DnormalI

- La méthode **g:DnormalI(f, x0, y0 [, long, draw_options])** dessine la normale à la courbe implicite d'équation $f(x,y)=0$, au point ⟨*(x0,y0)*⟩ supposé sur la courbe. Si l'argument ⟨*long*⟩ vaut `nil` (valeur par défaut) alors c'est la droite entière qui est dessinée. Si c'est un nombre alors c'est un segment de longueur ⟨*long*⟩ centré sur le point où est effectué le calcul, noté A. Si c'est une table de deux nombres $\{l_1, l_2\}$, alors c'est un segment dont la première partie (jusqu'au point A) a pour longueur l_1, et la deuxième partie (à partir du point A) a pour longueur l_2. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.normalI(f, x0, y0 [, long])** renvoie soit la droite, soit un segment (suivant que ⟨*long*⟩ vaut `nil` ou pas).

```
\begin{luadraw}{name=orthocentre}
local ld = luadraw
local cpx = ld.cpx
local g = ld.graph:new{window={-5,5,-5,5},bg="cyan!30",size={10,10}}
local i = cpx.I
local A, B, C = 4*i, -2-2*i, 3.5
local h1, h2 = ld.perp({B,C-B},A), ld.perp({A,B-A},C) -- hauteurs
local A1, F = ld.proj(A,{B,C-B}), ld.proj(C,{A,B-A}) -- projetés
local H = ld.interD(h1,h2) -- orthocentre
local A2 = 2*A1-H -- symétrique de H par rapport à BC
g:Dpolyline({A,B,C},true, "draw=none,fill=Maroon,fill opacity=0.3") -- fond du triangle
g:Linewidth(6); g:Filloptions("full", "blue", 0.2)
g:Dangle(C,A1,A,0.25); g:Dangle(B,F,C,0.25) -- angles droits
g:Linecolor("black"); g:Filloptions("full","cyan",0.5)
g:Darc(H,C,A2,1); g:Darc(B,A,A1,1) -- arcs
g:Filloptions("none","black",1) -- on rétablit l'opacité à 1
g:Dmarkarc(H,C,A1,1,2); g:Dmarkarc(A1,C,A2,1,2) -- marques
g:Dmarkarc(B,A,H,1,2)
g:Linewidth(8); g:Linecolor("black")
g:Dseg({A,B},1.25); g:Dseg({C,B},1.25); g:Dseg({A,C},1.25) -- côtés
g:Linecolor("red"); g:Dcircle(A,B,C) -- cercle
g:Linecolor("blue"); g:Dline(h1); g:Dline(h2) -- hauteurs
g:Dseg({A2,C}); g:Linecolor("red"); g:Dseg({H,A2}) -- segments
g:Dmarkseg(H,A1,2); g:Dmarkseg(A1,A2,2) -- marques
g:Labelcolor("blue") -- pour les labels
g:Dlabel("$A$",A, {pos="NW",dist=0.1}, "$B$",B, {pos="SW"}, "$A_2$",A2,{pos="E"},
"$C$", C, {pos="S" }, "$H$", H, {pos="NE"}, "$A_1$", A1, {pos="SW"})
g:Linecolor("black"); g:Filloptions("full"); g:Ddots({A,B,C,H,A1,A2}) -- dessin des points
g:Show()
\end{luadraw}
```

FIGURE 5 : Symétrique de l'orthocentre



3) Figures géométriques

Darc

- La méthode **g:Darc(B, A, C, r [, sens, draw_options])** dessine un arc de cercle de centre ⟨*A*⟩ (complexe), de rayon ⟨*r*⟩, allant de ⟨*B*⟩ (complexe) vers ⟨*C*⟩ (complexe) dans le sens trigonométrique si l'argument ⟨*sens*⟩ vaut 1 (valeur par défaut), le sens inverse sinon. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.arc(B, A, C, r, sens)** renvoie la liste des points de cet arc (ligne polygonale).
- La fonction **ld.arcb(B, A, C, r, sens)** renvoie cet arc sous forme d'un chemin (voir *Dpath* page 40) utilisant des courbes de Bézier.

Dcircle

- La méthode **g:Dcircle(c, r [, d, draw_options])** trace un cercle. Lorsque l'argument ⟨*d*⟩ vaut `nil`, c'est le cercle de centre ⟨*c*⟩ (complexe) et de rayon ⟨*r*⟩, lorsque ⟨*d*⟩ est précisé (complexe) alors c'est le cercle passant par les points d'affixe ⟨*c*⟩, ⟨*r*⟩ et ⟨*d*⟩. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*. Autre syntaxe possible : **g:Dcircle(C [, draw_options])** où ⟨*C*⟩= {*c*, *r*[, *d*]}.
- La fonction **ld.circle(c, r [, d])** renvoie la liste des points de ce cercle (ligne polygonale).
- La fonction **ld.circle({c, r [, d]}, nbdots)** renvoie la liste des points de ce cercle (ligne polygonale) avec ⟨*nbdots*⟩ points.
- La fonction **ld.circleb(c, r [, d])** renvoie ce cercle sous forme d'un chemin (voir *Dpath* page 40) utilisant des courbes de Bézier.

Dellipse

- La méthode **g:Dellipse(c, rx, ry [, inclin, draw_options])** dessine l'ellipse de centre ⟨*c*⟩ (complexe), les arguments ⟨*rx*⟩ et ⟨*ry*⟩ précisent les deux rayons (sur x et sur y), l'argument facultatif ⟨*inclin*⟩ est un angle en degrés qui indique l'inclinaison de l'ellipse par rapport à l'axe Ox (angle nul par défaut). L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.ellipse(c, rx, ry [, inclin])** renvoie la liste des points de cette ellipse (ligne polygonale).

- La fonction **ld.ellipse({c, rx, ry [, inclin]}, nbdots)** renvoie la liste des points de cette ellipse (ligne polygonale) avec ⟨*nbdots*⟩ points.
- La fonction **ld.ellipseb(c, rx, ry [, inclin])** renvoie cette ellipse sous forme d'un chemin (voir *Dpath* page 40) utilisant des courbes de Bézier.

Dellipticarc

- La méthode **g:Dellipticarc(B, A, C, rx, ry [, sens, inclin, draw_options])** dessine un arc d'ellipse de centre ⟨*A*⟩ (complexe) de rayons ⟨*rx*⟩ et ⟨*ry*⟩, faisant un angle égal à ⟨*inclin*⟩ par rapport à l'axe O*x* (angle nul par défaut), allant de ⟨*B*⟩ (complexe) vers ⟨*C*⟩ (complexe) dans le sens trigonométrique si l'argument ⟨*sens*⟩ vaut 1 (valeur par défaut), le sens inverse sinon. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.ellipticarc(B, A, C, rx, ry [, sens, inclin])** renvoie la liste des points de cet arc (ligne polygonale).
- La fonction **ld.ellipticarcb(B, A, C, rx, ry [, sens, inclin])** renvoie cet arc sous forme d'un chemin (voir *Dpath* page 40) utilisant des courbes de Bézier.

Dparallelogram

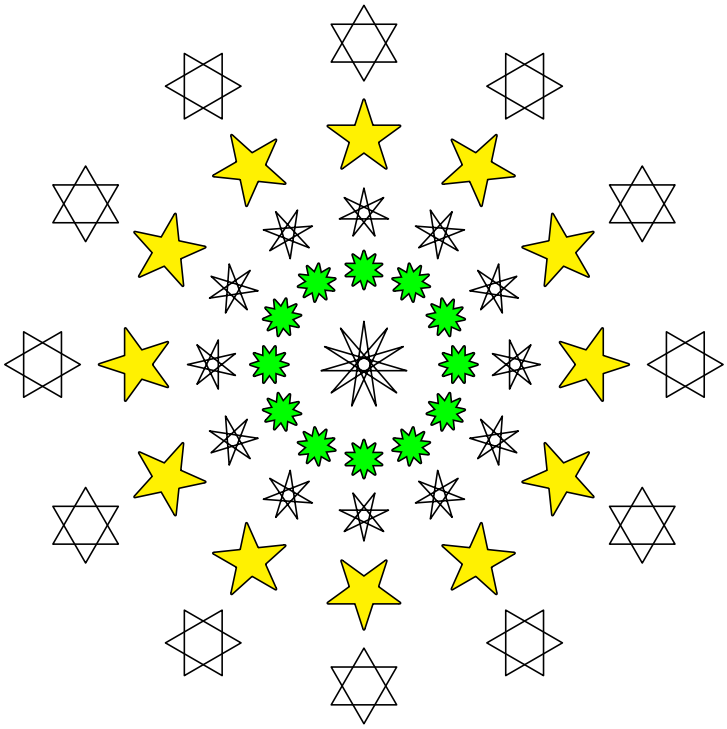
- La méthode **g:Dparallelogram(a, u, v [, draw_options])** où ⟨*a*⟩, ⟨*u*⟩, ⟨*v*⟩ désigne trois nombres complexes désignant un sommet et deux vecteurs, dessine le parallélogramme de sommets $\{a, a+u, a+u+v, a+v\}$. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.parallelogram(a, u, v)** renvoie la liste des sommets de ce parallélogramme.

Dpolyreg

- La méthode **g:Dpolyreg(sommet1, sommet2, nbcotes, sens [, draw_options)]** ou **g:Dpolyreg(centre, sommet, nbcotes [, draw_options])** dessine un polygone régulier. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.polyreg(sommet1, sommet2, nbcotes, sens)** et la fonction **ld.polyreg(centre, sommet, nbcotes)**, renvoient la liste des sommets de ce polygone régulier.
- {p/q}-gone régulier (polygone régulier étoilé) : la fonction **ld.pq_gonreg(C,S,type)** renvoie une liste de listes de nombres complexes représentant un polygone étoilé régulier de centre ⟨*C*⟩, passant par le sommet ⟨*S*⟩, et de type ⟨*type*⟩ où ⟨*type*⟩=$\{p, q\}$ avec p et q deux entiers strictement positifs, p est le nombre de sommets et q est le pas (chaque sommet est relié au q^{e} suivant), avec la condition : $2q < p$.

```
\begin{luadraw}{name=pq_gonreg}
local ld = luadraw
local cpx = ld.cpx
local Z, i = cpx.Z, cpx.I
local g = ld.graph:new{ size={10,10}, margin=0 }
g:Linewidth(6)
local pos1, pos2, pos3, pos4 = Z(0,3), Z(0,2), Z(0,1.25), Z(0,4.25)
local star1 = ld.pq_gonreg(pos1, pos1+i/2, {5,2})
local star2 = ld.pq_gonreg(pos2, pos2+i/3, {7,3})
local star3 = ld.pq_gonreg(pos3, pos3+i/4, {10,4})
local star4 = ld.pq_gonreg(pos4, pos4+i/2, {6,2})
for k = 0, 330, 30 do
    g:Dpolyline(star1, true, "line width=1.2"); g:Dpolyline(star1, "draw=none, fill=yellow")
    g:Dpolyline(star2, true)
    g:Dpolyline(star3, true, "line width=1.2"); g:Dpolyline(star3, "draw=none, fill=green")
    g:Dpolyline(star4, true)
    g:Rotate(30)
end
g:Dpolyline( ld.pq_gonreg(0,i/1.75,{11,5}), true)
g:Show()
\end{luadraw}
```

FIGURE 6 : polygone régulier étoilé



Drectangle

- La méthode **g:Drectangle(a, b, c [, draw_options])** dessine le rectangle ayant comme sommets consécutifs ⟨*a*⟩ et ⟨*b*⟩ et dont le côté opposé passe par ⟨*c*⟩. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.rectangle(a, b, c)** renvoie la liste des sommets de ce rectangle.
- La méthode **g:Drectangle(a, b [, draw_options])** dessine le rectangle dont les côtés sont parallèles aux axes et ayant comme sommets opposés ⟨*a*⟩ et ⟨*b*⟩. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.rectangle(a, b)** renvoie la liste des sommets de ce rectangle.

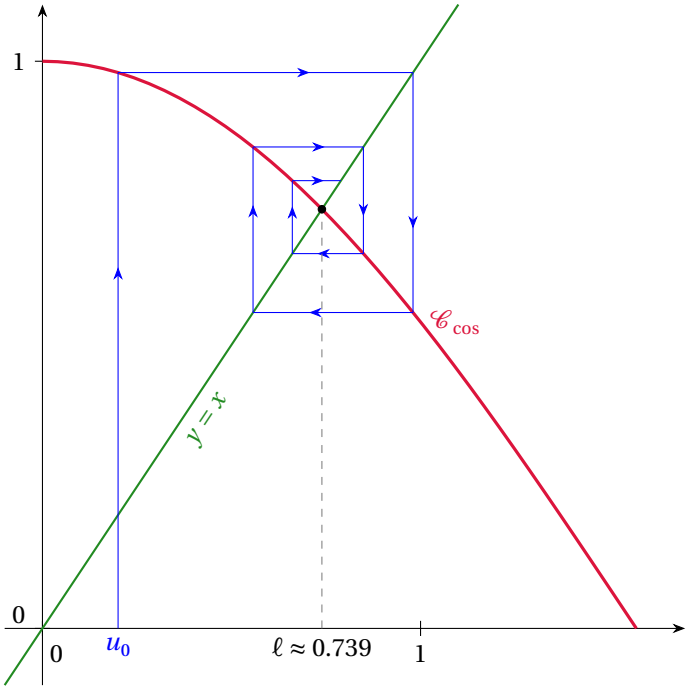
Dsequence

- La méthode **g:Dsequence(f, u0, n [, draw_options])** fait le dessin des "escaliers" de la suite récurrente définie par son premier terme ⟨*u0*⟩ et la relation $u_{k+1} = f(u_k)$. L'argument ⟨*f*⟩ doit être une fonction d'une variable réelle et à valeurs réelles, l'argument ⟨*n*⟩ est le nombre de termes calculés. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.sequence(f, u0, n)** renvoie la liste des points constituant ces "escaliers".

```
\begin{luadraw}{name=sequence}
local ld = luadraw
local cpx = ld.cpx
local g = ld.graph:new{window={-0.1,1.7,-0.1,1.1},size={10,10,0}}
local i, pi, cos = cpx.I, math.pi, math.cos
local f = function(x) return cos(x)-x end
local ell = ld.solve(f,0,pi/2)[1]
local L = ld.sequence(cos,0.2,5) -- u_{n+1}=cos(u_n), u_0=0.2
local seg, z = {}, L[1]
for k = 2, #L do
    table.insert(seg,{z,L[k]})
    z = L[k]
end -- seg est la liste des segments de l'escalier
local styleA = "\\tikzset{->-/.style={decoration={markings, mark="
local styleB = "at position #1 with {\\arrow{Stealth}}}, postaction={decorate}}}"
g:Writeln(styleA..styleB)
```

```
g:Daxes({0,1,1}, {arrows="-Stealth"})
g:DlineEq(1,-1,0,"line width=0.8pt,ForestGreen")
g:Dcartesian(cos, {x={0,pi/2},draw_options="line width=1.2pt,Crimson"})
g:Dpolyline(seg,false,"->-=0.65,blue")
g:Dlabel("$u_0$",0.2,{pos="S",node_options="blue"})
g:Dseg({ell, ell*(1+i)},1,"dashed,gray")
g:Dlabel("$\\ell\\approx"..ld.round(ell,3).."$", ell,{pos="S"})
g:Ddots(ell*(1+i)); g:Labelcolor("Crimson")
g:Dlabel("${\\mathcal C}_{\\cos}$",1+i*cos(1),{pos="E"})
g:Labelcolor("ForestGreen"); g:Labelangle(g:Arg(1+i)*180/pi)
g:Dlabel("$y=x$",0.4+i*0.4,{pos="S",dist=0.1})
g:Show()
\end{luadraw}
```

FIGURE 7 : Suite $u_{n+1} = \cos(u_n)$



La méthode **g:Arg(z)** calcule et renvoie l'argument *réel* du complexe z, c'est à dire son argument (en radians) à l'export dans le repère de TikZ (il faut pour cela appliquer la matrice de transformation du graphe à z, puis faire le changement de repère vers celui de TikZ). Si le repère du graphe est orthonormé et si la matrice de transformation est l'identité alors le résultat est identique à celui de `cpx.arg(z)` (ce n'est pas le cas dans l'exemple ci-dessus).

De même, la méthode **g:Abs(z)** calcule et renvoie le module *réel* du complexe z, c'est à dire son module à l'export dans le repère de TikZ, c'est donc une longueur en centimètres. Si le repère du graphe est orthonormé avec 1cm par unité sur chaque axe, et si la matrice de transformation est une isométrie alors le résultat est identique à celui de `cpx.abs(z)`.

Dsquare

- La méthode **g:Dsquare(a, b [, sens, draw_options])** dessine le carré ayant comme sommets consécutifs ⟨*a*⟩ et ⟨*b*⟩, dans le sens trigonométrique lorsque ⟨*sens*⟩ vaut 1 (valeur par défaut). L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ld.square(a, b, sens)** renvoie la liste des sommets de ce carré.

Dwedge

La méthode **g:Dwedge(B, A, C, r, sens [, draw_options])** dessine un secteur angulaire de centre ⟨*A*⟩ (complexe), de rayon ⟨*r*⟩, allant de ⟨*B*⟩ (complexe) vers ⟨*C*⟩ (complexe) dans le sens trigonométrique si l'argument ⟨*sens*⟩ vaut 1, le sens inverse sinon. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

4) Courbes

Paramétriques : Dparametric

- La fonction **ld.parametric(p, t1, t2 [, nbdots, discont, nbdiv])** fait le calcul des points de la courbe et renvoie une ligne polygonale (une liste de listes de complexes, pas de dessin).
  - L'argument ⟨*p*⟩ est le paramétrage, ce doit être une fonction d'une variable réelle t et à valeurs complexes, par exemple : `local p = function(t) return cpx.exp(t*cpx.I) end`
  - Les arguments ⟨*t1*⟩ et ⟨*t2*⟩ sont obligatoires avec $t_1 < t_2$, ils forment les bornes de l'intervalle pour le paramètre.
  - L'argument ⟨*nbdots*⟩ est facultatif, c'est le nombre de points (minimal) à calculer, il vaut 40 par défaut.
  - L'argument ⟨*discont*⟩ est un booléen facultatif qui indique s'il y a des discontinuités ou non, c'est `false` par défaut.
  - L'argument ⟨*nbdiv*⟩ est un entier positif qui vaut 5 par défaut et indique le nombre de fois que l'intervalle entre deux valeurs consécutives du paramètre peut être coupé en deux (dichotomie) lorsque les points correspondants sont trop éloignés.
- La méthode **g:Dparametric(p, options)** fait le calcul des points et le dessin de la courbe paramétrée par ⟨*p*⟩. L'argument ⟨*options*⟩ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
  - `t={g:Xinf(),g:Xsup()}`, c'est l'intervalle pour le paramètre t (par défaut, c'est tout l'intervalle des abscisses de la fenêtre),
  - `nbdots=40`,
  - `discont=false`,
  - `nbdiv=5`,
  - `draw_options=""`, chaîne qui sera transmise telle quelle à l'instruction *\draw*,
  - `clip=nil`, cette option est soit `nil` (valeur par défaut), soit une table `{x1,x2,y1,y2}`, dans le premier cas la ligne est clippée par la fenêtre 2D courante **après** sa transformation par la matrice 2D du graphe, dans le second cas la ligne est clippée par la fenêtre $[x_1;x_2]\times[y_1;y_2]$ **avant** d'être transformée par la matrice du graphe.

Polaires : Dpolar

- La fonction **ld.polar(rho, t1, t2 [, nbdots, discont, nbdiv])** fait le calcul des points et renvoie une ligne polygonale (pas de dessin). L'argument ⟨*rho*⟩ est le paramétrage polaire de la courbe, ce doit être une fonction d'une variable réelle t et à valeurs réelles, par exemple :
  ```
  local rho = function(t) return 4*math.cos(2*t) end
  ```
  Les autres arguments sont identiques aux courbes paramétrées.
- La méthode **g:Dpolar(rho, options)** fait le calcul des points et le dessin de la courbe polaire paramétrée par *rho*. L'argument ⟨*options*⟩ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
  - `t={-pi,pi}`, c'est l'intervalle pour le paramètre t,
  - `nbdots=40`,
  - `discont=false`,
  - `nbdiv=5`,
  - `draw_options=""`, chaîne qui sera transmise telle quelle à l'instruction *\draw*,
  - `clip=nil`, cette option est soit `nil` (valeur par défaut), soit une table `{x1,x2,y1,y2}`, dans le premier cas la ligne est clippée par la fenêtre 2D courante **après** sa transformation par la matrice 2D du graphe, dans le second cas la ligne est clippée par la fenêtre $[x_1;x_2]\times[y_1;y_2]$ **avant** d'être transformée par la matrice du graphe.

Cartésiennes : Dcartesian

- La fonction **ld.cartesian(f, x1, x2 [, nbdots, discont, nbdiv])** fait le calcul des points de la courbe et renvoie une ligne polygonale (pas de dessin). L'argument ⟨*f*⟩ est la fonction dont on veut la courbe, ce doit être une fonction d'une variable réelle x et à valeurs réelles, par exemple :
  ```
  local f = function(x) return 1+3*math.sin(x)*x end
  ```

Les arguments ⟨*x1*⟩ et ⟨*x2*⟩ sont obligatoires et forment les bornes de l'intervalle pour la variable. Les autres arguments sont identiques aux courbes paramétrées.

- La méthode **g:Dcartesian(f, options)** fait le calcul des points et le dessin de la courbe de ⟨f⟩. L'argument ⟨*options*⟩ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
  - `x={g:Xinf(),g:Xsup()}`, c'est l'intervalle pour le paramètre x (par défaut, c'est tout l'intervalle des abscisses de la fenêtre),
  - `nbdots=40`,
  - `discont=false`,
  - `nbdiv=5`,
  - `draw_options=""`, chaîne qui sera transmise telle quelle à l'instruction *draw*,
  - `clip=nil`, cette option est soit `nil` (valeur par défaut), soit une table `{x1,x2,y1,y2}`, dans le premier cas la ligne est clippée par la fenêtre 2D courante **après** sa transformation par la matrice 2D du graphe, dans le second cas la ligne est clippée par la fenêtre $[x_1;x_2]\times[y_1;y_2]$ **avant** d'être transformée par la matrice du graphe.

Fonctions périodiques : Dperiodic

- La fonction **ld.periodic(f, period, x1, x2 [, nbdots, discont, nbdiv])** fait le calcul des points de la courbe et renvoie une ligne polygonale (pas de dessin).
  - L'argument ⟨f⟩ est la fonction dont on veut la courbe, ce doit être une fonction d'une variable réelle x et à valeurs réelles.
  - L'argument ⟨*period*⟩ est une table du type $\{a,b\}$ avec $a<b$ représentant une période de la fonction ⟨f⟩.
  - Les arguments ⟨*x1*⟩ et ⟨*x2*⟩ sont obligatoires et forment les bornes de l'intervalle pour la variable.
  - Les autres arguments sont identiques aux courbes paramétrées.
- La méthode **g:Dperiodic(f, period, options)** fait le calcul des points de la courbe et le dessin de la courbe de f. L'argument ⟨*options*⟩ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
  - `x={g:Xinf(),g:Xsup()}`, c'est l'intervalle pour le paramètre x (par défaut, c'est tout l'intervalle des abscisses de la fenêtre),
  - `nbdots=40`,
  - `discont=false`,
  - `nbdiv=5`,
  - `draw_options=""`, chaîne qui sera transmise telle quelle à l'instruction *draw*,
  - `clip=nil`, cette option est soit `nil` (valeur par défaut), soit une table `{x1,x2,y1,y2}`, dans le premier cas la ligne est clippée par la fenêtre 2D courante **après** sa transformation par la matrice 2D du graphe, dans le second cas la ligne est clippée par la fenêtre $[x_1;x_2]\times[y_1;y_2]$ **avant** d'être transformée par la matrice du graphe.

Fonctions en escaliers : Dstepfunction

- La fonction **ld.stepfunction(def, discont)** fait le calcul des points de la courbe et renvoie une ligne polygonale (pas de dessin).
  - L'argument ⟨*def*⟩ permet de définir la fonction en escaliers, c'est une table à deux éléments :

    ```
    { {x1,x2,x3,...,xn}, {c1,c2,...} }
    ```

    Le premier élément `{x1,x2,x3,…,xn}` doit être une subdivision du segment $[x_1;x_n]$.
    Le deuxième élément `{c1,c2,…}` est la liste des constantes avec `c1` pour le morceau $[x_1;x_2]$, `c2` pour le morceau $[x_2;x_3]$, etc.
  - L'argument ⟨*discont*⟩ est un booléen qui vaut `true` par défaut.
- La méthode **g:Dstepfunction(def, options)** fait le calcul des points et le dessin de la courbe de la fonction en escalier.
  - L'argument ⟨*def*⟩ est le même que celui décrit au-dessus (définition de la fonction en escalier).
  - L'argument ⟨*options*⟩ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
    * `discont=true`,
    * `draw_options=""`, chaîne qui sera transmise telle quelle à l'instruction *draw*,

* `clip=nil`, cette option est soit `nil` (valeur par défaut), soit une table `{x1,x2,y1,y2}`, dans le premier cas la ligne est clippée par la fenêtre 2D courante **après** sa transformation par la matrice 2D du graphe, dans le second cas la ligne est clippée par la fenêtre $[x_1;x_2]\times[y_1;y_2]$ **avant** d'être transformée par la matrice du graphe.

Fonctions affines par morceaux : Daffinebypiece

- La fonction **ld.affinebypiece(def, discont)** fait le calcul des points de la courbe et renvoie une ligne polygonale (pas de dessin).
  - L'argument ⟨*def*⟩ permet de définir la fonction en escaliers, c'est une table à deux champs :
    ```
    { {x1,x2,x3,...,xn}, { {a1,b1}, {a2,b2},...} }
    ```
    Le premier élément `{x1,x2,x3,…,xn}` doit être une subdivision du segment $[x_1;x_n]$.
    Le deuxième élément `{ {a1,b1},{a2,b2},…}` signifie que sur $[x_1;x_2]$ la fonction est $x\mapsto a_1x+b_1$, sur $[x_2;x_3]$ la fonction est $x\mapsto a_2x+b_2$, etc.
  - L'argument ⟨*discont*⟩ est un booléen qui vaut `true` par défaut.
- La méthode **g:Daffinebypiece(def, options)** fait le calcul des points et le dessin de la courbe de la fonction affine par morceaux.
  - L'argument ⟨*def*⟩ est le même que celui décrit au-dessus (définition de la fonction affine par morceaux).
  - L'argument ⟨*options*⟩ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
    * `discont=true`,
    * `draw_options=""`, chaîne qui sera transmise telle quelle à l'instruction *\draw*,
    * `clip=nil`, cette option est soit `nil` (valeur par défaut), soit une table `{x1,x2,y1,y2}`, dans le premier cas la ligne est clippée par la fenêtre 2D courante **après** sa transformation par la matrice 2D du graphe, dans le second cas la ligne est clippée par la fenêtre $[x_1;x_2]\times[y_1;y_2]$ **avant** d'être transformée par la matrice du graphe.

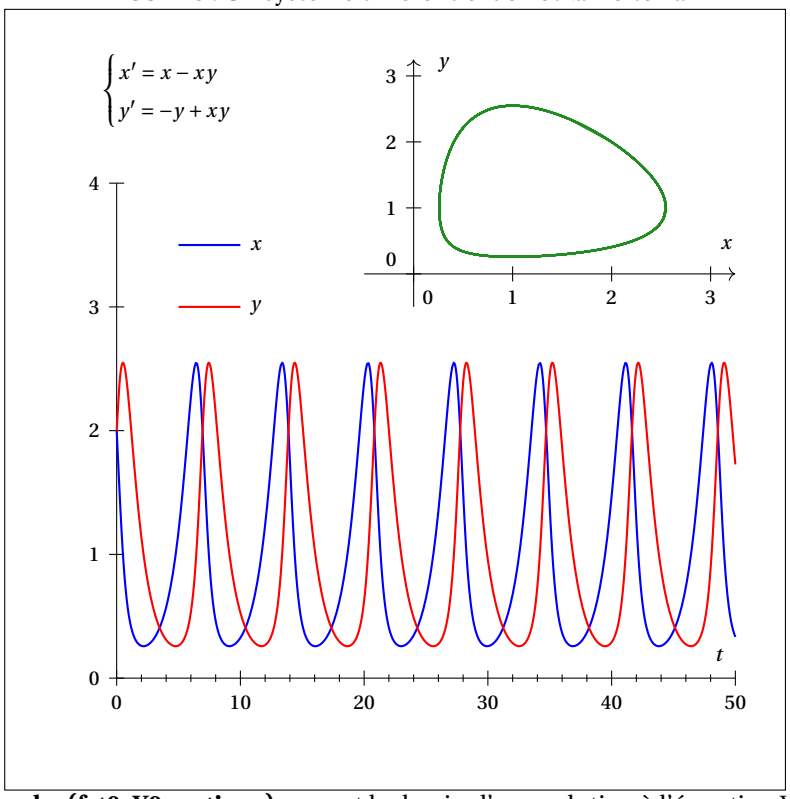
Équations différentielles : Dodesolve

- La fonction **ld.odesolve(f, t0, Y0, tmin, tmax, nbdots [, method])** permet une résolution approchée de l'équation différentielle $\mathrm{Y}'(t)=f(t,\mathrm{Y}(t))$ dans l'intervalle $[t_{\min};t_{\max}]$ qui doit contenir ⟨*t0*⟩, avec la condition initiale $\mathrm{Y}(t_0)=\mathrm{Y}_0$.
  - L'argument ⟨*f*⟩ est une fonction $f:(t,\mathrm{Y})\mapsto f(t,\mathrm{Y})$ à valeurs dans $\mathbf{R}^n$ et où Y est également dans $mathbf\mathrm{R}^n$: $\mathrm{Y}=\{y_1,y_2,\dots,y_n\}$, mais lorsque $n=1$, Y est un réel.
  - Les arguments ⟨*t0*⟩ et ⟨*Y0*⟩ donnent les conditions initiales avec $\mathrm{Y}_0=\{y_1(t_0),\dots,y_n(t_0)\}$ (les y_i sont réels), ou $\mathrm{Y}_0=y_1(t_0)$ lorsque $n=1$.
  - Les arguments ⟨*tmin*⟩ et ⟨*tmax*⟩ définissent l'intervalle de résolution, celui-ci doit contenir ⟨*t0*⟩.
  - L'argument ⟨*nbdots*⟩ indique le nombre de points calculés de part et d'autre de ⟨*t0*⟩.
  - L'argument optionnel ⟨*method*⟩ est une chaîne qui peut valoir `"rkf45"` (valeur par défaut), ou `"rk4"`. Dans le premier cas, on utilise la méthode de Runge Kutta-Fehlberg (à pas variable), dans le second cas c'est la méthode classique de Runge-Kutta d'ordre 4.
  - En sortie, la fonction renvoie la matrice suivante (liste de listes de réels) :
    ```
    { {tmin,...,tmax}, {y1(tmin),...,y1(tmax)}, {y2(tmin),...,y2(tmax)},...}
    ```
    La première composante est la liste des valeurs de t (dans l'ordre croissant), la deuxième est la liste des valeurs (approchées) de la composante y_1 correspondant à ces valeurs de t, etc.
- La méthode **g:DplotXY(X, Y [, draw_options, clip])**, où les arguments ⟨*X*⟩ et ⟨*Y*⟩ sont deux listes de réels de même longueur, dessine la ligne polygonale constituée des points $(\mathrm{X}[k],\mathrm{Y}[k])$. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*. L'argument ⟨*clip*⟩ est soit `nil` (valeur par défaut), soit une table `{x1,x2,y1,y2}`, dans le premier cas la ligne est clippée par la fenêtre 2D courante **après** sa transformation par la matrice 2D du graphe, dans le second cas la ligne est clippée par la fenêtre $[x_1;x_2]\times[y_1;y_2]$ **avant** d'être transformée par la matrice du graphe.

```
\begin{luadraw}{name=lotka_volterra}
local ld = luadraw
local cpx = ld.cpx
```

```
local g = ld.graph:new{window={-5,50,-0.5,5},size={10,10,0}, border=true}
local i = cpx.I
local f = function(t,y) return {y[1]-y[1]*y[2],-y[2]+y[1]*y[2]} end
g:Labelsize("footnotesize")
g:Daxes({0,10,1},{limits={{0,50},{0,4}}, nbsubdiv={4,0}, legendsep={0.1,0},
originpos={"center","center"}, legend={"$t$",""}})
local y0 = {2,2}
local M = ld.odesolve(f,0,y0,0,50,250) -- résolution approchée
-- M est une table à 3 éléments: t, x et y
g:Lineoptions("solid","blue",8)
g:Dseg({5+3.5*i,10+3.5*i}); g:Dlabel("$x$",10+3.5*i,{pos="E"})
g:DplotXY(M[1],M[2]) -- points (t,x(t))
g:Linecolor("red"); g:Dseg({5+3*i,10+3*i}); g:Dlabel("$y$",10+3*i,{pos="E"})
g:DplotXY(M[1],M[3]) -- points (t,y(t))
g:Lineoptions(nil,"black",4)
g:Saveattr(); g:Viewport(20,50,3,5) -- changement de vue
g:Coordsystem(-0.5,3.25,-0.5,3.25) -- nouveau repère associé
g:Daxes({0,1,1},{legend={"$x$","$y$"},arrows="->"})
g:Lineoptions(nil,"ForestGreen",8); g:DplotXY(M[2],M[3]) -- points (x(t),y(t))
g:Restoreattr() -- retour à l'ancienne vue
g:Dlabel("$\\begin{cases}x'=x-xy\\\\y'=-y+xy\\end{cases}$", 5+4.75*i,{})
g:Show()
\end{luadraw}
```

FIGURE 8 : Un système différentiel de Lotka-Volterra



- La méthode **g:Dodesolve(f, t0, Y0, options)** permet le dessin d'une solution à l'équation $\mathrm{Y}'(t) = f(t, \mathrm{Y}(t))$.
  - L'argument obligatoire ⟨*f*⟩ est une fonction $f : (t, \mathrm{Y}) \mapsto f(t, \mathrm{Y})$ à valeurs dans $\mathbf{R}^n$ et où Y est également dans $\mathbf{R}^n$: $\mathrm{Y} = \{y_1, y_2, \ldots, y_n\}$, mais lorsque $n = 1$, Y est un réel.
  - Les arguments ⟨*t0*⟩ et ⟨*Y0*⟩ donnent les conditions initiales avec $\mathrm{Y}_0 = \{y_1(t_0), \ldots, y_n(t_0)\}$ (les y_i sont réels), ou $\mathrm{Y}_0 = y_1(t_0)$ lorsque $n = 1$.
  - L'argument ⟨*options*⟩ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
    * `t={g:Xinf(), g:Xsup()}`, détermine l'intervalle pour la variable t,
    * `out={1,2}`, table de deux entiers $\{i_1, i_2\}$, si M désigne la matrice renvoyée par la fonction `odesolve`, les points dessinés auront pour abscisses les $\mathrm{M}[i_1]$ et pour ordonnées les $\mathrm{M}[i_2]$. Par défaut on a $i_1 = 1$ et $i_2 = 2$, ce qui correspond à la fonction y_1 en fonction de t,
    * `nbdots=50`, détermine le nombre de points à calculer pour la fonction,

* `method="rkf45"`, détermine la méthode à utiliser, les valeurs possibles sont `"rkf45"` ou `"rk4"`,
* `draw_options=""`, est une chaîne qui sera passée telle quelle à l'instruction *\draw*,
* `clip=nil`, cette option est soit `nil` (valeur par défaut), soit une table `{x1,x2,y1,y2}`, dans le premier cas la ligne est clippée par la fenêtre 2D courante **après** sa transformation par la matrice 2D du graphe, dans le second cas la ligne est clippée par la fenêtre $[x_1;x_2]\times[y_1;y_2]$ **avant** d'être transformée par la matrice du graphe.

Courbes implicites : Dimplicit

- La fonction **ld.implicit(f, x1, x2, y1, y2, grid)** calcule et renvoie une ligne polygonale constituant la courbe implicite d'équation $f(x,y)=0$ dans le pavé $[x_1,x_2]\times[y_1,y_2]$. Ce pavé est découpé en fonction de l'argument $\langle grid\rangle$.
  - L'argument $\langle f\rangle$ est une fonction $f:(x,y)\mapsto f(x,y)$ à valeurs dans **R**.
  - Les arguments $\langle x1\rangle$, $\langle x2\rangle$,$\langle y1\rangle$, $\langle y2\rangle$ définissent la fenêtre du tracé, qui sera le pavé $[x_1,x_2]\times[y_1,y_2]$, on doit avoir $x_1<x_2$ et $y_1<y_2$.
  - L'argument $\langle grid\rangle$ est une table contenant deux entiers positifs : $\{n_1,n_2\}$, le premier entier indique le nombre de subdivisions suivant l'axe des x, et le second le nombre de subdivisions suivant l'axe des y.
- La méthode **g:Dimplicit(f, options)** fait le dessin de la courbe implicite d'équations $f(x,y)=0$.
  - L'argument $\langle f\rangle$ est une fonction $f:(x,y)\mapsto f(x,y)$ à valeurs dans **R**.
  - L'argument $\langle options\rangle$ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
    * `view={g:Xinf(), g:Xsup(), g:Yinf(), g:Ysup()}`, table de la forme `{x1,x2,y1,y2}` qui détermine la zone de dessin $[x_1,x_2]\times[y_1,y_2]$.
    * `grid={50,50}`, détermine les subdivisions,
    * `draw_options=""`, est une chaîne qui sera passée telle quelle à l'instruction *\draw*.

Courbes de niveau : Dcontour

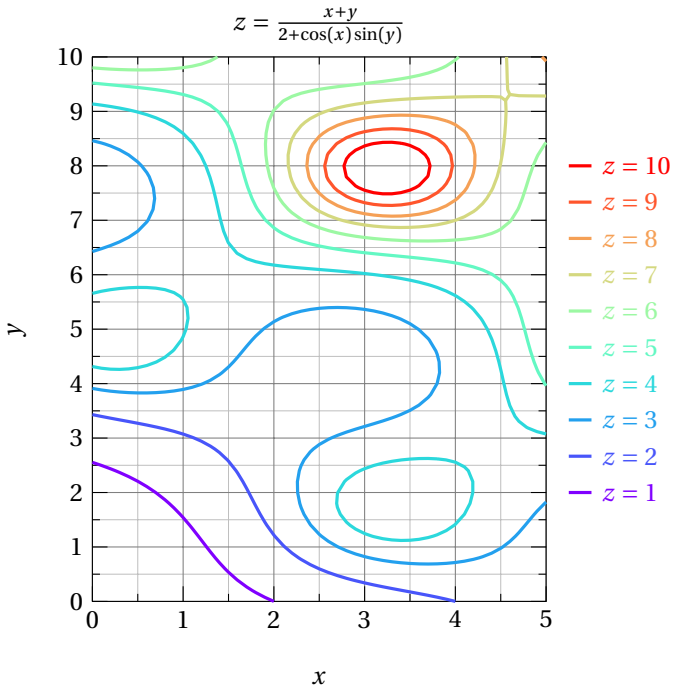
La méthode **g:Dcontour(f, z, options)** fait le dessin de **lignes de niveau** de la fonction $f:(x,y)\mapsto f(x,y)$ à valeurs réelles.

- L'argument $\langle f\rangle$ est la fonction.
- L'argument $\langle z\rangle$ est la liste des différents niveaux à tracer.
- L'argument $\langle options\rangle$ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
  - `view={g:Xinf(), g:Xsup(), g:Yinf(), g:Ysup()}`, table de la forme `{x1,x2,y1,y2}` qui détermine la zone de dessin $[x_1,x_2]\times[y_1,y_2]$.
  - `grid={50,50}`, détermine les subdivisions,
  - `colors {}`, la liste des couleurs par niveau, par défaut cette liste est vide et c'est la couleur courante de tracé qui est utilisée.
  - `draw_options=""`, est une chaîne qui sera passée telle quelle à l'instruction *\draw*.

```
\begin{luadraw}{name=Dcontour}
local ld = luadraw
local cpx = ld.cpx
local g = ld.graph:new{window={-1,6.5,-1.5,11},size={10,10,0}}
local i, sin, cos = cpx.I, math.sin, math.cos
local f = function(x,y) return (x+y)/(2+cos(x)*sin(y)) end
local Lz = ld.range(1,10) -- niveaux à tracer
local Colors = ld.getpalette(ld.palRainbow,10) -- 10 couleurs équiréparties dans la palette ld.palRainbow
g:Dgradbox({0,5+10*i,1,1},{legend={"$x$","$y$"}, grid=true, title="$z=\\frac{x+y}{2+\\cos(x)\\sin(y)}$"})
g:Linewidth(12); g:Dcontour(f,Lz,{view={0,5,0,10}, colors=Colors})
for k = 1, 10 do
    local y = (2*k+4)/3*i
    g:Dseg({5.25+y,5.5+y},1,"color="..Colors[k])
    g:Labelcolor(Colors[k])
    g:Dlabel("$z="..k.."$",5.5+y,{pos="E"})
end
```

```
g:Show()
\end{luadraw}
```

FIGURE 9 : Exemple avec Dcontour



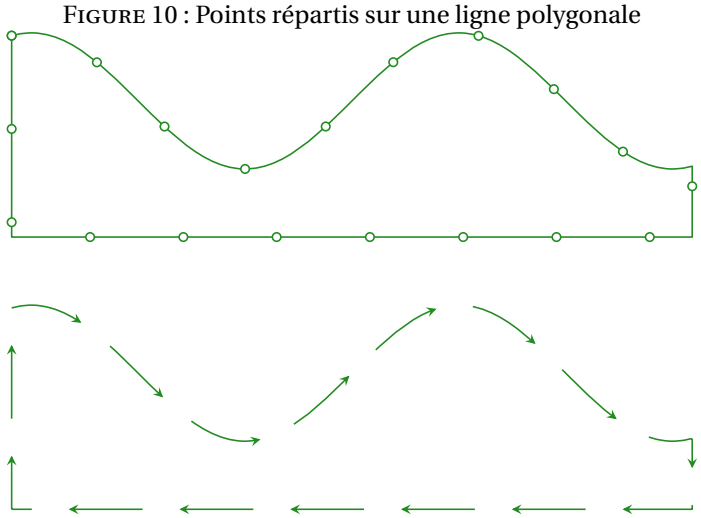
Paramétrisation d'une ligne polygonale : *curvilinear_param*

Soit L une liste de complexe représentant une ligne « continue », il est possible d'obtenir une paramétrisation de cette ligne en fonction d'un paramètre t entre 0 et 1 (t est l'abscisse curviligne divisée par la longueur totale de L).

La fonction **ld.curvilinear_param(L [, close])** renvoie une fonction d'une variable $t \in [0;1]$ et à valeurs sur la ligne $\langle L \rangle$ (liste de nombres complexes), la valeur en $t = 0$ est le premier point de $\langle L \rangle$, et la valeur en $t = 1$ est le dernier point ; cette fonction est suivie d'un nombre qui représente la longueur total de $\langle L \rangle$. L'argument optionnel $\langle close \rangle$ indique si la ligne $\langle L \rangle$ doit être refermée (`false` par défaut).

```
\begin{luadraw}{name=curvilinear_param}
local ld = luadraw
local cpx = ld.cpx
local g = ld.graph:new{bbox=false,size={10,10}}
local i = cpx.I; g:Linewidth(8)
local L = ld.cartesian(math.sin,-5,5)[1]
table.append(L, {5-2*i, -5-2*i})
local f = ld.curvilinear_param(L, true)
local I = ld.map(f,ld.linspace(0,1,20)) -- 20 points répartis sur L
g:Shift(4*i)
g:Lineoptions(nil,"ForestGreen",6); g:Dpolyline(L,true)
g:Filloptions("full","white"); g:Ddots(I) -- le premier et le dernier point sont confondus car L est fermée
-- autre exemple d'utilisation:
local nb = 16 -- nombre de flèches
local t = ld.linspace(0,1,3*nb+1)
g:Shift(-4*i)
for k = 0,nb-1 do
g:Dparametric(f,{t={t[3*k+1],t[3*k+3]},nbdots=10,nbdiv=2,draw_options="-stealth"})
end
g:Show()
\end{luadraw}
```

FIGURE 10 : Points répartis sur une ligne polygonale



5) Domaines liés à des courbes cartésiennes

Ddomain1

- La fonction **ld.domain1(f, a, b [, nbdots, discont, nbdiv])** renvoie une liste de complexes qui représente le contour de la partie du plan délimitée par la courbe de la fonction $\langle f \rangle$ sur un intervalle déterminé par $\langle a \rangle$ et $\langle b \rangle$, l'axe Ox, et les droites $x = a$, $x = b$.
- La méthode **g:Ddomain1(f, options)** dessine ce contour. L'argument $\langle options \rangle$ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
  - `x={g:Xinf(),g:Xsup()}`, c'est l'intervalle pour le paramètre x (par défaut, c'est tout l'intervalle des abscisses de la fenêtre),
  - `nbdots=40`,
  - `discont=false`,
  - `nbdiv=5`,
  - `draw_options=""`, chaîne qui sera transmise telle quelle à l'instruction *\draw*.

Ddomain2

- La fonction **ld.domain2(f, g, a, b [, nbdots, discont, nbdiv])** renvoie une liste de complexes qui représente le contour de la partie du plan délimitée par la courbe de la fonction $\langle f \rangle$, la courbe de la fonction $\langle g \rangle$, et les droites $x = a$, $x = b$.
- La méthode **g:Ddomain2(f, g, options)** dessine ce contour. L'argument $\langle options \rangle$ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
  - `x={g:Xinf(),g:Xsup()}`, c'est l'intervalle pour le paramètre x (par défaut, c'est tout l'intervalle des abscisses de la fenêtre),
  - `nbdots=40`,
  - `discont=false`,
  - `nbdiv=5`,
  - `draw_options=""`, chaîne qui sera transmise telle quelle à l'instruction *\draw*.

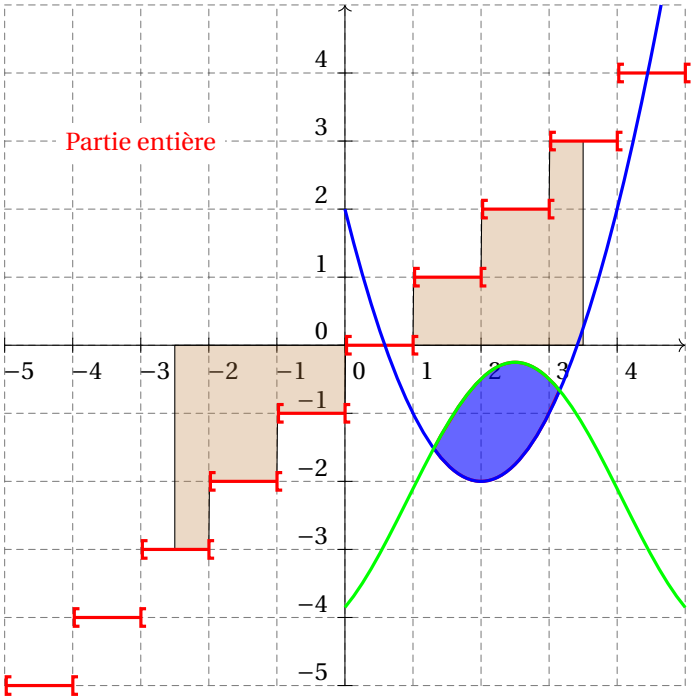
Ddomain3

- La fonction **ld.domain3(f, g, a, b [, nbdots, discont, nbdiv])** renvoie une liste de complexes qui représente le contour de la partie du plan délimitée par la courbe de la fonction $\langle f \rangle$ et celle de la fonction $\langle g \rangle$ avec recherche des points d'intersection dans l'intervalle défini par $\langle a \rangle$ et $\langle b \rangle$.
- La méthode **g:Ddomain3(f, g, options)** dessine ce contour. L'argument $\langle options \rangle$ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
  - `x={g:Xinf(),g:Xsup()}`, c'est l'intervalle pour le paramètre x (par défaut, c'est tout l'intervalle des abscisses de la fenêtre),
  - `nbdots=40`,

- `discont=false`,
- `nbdiv=5`,
- `draw_options=""`, chaîne qui sera transmise telle quelle à l'instruction *\draw*.

```
\begin{luadraw}{name=courbe}
local ld = luadraw
local Z = ld.cpx.Z
local g = ld.graph:new{ window={-5,5,-5,5}, bg="", size={10,10} }
local f = function(x) return (x-2)^2-2 end
local h = function(x) return 2*math.cos(x-2.5)-2.25 end
g:Daxes( {0,1,1},{grid=true,gridstyle="dashed", arrows="->"})
g:Filloptions("full","brown",0.3)
g:Ddomain1( math.floor, { x={-2.5,3.5} })
g:Filloptions("none","white",1); g:Lineoptions("solid","red",12)
g:Dstepfunction( {ld.range(-5,5), ld.range(-5,4)},{draw_options="arrows={Bracket-Bracket[reversed]},shorten >=-2pt"})
g:Labelcolor("red")
g:Dlabel("Partie entière",Z(-3,3),{node_options="fill=white"})
g:Ddomain3(f,h,{draw_options="fill=blue,fill opacity=0.6"})
g:Dcartesian(f, {x={0,5}, draw_options="blue"})
g:Dcartesian(h, {x={0,5}, draw_options="green"})
g:Show()
\end{luadraw}
```

FIGURE 11 : Partie entière, fonctions Ddomain1 et Ddomain3



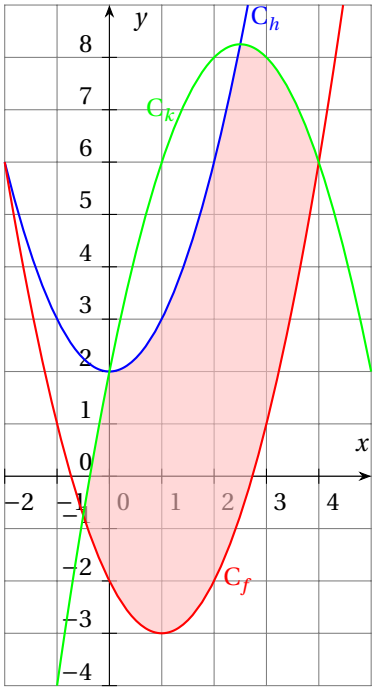
Dinequalities

- La fonction **ld.inequalities(constraints, x1, x2, y1, y2)** renvoie une liste de nombres complexes représentant le contour de la partie du plan située dans le pavé $[x_1;x_2]\times[y_1;y_2]$ et vérifiant les contraintes exprimées dans l'argument ⟨*contraints*⟩, celui-ci est une liste de la forme `{f1,sg1,f2,sg2,...fn,sgn}` où les ⟨*fi*⟩ sont des fonctions (f_i: $x\mapsto f_i(x)\in\mathbf{R}$), et les ⟨*sgi*⟩ sont soit `">"`, soit `"<"`, la première contrainte est donnée par ⟨*f1*⟩ et ⟨*sg1*⟩, cette contrainte est soit $y>f_1(x)$, soit $y<f_1(x)$ en fonction de la valeur de ⟨*sg1*⟩. Il en va de même pour les contraintes suivantes.
- La méthode **g:Dinequalities(constraints, options)** permet de dessiner ce contour. L'argument ⟨*options*⟩ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
  - `view={g:Xinf(),g:Xsup(),g:Yinf(),g:Ysup()}`, c'est la fenêtre dans laquelle va se faire la résolution. Par défaut c'est la fenêtre 2D courante.

- `draw_options=""`, chaîne qui sera transmise telle quelle à l'instruction *\draw*.
- `useclip=false`, avec la valeur `false` la méthode utilise la fonction précédente (qui calcule le contour de la solution), avec la valeur `true` la méthode ne calcule pas le contour mais utilise des "clipping" (un par contrainte).

```
\begin{luadraw}{name=Dinequalities}
local ld = luadraw
local Z = ld.cpx.Z
local g = ld.graph:new{window={-2,5,-4,9}, size={10,10}}
g:Daxes({0,1,1}, {grid=true, arrows="-Stealth", legend={"$x$","$y$"}})
local f = function(x) return x*x - 2*x - 2 end
local h = function(x) return x*x + 2 end
local k = function(x) return -x*x+5*x + 2 end
g:Dinequalities({f,'>', h,'<', k,'<'}, {draw_options="draw=none,fill=pink,fill opacity=0.6"})
g:Dcartesian(h, {draw_options="blue, line width=0.8pt"}); g:Dlabel("$C_h$",Z(3,8.75),{node_options="blue"})
g:Dcartesian(f, {draw_options="red, line width=0.8pt"}); g:Dlabel("$C_f$",Z(2,-2),{pos="E",node_options="red"})
g:Dcartesian(k, {draw_options="green, line width=0.8pt"}); g:Dlabel("$C_k$",Z(1,7),{node_options="green"})
g:Show()
\end{luadraw}
```

FIGURE 12 : Dinequalities



Dimplicit_inequalities

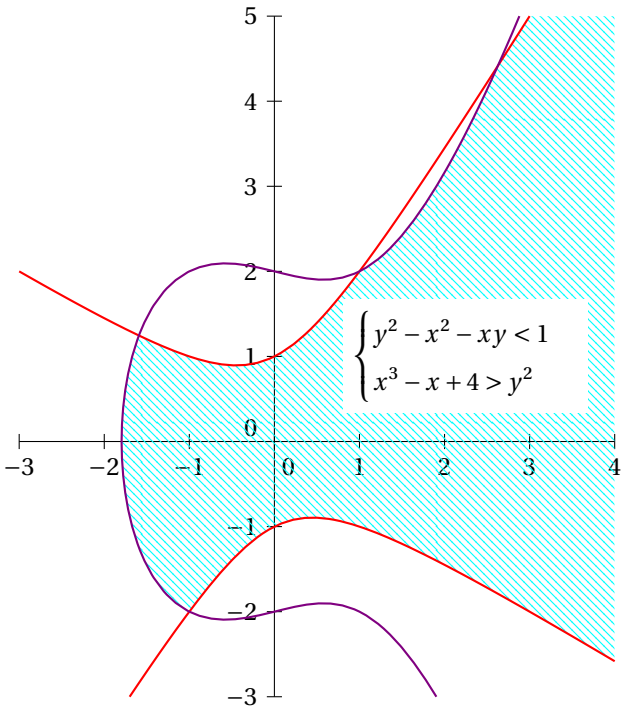
- La fonction **ld.implicit_inequality(f, sg, x1, x2, y1, y2, grid)** renvoie un **chemin** représentant le contour de la partie du plan située dans le pavé $[x_1;x_2]\times[y_1;y_2]$ et vérifiant la condition $f(x,y)>=0$ si ⟨*sg*⟩=">", ou $f(x,y)<=0$ si ⟨*sg*⟩="<". L'argument ⟨*f*⟩ est une fonction à deux variables ⟨*f*⟩ : $(x,y)\mapsto f(x,y)\in\mathbf{R}$. L'argument ⟨*grid*⟩ est une table de deux entiers : ⟨*grid*⟩={⟨*n1*⟩,⟨*n2*⟩}, le premier indique le nombre de subdivisions de l'intervalle $[x_1;x_2]$, et le second, le nombre de subdivisions de l'intervalle $[y_1;y_2]$.
- La méthode **g:Dimplicit_inequalities(constraints, options)** permet de remplir une partie du plan vérifiant les contraintes exprimées dans l'argument ⟨*contraints*⟩, celui-ci est une liste de la forme `{f1,sg1,f2,sg2,...fn,sgn}` où les ⟨*fi*⟩ sont des fonctions (f_i : $(x,y)\mapsto f_i(x,y)\in\mathbf{R}$), et les ⟨*sgi*⟩ sont soit `">"`, soit `"<"`, la première contrainte est donnée par ⟨*f1*⟩ et ⟨*sg1*⟩, cette contrainte est soit $f_1(x,y)>0$, soit $f_1(x,y)<0$ en fonction de la valeur de ⟨*sg1*⟩. L'argument ⟨*options*⟩ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
  - `view={g:Xinf(),g:Xsup(),g:Yinf(),g:Ysup()}`, c'est la fenêtre dans laquelle va se faire la résolution. Par défaut c'est la fenêtre 2D courante.

– `draw_options=""`, chaîne qui sera transmise telle quelle à l'instruction *\draw*.
– `grid={50,50}` : table de deux entiers, le premier indique le nombre de subdivisions de l'intervalle des x, et le second, le nombre de subdivisions de l'intervalle ds y.

NB : la méthode ne calcule pas le contour mais utilise des "clipping" (un par contrainte).

```
\begin{luadraw}{name=implicit_inequalities}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z
local g = ld.graph:new{ window = {-3, 4, -3, 5}, size = {10, 10} }
local f1 = function(x,y) return -x*y+y^2-x^2-1  end
local f2 = function(x,y) return x^3-x-y^2+4 end
local filloptions = "draw=none,pattern= north west lines, pattern color=cyan"
g:Daxes()
g:Dimplicit_inequalities( {f1,'<',f2,">"}, {view={-2,5,-5,5}, draw_options=filloptions})
g:Dimplicit(f1, {draw_options="red,thick"})
g:Dimplicit(f2, {view={-2,5,-5,5}, draw_options="violet,thick"})
local eq = \luastring0{$\begin{cases}y^2-x^2-xy < 1\\ x^3-x+4 > y^2\end{cases}$}
g:Dlabel( eq, Z(2.25,1), {node_options="fill=white"})
g:Show()
\end{luadraw}
```

FIGURE 13 : La méthode **g:Dimplicit_inequalities**



6) Points (Ddots) et labels (Dlabel)

- La méthode pour dessiner un ou plusieurs points est : **g:Ddots(dots [, mark_options])**.
  – L'argument ⟨*dots*⟩ peut être soit un seul point (donc un complexe), soit une liste (une table) de complexes, soit une liste de listes de complexes. Les points sont dessinés dans la couleur courante du tracé de lignes.
  – L'argument ⟨*mark_options*⟩ est une chaîne de caractères (vide par défaut) qui sera passée telle quelle à l'instruction *\draw* (modifications locales), exemple :

```
mark_options = "color=green, line width=1.2, scale=0.25"
```

  – Deux méthodes pour modifier globalement l'apparence des points :
    * La méthode **g:Dotstyle(style)** qui définit le style de point, l'argument ⟨*style*⟩ est une chaîne de caractères qui vaut par défaut `"*"`. Les styles possibles sont ceux de la librairie *plotmarks*.
    * La méthode **g:Dotscale(scale)** permet de jouer sur la taille du point, l'argument ⟨*scale*⟩ est un entier positif qui vaut 1 par défaut, il sert à multiplier la taille par défaut du point. La largeur courante de tracé de ligne intervient

également dans la taille du point. Pour les style de points "pleins" (par exemple le style `"triangle*"`), le style et la couleur de remplissage courants sont utilisés par la librairie.

- La méthode pour placer un label est :

  g:Dlabel(text1, anchor1, options1, text2, anchor2, options2, ...)

  – Les arguments ⟨*text1*⟩, ⟨*text2*⟩, ..., sont des chaînes de caractères, ce sont les labels.
  – Les arguments ⟨*anchor1*⟩, ⟨*anchor2*⟩, ..., sont des complexes représentant les points d'ancrage des labels.
  – Les arguments ⟨*options1*⟩, ⟨*options2*⟩, ..., permettent de définir localement les paramètres des labels, ces paramètres sont des tables dont les champs définissent les options, qui sont :
    * `pos="center"`, indique la position du label par rapport au point d'ancrage, il peut valoir `"center"` (centré), `"N"` (nord), `"NE"` (nord est), `"E"` (est), `"SE"` (sud est), `"S"` (sud), `"SW"` (sud ouest), `"W"` (ouest), `"NW"` (nord ouest). Par défaut, il vaut `center`, et dans ce cas le label est centré sur le point d'ancrage,
    * `dist=0`, distance en cm entre le label et son point d'ancrage lorsque `pos` n'est pas égal à `"center"`,
    * `dir= {}`, cette option est une table de la forme `{dirX [,dirY,dep]}` qui indique la direction de l'écriture. Les 3 valeurs `dirX`, `dirY` et `dep` sont trois complexes représentant 3 vecteurs, les deux premiers indiquent le sens de l'écriture, le troisième un déplacement (translation) du label par rapport au point d'ancrage. Le vecteur `dep` est nul par défaut, et le vecteur `dirY`, s'il est absent, est égal au vecteur `dirX` tourné de 90 degrés dans le sens direct. Par défaut l'option `dir` est égale à la valeur courante de la direction de l'écriture,
    * `node_options=""` est une chaîne (vide par défaut) destinée à recevoir des options qui seront directement passées à TikZ dans l'instruction *node[]*.
    * Les labels sont dessinés dans la couleur courante du texte du document, mais on peut changer de couleur avec l'option `node_options` en mettant par exemple : `node_options="color=blue"`.

      Attention : les options choisies pour un label s'appliquent aussi aux labels suivants si elles sont inchangées.

  Options globales pour les labels :
  – la méthode **g:Labelstyle(position)** permet de préciser la position des labels par rapport aux points d'ancrage. L'argument ⟨*position*⟩ est une chaîne qui peut valoir : `"N"` pour nord, `"NE"` pour nord-est, `"NW"` pour nord-ouest, ou encore `"S"`, `"SE"`, `"SW"`. Par défaut, il vaut `"center"`, et dans ce cas le label est centré sur le point d'ancrage.
  – La méthode **g:Labelcolor(color)** permet de définir la couleur des labels. L'argument ⟨*color*⟩ est une chaîne représentant une couleur pour TikZ. Par défaut l'argument est une chaîne vide ce qui représente la couleur courante du document.
  – La méthode **g:Labelangle(angle)** permet de préciser un ⟨*angle*⟩ (en degrés) de rotation des labels autour du point d'ancrage, cet angle est nul par défaut.
  – La méthode **g:Labelsize(size)** permet de gérer la taille des labels. L'argument ⟨*size*⟩ est une chaîne qui peut valoir : `"tiny"`, ou `"scriptsize"` ou `"footnotesize"`, etc. Par défaut l'argument est une chaîne vide, ce qui représente la taille `"normalsize"`.
  – La méthode **g:Labeldir(dir)** permet de gérer la direction l'écriture. l'argument ⟨*dir*⟩ est une table de la forme `{dirX [,dirY,dep]}`, les 3 valeurs `dirX`, *dirY* et `dep` sont trois complexes représentant 3 vecteurs, les deux premiers indiquent le sens de l'écriture, le troisième un déplacement (translation) du label par rapport au point d'ancrage. Le vecteur `dep` est nul par défaut, et le vecteur `dirY`, s'il est absent, est égal au vecteur `dirX` tourné de 90 degrés dans le sens direct. Lorsque ⟨*dir*⟩ est une liste vide, cela représente le sens usuel de l'écriture.

- La méthode **g:Dlabeldot(text, anchor, options)** permet de placer un label et de dessiner le point d'ancrage en même temps.
  – L'argument ⟨*texte*⟩ est une chaîne de caractères, c'est le label,
  – L'argument ⟨*anchor*⟩ est un complexe représentant le point d'ancrage du label,
  – L'argument ⟨*options*⟩ est une table dont les champs sont les options possibles. Celles-ci sont à celles de la méthode **Dlabel**, plus l'option `mark_options=""` qui est une chaîne de caractères qui sera passée telle quelle à l'instruction *\draw* lors du dessin du point d'ancrage.

7) Chemins : Dpath, Dspline et Dtcurve

Qu'est ce qu'un chemin

Un chemin est une table de nombres complexes et d'instructions (sous forme de chaînes), cette table représente une succession de différents "morceaux", chaque morceau est une succession de données (points 2D et parfois valeurs numériques) et se termine par une chaîne de caractères qui représente une instruction. Le chemin est régi par la règle suivante :

le dernier point d'un morceau est le premier point du morceau suivant (il n'est donc pas répété)

Exemple :

```
local Z = cpx.Z
local L = { Z(-3,2),"m",-3,-2,"l",0,2,2,-1,"ca",3,Z(3,3),0.5,"la",1,Z(-1,5),Z(-3,2),"b" }
```

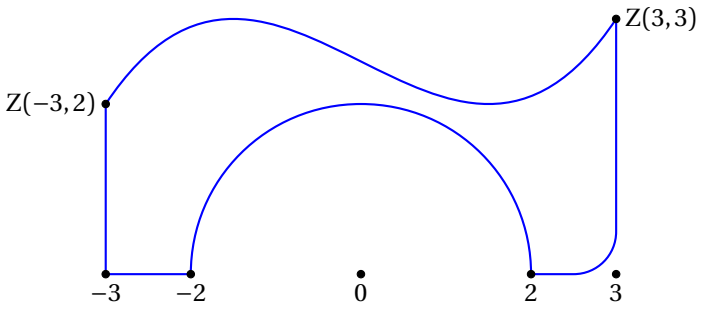
Le chemin L est composé de cinq morceaux, qui sont :

1. `{Z(-3,2),"m"}` : il y a une donnée et l'instruction `"m"` qui signifie *moveto*, cette instruction ne fait pas de dessin proprement dit, mais elle permet de commencer une nouvelle composante dont le premier point est $Z(-3,2)$ (le dernier point du morceau précédent, s'il y en a un, n'est pas pris en compte par cette instruction, c'est la seule exception).
2. `{Z(-3,2),-3,-2,"l"}` : le premier point du deuxième morceau est bien $Z(-3,2)$ et non pas -3, car $Z(-3,2)$ est le dernier point du morceau précédent. Il y a donc trois données et l'instruction "l" qui signifie *lineto*, c'est comme si on exécutait l'instruction `g:Dpolyline(Z(-3,2),-3,-2)`, ces trois points sont donc reliés par un segment. Le dernier point de ce morceau est -2.
3. `{-2,0,2,2,-1,"ca"}` : le premier point du troisième morceau est bien -2 et non pas 0, car -2 est le dernier point du morceau précédent. Il y a cinq données et l'instruction `"ca"` qui signifie *circle arc*, c'est comme si on exécutait l'instruction `g:Darc(-2,0,2,2,-1)`, donc le centre est 0, l'arc va de -2 à 2 avec un rayon égal à 2 et dans le sens des aiguilles d'un montre (dernière valeur -1). Le dernier point de ce morceau est 2.
4. `{2,3,Z(3,3),0.5,"la"}` : le premier point du quatrième morceau est 2 (et non pas 3), il y a quatre données et l'instruction `"la"` qui signifie *line arc*, c'est une ligne polygonale aux angles arrondis avec un arc de cercle de rayon 0.5 (valeur qui précède l'instruction). Les points de cette ligne sont `{2,3,Z(3,3)}`, il y aura donc un arrondi en 3. Le dernier point de ce morceau est $Z(3,3)$.
5. `{Z(3,3),1,Z(-1,5),Z(-3,2),"b"}` : le premier point du cinquième morceau est $Z(3,3)$ (et non pas 1), l'instruction `"b"` signifie *bezier*, on dessine donc une courbe de Bézier allant de $Z(3,3)$ jusqu'à $Z(-3,2)$, les deux autres points, 1 et $Z(-1,5)$ sont le premier et le deuxième point de contrôle de la courbe.

Voici ce que donne ce chemin :

```
\begin{luadraw}{name=path_example}
local ld = luadraw
local Z = ld.cpx.Z
local g = ld.graph:new{window={-4,4,-0.5,3}, size={10,10}}
local L = { Z(-3,2),"m",-3,-2,"l",0,2,2,-1,"ca",3,Z(3,3),0.5,"la",1,Z(-1,5),Z(-3,2),"b" }
g:Dpath(L, "line width=0.8pt, blue")
g:Ddots({Z(-3,2),-3,-2,0,2,3,Z(3,3)})
g:Dlabel("$Z(-3,2)$",Z(-3,2),{pos="W"}, "$-3$",-3,{pos="S"}, "$-2$",-2,{},
"$0$",0,{}, "$2$",2,{}, "$3$",3,{}, "$Z(3,3)$",Z(3,3),{pos="E"})
g:Show()
\end{luadraw}
```

FIGURE 14 : Path example



Remarque : dans l'exemple ci-dessus, vous pouvez remplacer la partie : `Z(-3,2),"m",-3,-2,"l"`, par : `Z(-3,2),-3,-2,"l"` car il n'y a pas d'autre morceau avant le *moveto*.

Instructions disponibles et leur syntaxe, le mot *last* représente le dernier point du morceau précédent :

- `z1,"m"` (moveto), permet de commencer une nouvelle composante du chemin au point d'affixe z_1.
- `z1,...,zn,"l"` (lineto), dessine la ligne polygonale `{last,z1,...,zn}`.
- `c1,c2,z2,"b"` (bézier) dessine la courbe de Bézier `{last,c1,c2,z2}`, où c_1 et c_2 sont les deux points de contrôle.
- `z1,...,zn,"s"` (spline), dessine la spline cubique naturelle passant par les points `{last,z1,...,zn}`.
- `z1,"c"` (cercle), dessine le cercle de centre z_1 et passant par le point *last*. Il y a une autre syntaxe possible pour le cercle : `z1,z2,"c"`, dans ce cas on dessine le cercle passant par les points *last*, z_1 et z_2.
- `z1,z2,r,sens,"ca"` (arc de cercle), dessine un arc de cercle de centre z_1, de rayon r, allant de *last* vers z_2, dans le sens trigonométrique lorsque `sens=1` (et donc dans le sens inverse si `sens=-1`).
- `z1,z2,rx,ry,sens,inclinaison,"ea"` (arc d'ellipse), dessine un arc d'ellipse de centre z_1 allant de *last* vers z_2, rx et ry sont les deux rayons suivant les deux axes de l'ellipse, `inclinaison` est l'angle en degrés que fait le premier axe de l'ellipse (celui qui porte rx) avec l'horizontal. Le paramètre `inclinaison` est facultatif, il vaut 0 par défaut.
- `z1,rx,ry,inclinaison,"e"` (ellipse), dessine l'ellipse de centre z_1, passant par *last*, rx et ry sont les deux rayons suivant les deux axes de l'ellipse, `inclinaison` est l'angle en degrés que fait le premier axe de l'ellipse (celui qui porte rx) avec l'horizontal. Le paramètre `inclinaison` est facultatif, il vaut 0 par défaut. Lorsque le point *last* n'est pas sur cette ellipse, alors un segment est tracé entre ce point et un point de l'ellipse.
- `z1,...,zn,r,"la"` (line arc), dessine la ligne polygonale `{last,z1,...,zn}` en remplaçant chaque "angle" par un arc de cercle de rayon r.
- `z1,...,zn,r,"cla"` (closed line arc), même chose que l'instruction précédente, sauf que la ligne est refermée.
- `"cl"` (closepath), cette instruction s'utilise seule, elle permet de refermer la composante courante en traçant un segment reliant le dernier point au premier point (de la composante courante).

Fonctions sur les chemins

- La fonction **ld.path(chemin [, nbdots])** renvoie une ligne polygonale contenant les points constituant le ⟨*chemin*⟩. L'argument facultatif, ⟨*nbdots*⟩ est le nombre minimal de points calculés pour chaque courbe de Bézier contenue dans le chemin, sa valeur par défaut est la variable globale `ld.bezier_nbdots` qui est initialisée à 12.
- À l'inverse, la fonction **ld.polyline2path(L [, close])** où ⟨L⟩ est une ligne polygonale (liste de nombres complexes ou une liste de listes de nombres complexes), renvoie ⟨L⟩ sous la forme d'un chemin. L'argument optionnel ⟨*close*⟩ permet de refermer ou non chaque composante du chemin (`false` par défaut).
- La fonction **ld.convpath(chemin)** convertit le ⟨*chemin*⟩ en un autre chemin n'utilisant que les instructions "m", "l" et "b" (toutes les autres instructions sont transformées en courbes de Bézier). La fonction renvoie le nouveau chemin, l'intérêt de ce nouveau chemin est qu'on peut lui appliquer une matrice de transformation (fonction **ld.mtransform()**).
- La fonction **ld.parabola(A, B, C)** renvoie un chemin qui est une courbe de Bézier représentant un arc de parabole allant du point ⟨A⟩ (nombre complexe) au point ⟨C⟩ (nombre complexe), en passant par le point ⟨B⟩ (nombre complexe).
- La fonction **ld.splitpath(P,I)**, où ⟨P⟩ est un chemin et ⟨I⟩ un point qui appartient à ce chemin, renvoie une séquence de deux chemins, le premier correspond à la partie de ⟨P⟩ qui précède le point I, le second est la partie qui suit le

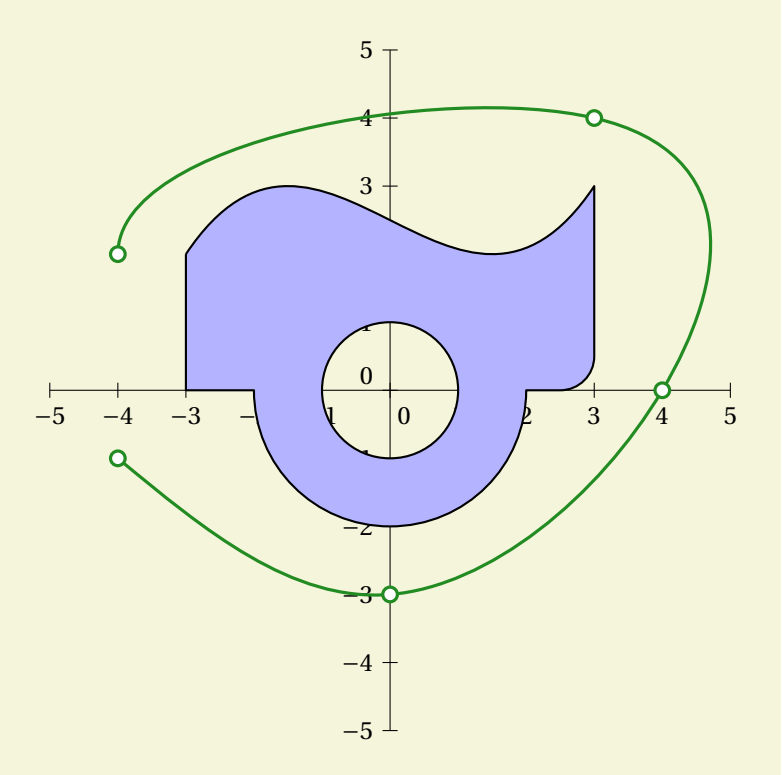
point I.

Dessiner un chemin

- La méthode **g:Dpath(chemin [, draw_options])** fait le dessin du ⟨*chemin*⟩ (qui été décrit ci-dessus) en utilisant au maximum les courbes de Bézier, y compris pour les arcs, les ellipses, etc. L'argument ⟨*draw_options*⟩ est une chaîne de caractères qui sera passée directement à l'instruction *\draw*.
- La fonction **ld.spline(points [, v1, v2])** renvoie sous forme de chemin (à dessiner avec **Dpath**) la spline cubique passant par les points de l'argument ⟨*points*⟩ (qui doit être une liste de complexes). Les arguments ⟨*v1*⟩ et ⟨*v2*⟩ sont vecteurs tangents imposés aux extrémités (contraintes), lorsque ceux-ci sont égaux à `nil` (valeur par défaut), c'est une spline cubique naturelle (c'est-à-dire sans contrainte) qui est calculée.
- La méthode **g:Dspline(points [, v1, v2, draw_options])** fait le dessin de la spline décrite ci-dessus. L'argument ⟨*draw_options*⟩ est une chaîne de caractères qui sera passée directement à l'instruction *\draw*.

```
\begin{luadraw}{name=path_spline}
local ld = luadraw
local g = ld.graph:new{window={-5,5,-5,5},size={10,10},bg="Beige"}
local i = ld.cpx.I
local p = {-3+2*i,"m",-3,-2,"l",0,2,2,1,"ca",3,3+3*i,0.5,"la",1,-1+5*i,-3+2*i,"b",-1,"m",0,"c"}
g:Daxes( {0,1,1} )
g:Filloptions("full","blue!30",1,true); g:Dpath(p,"line width=0.8pt")
g:Filloptions("none")
local A,B,C,D,E = -4-i,-3*i,4,3+4*i,-4+2*i
g:Lineoptions(nil,"ForestGreen",12); g:Dspline({A,B,C,D,E},nil,-5*i) -- contrainte en E
g:Ddots({A,B,C,D,E},"fill=white,scale=1.25")
g:Show()
\end{luadraw}
```

FIGURE 15 : Path et Spline



- La fonction **ld.tcurve(L)** renvoie sous forme de chemin une courbe passant par des points donnés avec des vecteurs tangents (à gauche et à droite) imposés à chaque point. L'argument ⟨*L*⟩ est une table de la forme :

```
L = {point1,{t1,a1,t2,a2}, point2,{t1,a1,t2,a2}, ..., pointN,{t1,a1,t2,a2}}
```

⟨*point1*⟩, ..., ⟨*pointN*⟩ sont les points d'interpolation de la courbe (affixes), et chacun d'eux est suivi d'une table de la forme `{t1,a1,t2,a2}` qui précise les vecteurs tangents à la courbe à gauche du point (avec ⟨*t1*⟩ et ⟨*a1*⟩) et à droite du point (avec ⟨*t2*⟩ et ⟨*a2*⟩). Le vecteur tangent à gauche est donné par la formule $V_g = t_1 \times e^{ia_1\pi/180}$, donc ⟨*t1*⟩ représente le module et ⟨*a1*⟩ est un argument **en degrés** de ce vecteur. C'est la même chose avec ⟨*t2*⟩ et ⟨*a2*⟩ pour

le vecteur tangent à droite, **mais ceux-ci sont facultatifs**, et s'ils ne sont pas précisés alors ils prennent les mêmes valeurs que ⟨*t1*⟩ et ⟨*a1*⟩.

Deux points consécutifs seront reliés par une courbe de Bézier, la fonction calcule les points de contrôle pour avoir les vecteurs tangents souhaités.

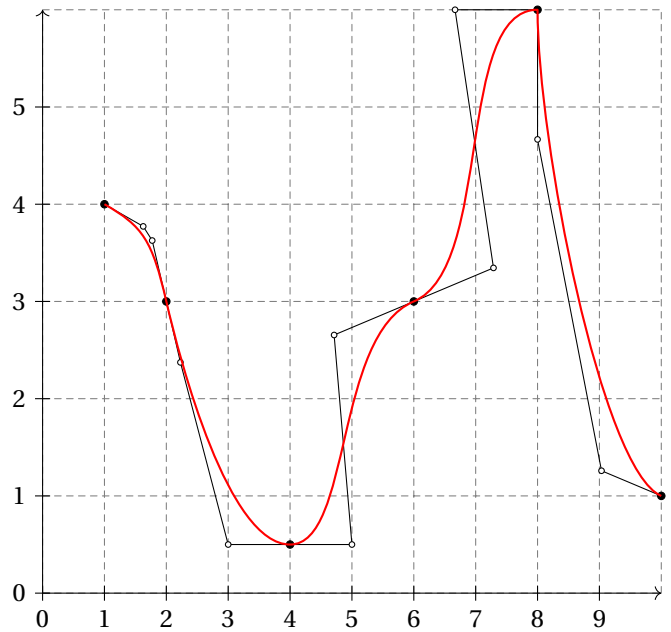
- La méthode **g:Dtcurve(L, options)** fait le dessin du chemin obtenu par ⟨*tcurve*⟩ décrit ci-dessus. L'argument ⟨*options*⟩ est une table dont les champs sont les options possibles, qui sont, avec leur valeur par défaut :
  - `showdots=false`, cette option permet de dessiner ou non les points d'interpolation donnés ainsi que les points de contrôles calculés, ce qui permet une visualisation des contraintes.
  - `draw_options=""`, c'est une chaîne de caractères qui sera passée directement à l'instruction *\draw*.
- Un chemin qui ne contient que des courbes de Bézier peut être dessiné avec la méthode :

  g:Dbezier(L [, draw_options])

  où ⟨*L*⟩ est une liste de la forme $\{A_1, c_1, c_2, A_2, c_3, c_4, A_3, \ldots\}$, les complexes $A_1, A_2, \ldots$, sont les points de la courbe, et les complexes, c_1, c_2, c_3, c_4, …, sont les points de contrôle.

```
\begin{luadraw}{name=tcurve}
local ld = luadraw
local g = ld.graph:new{window={-0.5,10.5,-0.5,6.5},size={10,10,0}}
local i = ld.cpx.I
local L = {
1+4*i,{2,-20},
2+3*i,{2,-70},
4+i/2,{3,0},
6+3*i,{4,15},
8+6*i,{4,0,4,-90}, -- point anguleux
10+i,{3,-15}}
g:Dgrid({0,10+6*i},{gridstyle="dashed"})
g:Daxes(nil,{limits={{0,10},{0,6}},originpos={"center","center"}, arrows="->"})
g:Dtcurve(L,{showdots=true,draw_options="line width=0.8pt,red"})
g:Show()
\end{luadraw}
```

FIGURE 16 : Courbe d'interpolation avec vecteurs tangents imposés

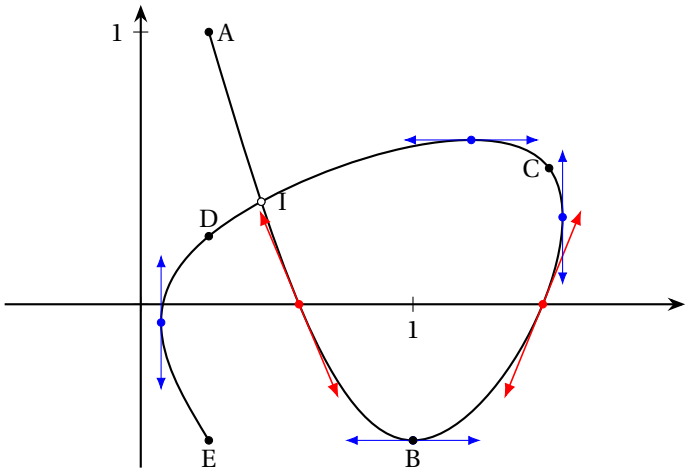


8) Paramétrer un chemin

La fonction **ld.param_path(P)** où ⟨*P*⟩ est une table représentant une chemin, renvoie une fonction $p\colon t \mapsto p(t) \in \mathbb{C}$ avec $t \in [0;1]$, cette fonction est une paramétrisation du chemin ⟨*P*⟩.

```
\begin{luadraw}{name=param_path}
local ld = luadraw
local Z = ld.cpx.Z
local g = ld.graph:new{ window={-0.5,2,-0.6,1.1}, size={10,10}}
local A, B, C, D, E = Z(0.25,1), Z(1,-0.5), Z(1.5,0.5), Z(0.25,0.25), Z(0.25,-0.5)
P = ld.spline({A,B,C,D,E})
local p = ld.param_path(P)
local T1, T2 = table.unpack(ld.solve(function(t) return p(t).im end, 0, 2))
local L = ld.parametric(p,0,1)
local I = ld.interL(L,L) -- self intersection
g:Daxes({0,1,1},{arrows="thick,-Stealth",originpos={"none","none"}})
g:Dpath(P,"thick")
g:Dtangent(p, T1, 0.75, "red,semithick,Latex-Latex")
g:Dtangent(p, T2, 0.75, "red,semithick,Latex-Latex")
g:Ddots({p(T1), p(T2)}, "red")
local dots = {}
g:Dparallel_tangent(p,Z(1,0),{t={0,1}, len=0.5, draw_options="blue,Latex-Latex", out=dots})
g:Dparallel_tangent(p,Z(0,1),{t={0,1}, len=0.5, draw_options="blue,Latex-Latex", out=dots})
for _,t in ipairs(dots) do g:Ddots(p(t),"blue") end
g:Ddots(I,"circle,fill=white")
g:Ddots({A,B,C,D,E})
g:Dlabel("$A$",A, {pos="E"},"$B$",B, {pos="S"},"$C$",C, {pos="W"},"$D$",D, {pos="N"},
"$E$",E, {pos="S"}, "$I$", I[1],{pos="E",dist=0.1} )
g:Show()
\end{luadraw}
```

FIGURE 17 : Paramétrisation d'une spline



9) Chemins et clipping : Beginclip() et Endclip()

Un chemin peut être utilisé pour faire du clipping grâce à deux méthodes :

g:Beginclip(chemin 2D ou 3D [, inverse]) et **g:Endclip([draw_path, draw_options])**.

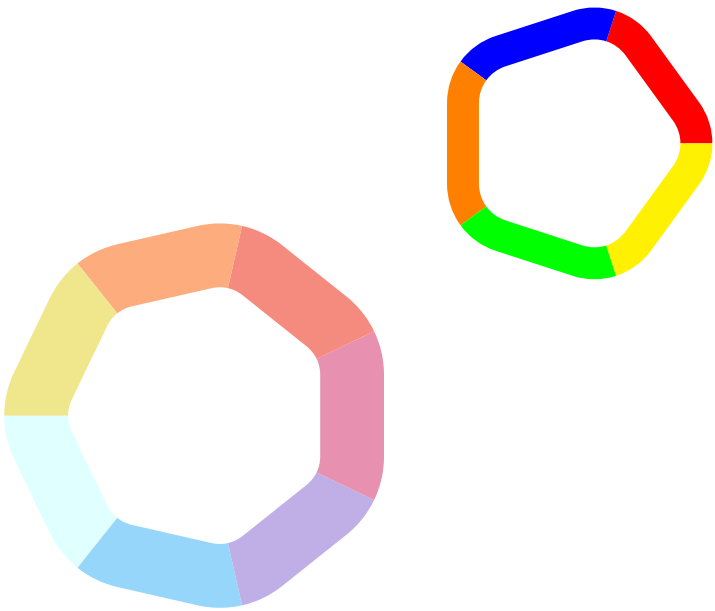
La première ouvre un groupe *scope* et passe le ⟨*chemin*⟩ comme argument à la fonction *\clip* de TikZ. L'argument ⟨*inverse*⟩ est un booléen qui vaut `false` par défaut, lorsqu'il a la valeur `true` le clipping est inversé, c'est à dire que seul ce qui est à l'extérieur du ⟨*chemin*⟩ sera dessiné, mais il faut pour cela que celui-ci soit dans le sens trigonométrique.

La seconde méthode referme le groupe *scope*, elle est indispensable (sinon il y aura une erreur de compilation). L'argument optionnel ⟨*draw_path*⟩ est un booléen (`false` par défaut), lorsqu'il a la valeur `true`, le chemin ⟨*chemin*⟩ est dessiné après la fermeture du groupe, et dans ce cas, l'argument optionnel ⟨*draw_options*⟩, qui doit être une chaîne de caractères (vide par défaut), est transmise à l'instruction *\draw*.

```
\begin{luadraw}{name=polygon_with_different_line_color_and_rounded_corners}
local ld = luadraw
local g = ld.graph:new{window={-5,5,-5,5},size={10,10}}
local i = ld.cpx.I
```

```
local Dcolored_polyreg = function(c,a,nb,r,wd,colors)
-- c=centre, a=sommet, nb=nombre de côtés, r=rayon, wd=épaisseur en point, colors=liste de couleurs
    local L = ld.polyreg(c,a,nb)
    table.append(L,{r,"cla"}) --polygone aux angles arrondis (radius=r)
    local angle = 360/nb
    local b = a
    for k = 1, nb do
        a = b; b = ld.rotate(a,angle,c)
        g:Beginclip({2*a-c,c,2*b-c,"l"}) -- définition d'un secteur angulaire pour clipper
        g:Dpath(L,"line width="..wd.."pt,color="..colors[k])
        g:Endclip()
    end
end
Dcolored_polyreg(3+2*i,5+2*i,5,0.8,12,{"red","blue","orange","green","yellow"}) -- pentagon
Dcolored_polyreg(-2.5-2*i,-5-2*i,7,1,24,ld.getpalette(ld.palGasFlame,7)) -- heptagon
g:Show()
\end{luadraw}
```

FIGURE 18 : Exemple de clipping



10) Axes et grilles

Variables globales utilisées pour les axes et les grilles :

- `ld.maxGrad = 100` : nombre max de graduations sur un axe.
- `ld.defaultlabelshift = 0.1875` : lorsqu'une grille est dessinée avec les axes (option `grid=true`) les labels sont automatiquement décalés le long de l'axe avec cette variable.
- `ld.defaultxylabelsep = 0` : définit la distance par défaut entre les labels et les graduations.
- `ld.defaultlegendsep = 0.2` : définit la distance par défaut entre la légende et l'axe.
- `ld.digits = 4` : nombre de décimales par défaut dans les conversions en chaînes de caractères, les 0 terminaux sont supprimés.
- `ld.dollar = true` : pour ajouter des dollars autour des labels des graduations.
- `ld.siunitx = false` : avec la valeur `true` les labels sont formatés avec la macro `\num{...}` du package *siunitx*, ce qui permet d'utiliser certaines options de ce package, comme remplacer le point décimal par une virgule en faisant :

```
\usepackage[local=FR]{siunitx}
```

ou bien en faisant :

```
\usepackage{siunitx}
```

```
\sisetup{output-decimal-marker={,}}
```

Pour les axes, en 2D comme en 3D, tous les labels sont formatés en chaînes de caractères avec la fonction **ld.num(x)**, celle-ci transforme le nombre x en une chaîne *str* avec le nombre de décimales fixées par la variable globale `ld.digits`, lorsque la variable `ld.siunitx` a la valeur `true`, la fonction renvoie `"\num{str}"`, sinon elle renvoie simplement *str*. Ceci vaut pour également pour les axes en 3D. Voici le code de cette fonction :

```
function ld.num(x) -- x is a real, returns a string
    local rep = ld.strReal(x) -- conversion to string with digits decimals max
    if ld.siunitx then rep = "\\num{"..rep.."}" end --needs \usepackage{siunitx}
    return rep
end
```

Daxes

Le tracé des axes s'obtient avec la méthode **g:Daxes([{A, xpas, ypas}, options])**.

- Le premier argument précise le point d'intersection des deux axes (c'est le complexe ⟨*A*⟩), le pas des graduations sur l'axe Ox (c'est ⟨*xpas*⟩) et le pas des graduations sur Oy (c'est ⟨*ypas*⟩). Par défaut le point ⟨*A*⟩ est l'origine Z(0, 0), et les deux pas sont égaux à 1.
- L'argument ⟨*options*⟩ est une table précisant les options possibles. Voici ces options avec leur valeur par défaut :
  - `showaxe={1,1}`. Cette option précise si les axes doivent être tracés ou pas (1 ou 0). La première valeur est pour l'axe Ox et la seconde pour l'axe Oy.
  - `arrows="-"`. Cette option permet d'ajouter ou non une flèche aux axes (pas de flèche par défaut, mettre `"->"` par exemple pour ajouter une flèche).
  - `limits={"auto","auto"}`. Cette option permet de préciser l'étendue des deux axes (première valeur pour Ox, seconde valeur pour Oy). La valeur "auto" signifie que c'est la droite en entier, mais on peut préciser les abscisses extrêmes, par exemple : `limits={{-4,4},"auto"}`.
  - `gradlimits={"auto","auto"}`. Cette option permet de préciser l'étendue des graduations sur les deux axes (première valeur pour Ox, seconde valeur pour Oy). La valeur `"auto"` signifie que c'est la droite en entier, mais on peut préciser les graduations extrêmes, par exemple : `gradlimits={{-4,4},{-2,3}}`.
  - `unit={"",""}`. Cette option permet de préciser de combien en combien vont les graduations sur les axes. La valeur par défaut signifie qu'il faut prendre la valeur du pas (⟨*xpas*⟩ sur Ox, ou ⟨*ypas*⟩ sur Oy), SAUF lorsque l'option `labeltext` n'est pas la chaîne vide, dans ce cas `unit` prend la valeur 1.
  - `nbsubdiv={0,0}`. Cette option permet de préciser le nombre de subdivisions entre deux graduations principales sur l'axe.
  - `tickpos={0.5,0.5}`. Cette option précise la position des graduations par rapport à chaque axe, ce sont deux nombres entre 0 et 1, la valeur par défaut de 0.5 signifie qu'ils sont centrés sur l'axe (0 et 1 représentent les extrémités).
  - `tickdir={"auto","auto"}`. Cette option indique la direction des graduations sur l'axe. Cette direction est un vecteur (complexe) non nul. La valeur par défaut `"auto"` signifie que les graduations sont orthogonales à l'axe.
  - `xyticks={0.2,0.2}`. Cette option précise la longueur des graduations sur l'axe.
  - `xylabelsep={0,0}`. Cette option précise la distance entre les labels et les graduations sur l'axe.
  - `originpos={"right","top"}`. Cette option précise la position du label à l'origine sur l'axe, les valeurs possibles sont : `"none"`, `"center"`, `"left"`, `"right"` pour Ox, et `"none"`, `"center"`, `"bottom"`, `"top"` pour Oy.
  - `originnum={A.re,A.im}`. Cette option précise la valeur au point origine des graduations (graduation numéro 0).

    La formule qui définit le label à la graduation numéro n est :

    `(originnum + unit*n)"labeltext"/labelden.`
  - `originloc=A`. Cette option précise le point origine des graduations.
  - `legend={"",""}`. Cette option permet de préciser une légende pour l'axe.
  - `legendpos={0.975,0.975}`. Cette option précise la position (entre 0 et 1) de la légende par rapport à chaque axe.
  - `legendsep={ld.defaultlegendsep,ld.defaultlegendsep}`. Cette option précise la distance entre la légende et l'axe. La légende est de l'autre côté de l'axe par rapport aux graduations.

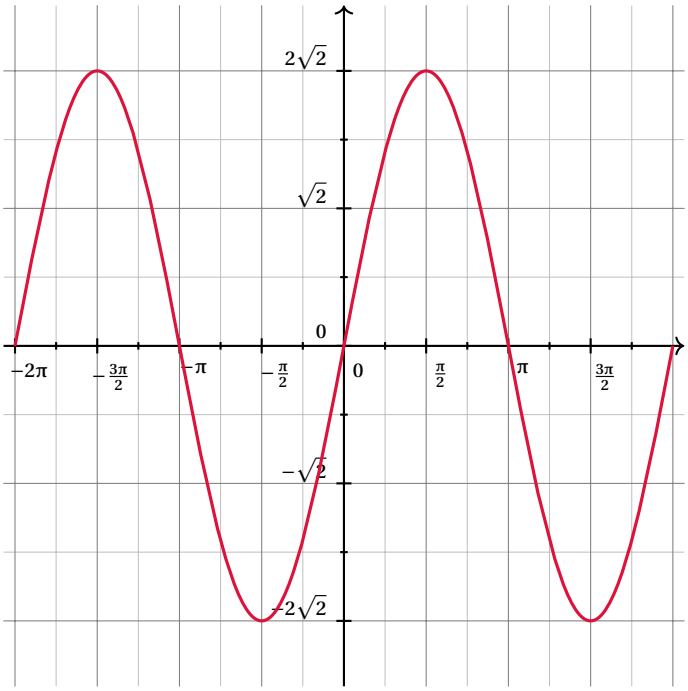
- `legendangle={"auto","auto"}`. Cette option précise l'angle (en degrés) que doit faire la légende pour l'axe. La valeur `"auto"` par défaut signifie que la légende doit être parallèle à l'axe si l'option `labelstyle` est aussi à `"auto"`, sinon la légende est horizontale.
- `legendstyle={"auto","auto"}`. Précise le style de label pour les légendes (position par rapport au point d'ancrage), avec la valeur `"auto"` celui-ci est déterminé automatiquement, sinon on peut utiliser les valeurs : `"center"`, `"N"`, `"NW"`, `"W"`, `"SW"`, `"S"`, `"SE"`, `"E"`, `"NE"`.
- `labelpos={"bottom","left"}`. Cette option précise la position des labels par rapport à l'axe. Pour l'axe Ox, les valeurs possibles sont : `"none"`, `"bottom"` ou `"top"`, pour l'axe Oy c'est : `"none"`, `"right"` ou `"left"`.
- `labelden={1,1}`. Cette option précise le dénominateur des labels (entier) pour l'axe. Rappelons que la formule qui définit le label à la graduation numéro n est :

  `(originnum + unit*n)"labeltext"/labelden.`
- `labeltext={"",""}`. Cette option définit le texte qui sera ajouté au numérateur des labels pour l'axe.
- `labelstyle={"S","W"}`. Cette option définit le style des labels pour chaque axe. Les valeurs possibles sont : `"auto"`, `"N"`, `"NW"`, `"W"`, `"SW"`, `"S"`, `"SE"`, `"E"`, `"NE"`.
- `labelangle={0,0}`. Cette option définit pour chaque axe l'angle des labels en degrés par rapport à l'horizontale.
- `labelcolor={"",""}`. Cette option permet de choisir une couleur pour les labels sur chaque axe. La chaîne vide représente la couleur par défaut.
- `labelshift={0,0}`. Cette option permet de définir un décalage systématique pour les labels sur l'axe (décalage de long de l'axe), avec l'option `grid=false` les valeurs par défaut sont 0 et 0, mais avec l'option `grid=true` on a par défaut `labelshift={ld.defaultlabelshift,ld.defaultlabelshift}` où `ld.defaultlabelshift` est une variable globale initialisée à 0.1875.
- `xynode_options=""`. Chaîne de caractères qui sera transmise telle quelle à l'instruction `\node{}` pour tous les labels sur les deux axes (mais pas pour les légendes).
- `xnode_options=xynode_options`. Chaîne de caractères qui sera transmise telle quelle à l'instruction `\node{}` pour tous les labels sur l'axe des x (mais pas pour la légende).
- `ynode_options=xynode_options`. Chaîne de caractères qui sera transmise telle quelle à l'instruction `\node{}` pour tous les labels sur l'axe des y (mais pas pour la légende).
- `nbdeci={2,2}`. Cette option précise le nombre de décimales pour les valeurs numériques sur l'axe.
- `use_siunitx={ld.siunitx,ld.siunitx}`. Cette option précise si les valeurs numériques doivent être formatées en utilisant le paquet *siunitx*, la valeur par défaut est celle de la variable globale `ld.siunitx` qui vaut `false` par défaut.
- `myxlabels=""`. Cette option permet d'imposer des labels personnels sur l'axe Ox. Lorsqu'il y en a, la valeur passée à l'option doit être une liste du type : `{pos1,"text1",pos2,"text2",…}`. Le nombre ⟨*pos1*⟩ représente une abscisse dans le repère `(A,xpas)`, ce qui correspond au point d'affixe `A+pos1*xpas`.
- `myylabels=""`. Cette option permet d'imposer des labels personnels sur l'axe Oy. Lorsqu'il y en a, la valeur passée à l'option doit être une liste du type : `{pos1,"text1",pos2,"text2",…}`. Le nombre ⟨*pos1*⟩ représente une abscisse dans le repère `(A,i*ypas)`, ce qui correspond au point d'affixe `A+pos1*ypas*i`.
- `grid=false`. Cette option permet d'ajouter ou non une grille.
- `showlines={true,true }`. Lorsque `grid=true`, cette option permet d'afficher ou non les traits verticaux de la grille (correspondant à l'axe des x, ainsi que les traits horizontaux de la grille (correspondant à l'axe des y).
- `drawbox=false`. Cette option de dessiner les axes sous la forme d'une boite, dans ce cas, les graduations sont sur le côté gauche et le côté bas.
- `gridstyle="solid"`. Cette option définit le style ligne pour la grille principale.
- `subgridstyle="solid"`. Cette option définit le style ligne pour la grille secondaire. Une grille secondaire apparaît lorsqu'il y a des subdivisions sur un des axes.
- `gridcolor="gray"`. Ceci définit la couleur de la grille principale.
- `subgridcolor="lightgray"`. Ceci définit la couleur de la grille secondaire.
- `gridwidth=4`. Épaisseur de trait de la grille principale (ce qui fait 0.4pt).
- `subgridwidth=2`. Épaisseur de trait de la grille secondaire (ce qui fait 0.2pt).

```
\begin{luadraw}{name=axes_grid}
local ld = luadraw
local g = ld.graph:new{window={-6.5,6.5,-3.5,3.5}, size={10,10,0}}
```

```
local i, pi, a = ld.cpx.I, math.pi, math.sqrt(2)
local f = function(x) return 2*a*math.sin(x) end
g:Labelsize("footnotesize"); g:Linewidth(8)
g:Daxes({0,pi/2,a},{labeltext={"\\pi","\\sqrt{2}"}, labelden={2,1},nbsubdiv={1,1},grid=true,arrows="->"})
g:Lineoptions("solid","Crimson",12); g:Dcartesian(f, {x={-2*pi,2*pi}})
g:Show()
\end{luadraw}
```

FIGURE 19 : Exemple avec axes avec grille



DaxeX et DaxeY

Les méthodes **g:DaxeX([{A, xpas}, options])** et **g:DaxeY([{A, ypas}, options])** permettent de tracer les axes séparément.

- Le premier argument précise le point servant d'origine (c'est le complexe $\langle A \rangle$) et le pas des graduations sur l'axe. Par défaut le point $\langle A \rangle$ est l'origine Z(0,0), et le pas est égal à 1.
- L'argument $\langle options \rangle$ est une table précisant les options possibles. Voici ces options avec leur valeur par défaut :
  - `showaxe=1`. Cette option précise si l'axe doit être tracé ou non (1 ou 0).
  - `arrows="-"`. Cette option permet d'ajouter ou non une flèche à l'axe (pas de flèche par défaut, mettre `"->"` pour ajouter une flèche).
  - `limits="auto"`. Cette option permet de préciser l'étendue des deux axes. La valeur "auto" signifie que c'est la droite en entier, mais on peut préciser les abscisses extrêmes, par exemple : `limits={-4,4}`.
  - `gradlimits="auto"`. Cette option permet de préciser l'étendue des graduations sur les deux axes. La valeur `"auto"` signifie que c'est la droite en entier, mais on peut préciser les graduations extrêmes, par exemple : `gradlimits={-2,3}`.
  - `unit=""`. Cette option permet de préciser de combien en combien vont les graduations sur l'axe. La valeur par défaut signifie qu'il faut prendre la valeur du pas, SAUF lorsque l'option `labeltext` n'est pas la chaîne vide, dans ce cas `unit` prend la valeur 1.
  - `nbsubdiv=0`. Cette option permet de préciser le nombre de subdivisions entre deux graduations principales.
  - `tickpos=0.5`. Cette option précise la position des graduations par rapport à l'axe, ce sont deux nombres entre 0 et 1, la valeur par défaut de 0.5 signifie qu'ils sont centrés sur l'axe. (0 et 1 représentent les extrémités).
  - `tickdir="auto"`. Cette option indique la direction des graduations sur l'axe. Cette direction est un vecteur (complexe) non nul. La valeur par défaut `"auto"` signifie que les graduations sont orthogonales à l'axe.
  - `xyticks=0.2`. Cette option précise la longueur des graduations.
  - `xylabelsep=0`. Cette option précise la distance entre les labels et les graduations.

- `originpos="center"`. Cette option précise la position du label à l'origine sur l'axe, les valeurs possibles sont : `"none"`, `"center"`, `"left"`, `"right"` pour Ox, et `"none"`, `"center"`, `"bottom"`, `"top"` pour Oy.
- `originnum=A.re` pour Ox et `originnum=A.im` pour Oy. Cette option précise la valeur de la graduation à l'origine (graduation numéro 0).

  La formule qui définit le label à la graduation numéro n est :

  ```
  (originnum + unit*n)"labeltext"/labelden.
  ```
- `legend=""`. Cette option permet de préciser une légende pour l'axe.
- `legendpos=0.975`. Cette option précise la position (entre 0 et 1) de la légende par rapport à l'axe.
- `legendsep=ld.defaultlegendsep`. Cette option précise la distance entre la légende et l'axe. La légende est de l'autre côté de l'axe par rapport aux graduations.
- `legendangle="auto"`. Cette option précise l'angle (en degrés) que doit faire la légende pour l'axe. La valeur "auto" par défaut signifie que la légende doit être parallèle à l'axe si l'option `labelstyle` est aussi à `"auto"`, sinon la légende est horizontale.
- `legendstyle="auto"`. Précise le style de label pour la légende (position par rapport au point d'ancrage), avec la valeur `"auto"` celui-ci est déterminé automatiquement, sinon on peut utiliser les valeurs : `"center"`, `"N"`, `"NW"`, `"W"`, `"SW"`, `"S"`, `"SE"`, `"E"`, `"NE"`.
- `labelpos="bottom"` pour Ox et `labelpos="left"` pour Oy. Cette option précise la position des labels par rapport à l'axe. Pour l'axe Ox, les valeurs possibles sont : `"none"`, `"bottom"` ou `"top"`, pour l'axe Oy c'est : `"none"`, `"right"` ou `"left"`.
- `labelden=1`. Cette option précise le dénominateur des labels (entier) pour l'axe. La formule qui définit le label à la graduation numéro n est :

  ```
  (originnum + unit*n)"labeltext"/labelden.
  ```
- `labeltext=""`. Cette option définit le texte qui sera ajouté au numérateur des labels.
- `labelstyle="S"` pour Ox et `labelstyle="W"` pour Oy. Cette option définit le style des labels. Les valeurs possibles sont : `"auto"`, `"N"`, `"NW"`, `"W"`, `"SW"`, `"S"`, `"SE"`, `"E"`, `"NE"`.
- `labelangle=0`. Cette option définit l'angle des labels en degrés par rapport à l'horizontale.
- `labelcolor=""`. Cette option permet de choisir une couleur pour les labels. La chaîne vide représente la couleur courante du texte.
- `labelshift=0`. Cette option permet de définir un décalage systématique pour les labels sur l'axe (décalage de long de l'axe).
- `nbdeci=2`. Cette option précise le nombre de décimales pour les labels numériques.
- `use_siunitx=ld.siunitx`. Cette option précise si les valeurs numériques doivent être formatées en utilisant le paquet *siunitx*, la valeur par défaut est celle de la variable globale `ld.siunitx` qui vaut `false` par défaut.
- `mylabels=""`. Cette option permet d'imposer des labels personnels. Lorsqu'il y en a, la valeur passée à l'option doit être une liste du type : `{pos1,"text1",pos2,"text2",…}`. Le nombre ⟨*pos1*⟩ représente une abscisse dans le repère `(A,xpas)` pour Ox, ou `(A,ypas*i)` pour Oy, ce qui correspond au point d'affixe `A+pos1*xpas` pour Ox, et `A+pos1*ypas*i` pour Oy.

Dgradline

Les méthodes de tracé des axes s'appuient sur la méthode **g:Dgradline({A, u}, options)**, où ⟨*{A,u}*⟩ représente la droite passant par ⟨A⟩ (un complexe) et dirigé par le vecteur ⟨u⟩ (un complexe non nul), le couple (A, u) sert de repère sur cette droite (et oriente cette droite), donc chaque point M de cette droite a une abscisse x telle M = A + $x\,u$. Cette méthode permet de dessiner cette droite graduée, l'argument ⟨*options*⟩ est une table précisant les options possibles, qui sont (avec leur valeur par défaut) :

- `showaxe=1`. Cette option précise si l'axe doit être tracé ou non (1 ou 0).
- `arrows="-"`. Cette option permet d'ajouter ou non une flèche à l'axe (pas de flèche par défaut, mettre `"->"` pour ajouter une flèche).
- `limits="auto"`. Cette option permet de préciser l'étendue des deux axes. La valeur `auto"` signifie que c'est la droite en entier, mais on peut préciser les abscisses extrêmes, par exemple : `limits={-4,4}`.
- `gradlimits="auto"`. Cette option permet de préciser l'étendue des graduations sur les deux axes. La valeur `"auto"` signifie que c'est la droite en entier, mais on peut préciser les graduations extrêmes, par exemple : `gradlimits={-`

2,3}.

- unit=1. Cette option permet de préciser de combien en combien vont les graduations sur l'axe.
- nbsubdiv=0. Cette option permet de préciser le nombre de subdivisions entre deux graduations principales.
- tickpos=0.5. Cette option précise la position des graduations par rapport à l'axe, ce sont deux nombres entre 0 et 1, la valeur par défaut de 0.5 signifie qu'ils sont centrés sur l'axe. (0 et 1 représentent les extrémités).
- tickdir="auto". Cette option indique la direction des graduations sur l'axe. Cette direction est un vecteur (complexe) non nul. La valeur par défaut "auto" signifie que les graduations sont orthogonales à l'axe.
- xyticks=0.2. Cette option précise la longueur des graduations.
- xylabelsep=defaultxylabelsep. Cette option précise la distance entre les labels et les graduations, defaultxylabelsep est une variable globale valant 0 par défaut.
- originpos="center". Cette option précise la position du label à l'origine sur l'axe, les valeurs possibles sont : "none", "center", "left", "right".
- originnum=0. Cette option précise la valeur de la graduation à l'origine A (graduation numéro 0).

  La formule qui définit le label à la graduation numéro n (au point A + nu) est :

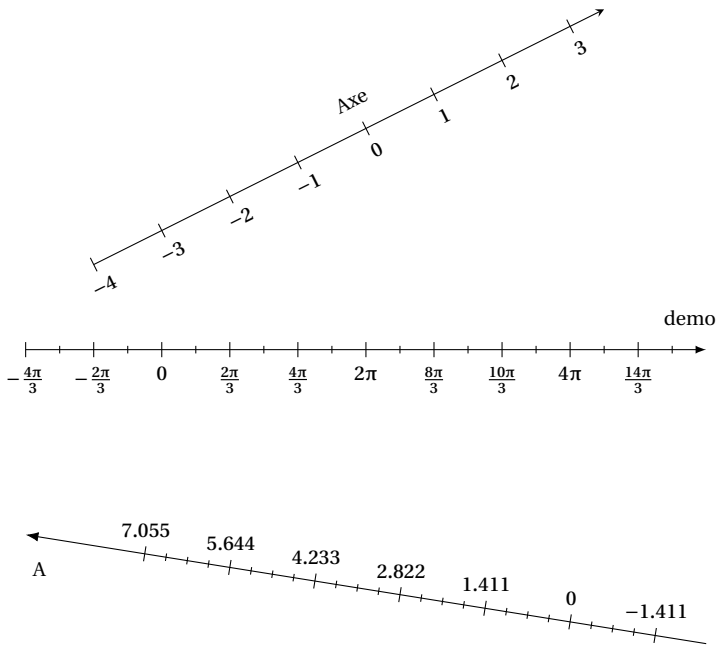
  (originnum + unit*n)"labeltext"/labelden.
- legend="". Cette option permet de préciser une légende pour l'axe.
- legendpos=0.975. Cette option précise la position (entre 0 et 1) de la légende par rapport à l'axe.
- legendsep=ld.defaultlegendsep. Cette option précise la distance entre la légende et l'axe. La légende est de l'autre côté de l'axe par rapport aux graduations, ld.defaultlegendsep est une variable globale qui vaut 0.2 par défaut.
- legendangle="auto". Cette option précise l'angle (en degrés) que doit faire la légende pour l'axe. La valeur "auto" par défaut signifie que la légende doit être parallèle à l'axe si l'option labelstyle est aussi à "auto", sinon la légende est horizontale.
- legendstyle="auto". Précise le style de label pour la légende (position par rapport au point d'ancrage), avec la valeur "auto" celui-ci est déterminé automatiquement, sinon on peut utiliser les valeurs : "center", "N", "NW", "W", "SW", "S", "SE", "E", "NE".
- labelpos="bottom". Cette option précise la position des labels par rapport à l'axe, les valeurs possibles sont : "none", "bottom", or "top". Cette position détermine en même temps celle de la légende : de l'autre côté de l'axe.
- labelden=1. Cette option précise le dénominateur des labels (entier) pour l'axe. La formule qui définit le label à la graduation numéro n est :

  (originnum + unit*n)"labeltext"/labelden.
- labeltext="". Cette option définit le texte qui sera ajouté au numérateur des labels.
- labelstyle="auto". Cette option définit le style des labels. Les valeurs possibles sont : "auto", "N", "NW", "W", "SW", "S", "SE", "E", "NE".
- labelangle=0. Cette option définit l'angle des labels en degrés par rapport à l'horizontale.
- labelcolor="". Cette option permet de choisir une couleur pour les labels. La chaîne vide représente la couleur courante du texte.
- labelshift=0. Cette option permet de définir un décalage systématique pour les labels sur l'axe (décalage de long de l'axe).
- nbdeci=2. Cette option précise le nombre de décimales pour les labels numériques.
- use_siunitx=ld.siunitx. Cette option précise si les valeurs numériques doivent être formatées en utilisant le paquet *siunitx*, la valeur par défaut est celle de la variable globale ld.siunitx qui vaut false par défaut.
- mylabels="". Cette option permet d'imposer des labels personnels. Lorsqu'il y en a, la valeur passée à l'option doit être une liste du type : {x1,"text1",x2,"text2",…}. Les nombres $\langle x1\rangle$, $\langle x2\rangle$, …, représentent des abscisses dans le repère (A, u).

```
\begin{luadraw}{name=gradline}
local ld = luadraw
local g = ld.graph:new{window={-5,5,-5,5},size={10,10}}
g:Labelsize("footnotesize")
local i = ld.cpx.I
g:Dgradline({3.25*i,1+i/2}, {limits={-4,4}, legend="Axe", legendpos=0.5, arrows="-stealth"})
g:Dgradline({-3,1}, {legend="demo", labeltext="\\pi", labelden=3, unit=2, nbsubdiv=1, arrows="-latex"})
g:Dgradline({3-4*i,-1.25+i/5}, {legend="A", labelstyle="N", gradlimits={-1,5},
```

```
nbsubdiv=3, unit=1.411, nbdeci=3, arrows="-Latex"})
g:Show()
\end{luadraw}
```

FIGURE 20 : Exemples de droites graduées



Dgrid

La méthode **g:Dgrid({A, B}, options** permet le dessin d'une grille.

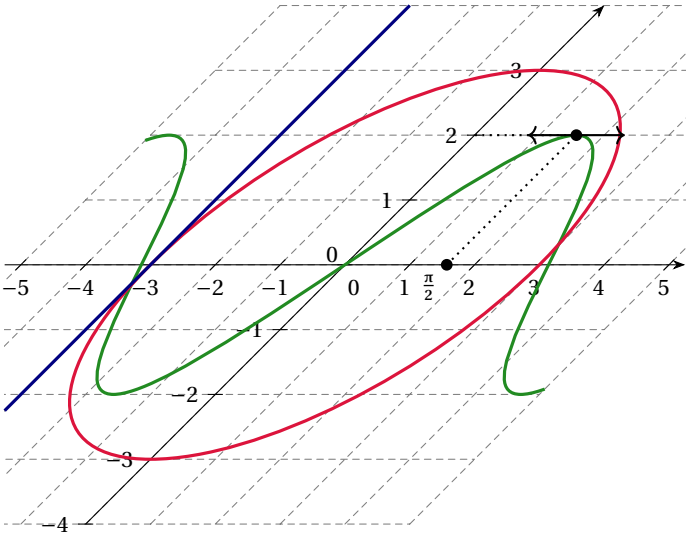
- Le premier argument est obligatoire, il précise le coin inférieur gauche (c'est le complexe ⟨*A*⟩), le coin supérieur droit (c'est le complexe ⟨*B*⟩) de la grille.
- L'argument ⟨*options*⟩ est une table précisant les options possibles. Voici ces options avec leur valeur par défaut :
  - `unit={1,1}`. Cette option définit les unités sur les axes pour la grille principale.
  - `showlines={true,true }`. Cette option permet d'afficher ou non les traits verticaux de la grille (correspondant à l'axe des x, ainsi que les traits horizontaux de la grille (correspondant à l'axe des y).
  - `gridwidth=4`. Cette option définit l'épaisseur du trait de la grille principale (0.4pt par défaut).
  - `gridcolor="gray"`. Couleur grille de la grille principale.
  - `gridstyle="solid"`. Style de trait pour la grille principale.
  - `nbsubdiv={0,0}`. Nombre de subdivisions (pour chaque axe) entre deux traits de la grille principale. Ces subdivisions déterminent la grille secondaire.
  - `subgridcolor="lightgray"`. Couleur de la grille secondaire.
  - `subgridwidth=2`. Épaisseur du trait de la grille secondaire (0.2pt par défaut).
  - `subgridstyle="solid"`. Style de trait pour la grille secondaire.
  - `originloc=A`. Localisation de l'origine de la grille.

Exemple : il est possible de travailler dans un repère non orthogonal. Voici un exemple où l'axe Ox est conservé, mais la première bissectrice devient le nouvel axe Oy, on modifie pour cela la matrice de transformation du graphe. À partir de cette modification les affixes représentent les coordonnées dans le nouveau repère.

```
\begin{luadraw}{name=axes_non_ortho}
local ld = luadraw
local g = ld.graph:new{window={-5.25,5.25,-4,4},size={10,10}}
local i, pi, Z = ld.cpx.I, math.pi, ld.cpx.Z
local f = function(x) return 2*math.sin(x) end
g:Setmatrix({0,1,1+i}); g:Labelsize("small")
```

```
g:Dgrid({-5-4*i,5+4*i},{gridstyle="dashed"})
g:Daxes({0,1,1}, {arrows="-Stealth"})
g:Lineoptions("solid","ForestGreen",12); g:Dcartesian(f,{x={-5,5}})
g:Dcircle(0,3,"Crimson")
g:DlineEq(1,0,3,"Navy") -- droite d'équation x=-3
g:Lineoptions("solid","black",8); g:DtangentC(f,pi/2,1.5,"<->")
g:Dpolyline({pi/2,pi/2+2*i,2*i},"dotted")
g:Ddots(Z(pi/2,2))
g:Dlabeldot("$\\frac{\\pi}2$",pi/2,{pos="SW"})
g:Show()
\end{luadraw}
```

FIGURE 21 : Exemple de repère non orthogonal



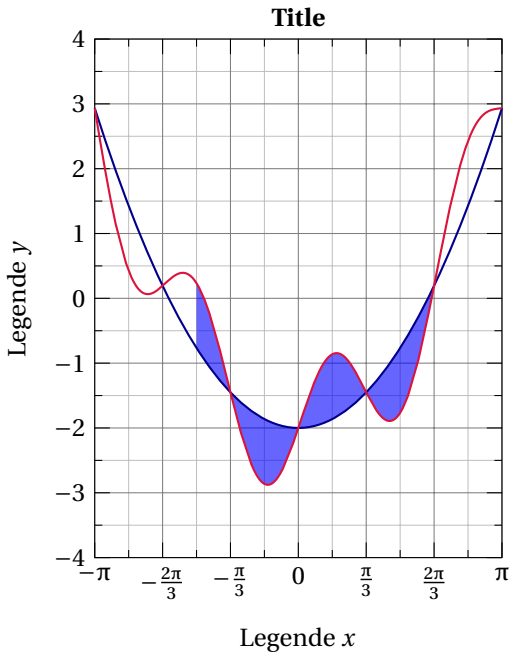
Dgradbox

La méthode **g:Dgradbox({A, B, xpas, ypas}, options)** permet le dessin d'une boîte graduée.

- Le premier argument est obligatoire, il précise le coin inférieur gauche (c'est le complexe ⟨*A*⟩) et le coin supérieur droit (c'est le complexe ⟨*B*⟩) de la boîte, ainsi que le pas sur chaque axe.
- L'argument ⟨*options*⟩ est une table précisant les options possibles. Ce sont les mêmes que pour les axes, mises à part certaines valeurs par défaut. À celles-ci s'ajoute l'option suivante : `title=""` qui permet d'ajouter un titre en haut de la boite, attention cependant à laisser suffisamment de place pour cela.

```
\begin{luadraw}{name=gradbox}
local ld = luadraw
local g = ld.graph:new{window={-5,4,-5.5,5},size={10,10}}
local i, pi = ld.cpx.I, math.pi
local h = function(x) return x^2/2-2 end
local f = function(x) return math.sin(3*x)+h(x) end
g:Dgradbox({-pi-4*i,pi+4*i,pi/3,1},{grid=true,originloc=0, originnum={0,0}, labeltext={"\\pi",""},
labelden={3,1}, title="\\textbf{Title}", legend={"Legende $x$","Legende $y$"}})
g:Saveattr(); g:Viewport(-pi,pi,-4,4) -- on limite la vue (clip)
g:Filloptions("full","blue",0.6); g:Linestyle("noline"); g:Ddomain2(f,h,{x={-pi/2,2*pi/3}})
g:Filloptions("none",nil,1); g:Lineoptions("solid",nil,8); g:Dcartesian(h,{x={-pi,pi}, draw_options="DarkBlue"})
g:Dcartesian(f,{x={-pi,pi},draw_options="Crimson"})
g:Restoreattr()
g:Show()
\end{luadraw}
```

FIGURE 22 : Utilisation de Dgradbox



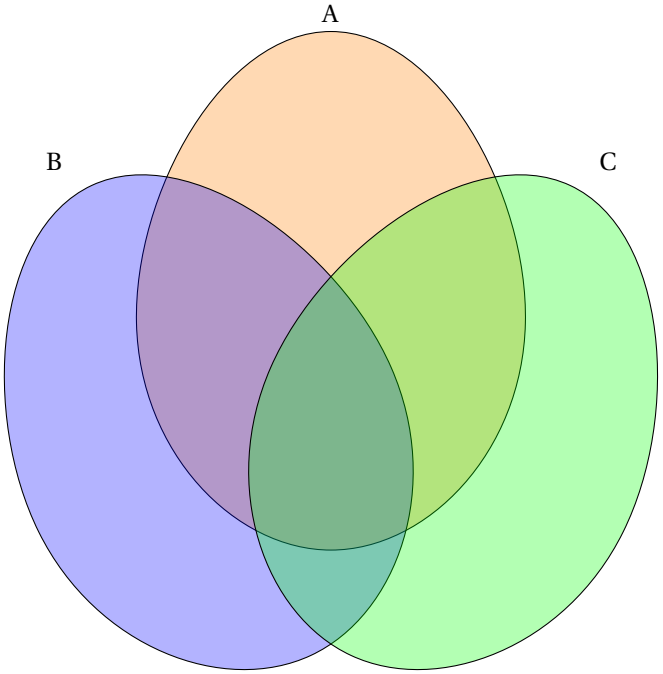
11) Dessins d'ensembles (diagrammes de Venn)

Dessiner un ensemble

La fonction **ld.set(center [, angle, scale])** renvoie un chemin représentant un ensemble (en forme d'œuf), dont le centre est ⟨*center*⟩ (complexe), l'argument ⟨*angle*⟩ représente l'inclinaison (en degrés) de l'axe vertical de l'ensemble (0 par défaut), et l'argument ⟨*scale*⟩ est un facteur d'échelle permettant de modifier la taille de l'ensemble (1 par défaut). Un tel chemin peut être dessiné avec la méthode **g:Dpath()**.

```
\begin{luadraw}{name=set}
local ld = luadraw
local g = ld.graph:new{window={-5.25,5.25,-5,5},size={10,10}}
local i = ld.cpx.I
local A, B, C = ld.set(i,0), ld.set(-2-i,25), ld.set(2-i,-25)
g:Fillopacity(0.3)
g:Dpath(A,"fill=orange"); g:Dpath(B,"fill=blue")
g:Dpath(C,"fill=green")
g:Fillopacity(1)
g:Dlabel("$A$",5*i,{pos="N"},"$B$",-4+3*i,{pos="W"},"$C$",4+3*i,{pos="E"})
g:Show()
\end{luadraw}
```

FIGURE 23 : Dessiner un ensemble



Opérations sur les ensembles

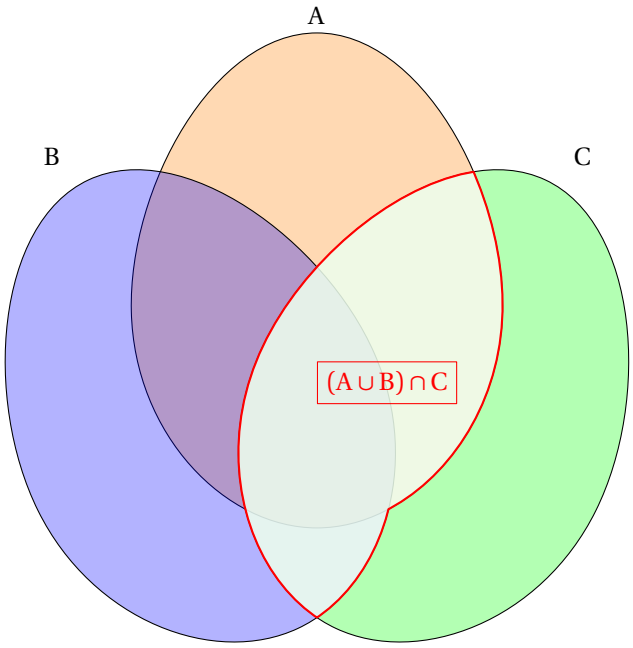
Notons C_1 et C_2 deux listes de complexes représentant le contour de deux ensembles (courbes fermées simples, d'un seul tenant). Les opérations possibles sont au nombre de trois :

- La fonction **ld.cap(C1, C2)** renvoie une liste de complexes représentant le contour de l'intersection des ensembles correspondant à C_1 et C_2.
- La fonction **ld.cup(C1, C2)** renvoie une liste de complexes représentant le contour de la réunion des ensembles correspondant à C_1 et C_2.
- La fonction **ld.setminus(C1, C2)** renvoie une liste de complexes représentant le contour de la différence des ensembles correspondant à C_1 et C_2 ($C_1 \setminus C_2$).

Le résultat de ces opérations, étant une liste de complexes, peut être dessiné avec la méthode **g:Dpolyline()**.

```
\begin{luadraw}{name=cap_and_cup}
local ld = luadraw
local g = ld.graph:new{window={-5.5,5.5,-5,5},size={10,10}}
local i = ld.cpx.I
local A, B, C = ld.set(i,0), ld.set(-2-i,25), ld.set(2-i,-25)
g:Fillopacity(0.3)
g:Dpath(A,"fill=orange"); g:Dpath(B,"fill=blue"); g:Dpath(C,"fill=green")
g:Fillopacity(1)
local C1, C2, C3 = ld.path(A), ld.path(B), ld.path(C) -- conversion: chemin -> liste de complexes
local I = ld.cap(ld.cup(C1,C2),C3)
g:Linecolor("red"); g:Filloptions("full","white")
g:Dpolyline(I,true,"line width=0.8pt,fill opacity=0.8")
g:Dlabel("$A$",5*i,{pos="N"},"$B$",-4+3*i,{pos="W"},"$C$",4+3*i,{pos="E"},
"$(A\\cup B) \\cap C$",-i,{pos="NE",node_options="red,draw"})
g:Show()
\end{luadraw}
```

FIGURE 24 : Opérations sur les ensembles



NB : le résultat n'est pas toujours satisfaisant lorsque les contours deviennent trop complexes, ou lorsque les contours ont des tronçons en commun.

12) Importer une image

Placer une image dans le graphique

Pour importer une image dans le graphique en cours (et éventuellement dessiner par dessus), il y a la méthode :

g:Dimage(filename, anchor, options)

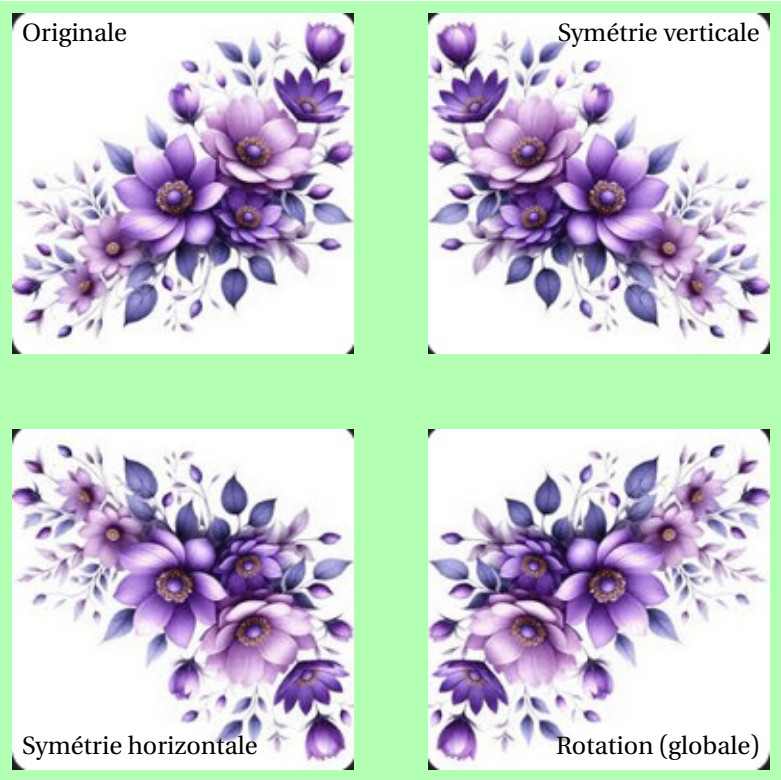
- ⟨*filename*⟩ est le nom complet du fichier image, celle-ci sera affichée par la commande `\includegraphics[]{}` dans un node,
- ⟨*anchor*⟩ est un nombre complexe représentant le point d'ancrage de l'image dans le graphique,
- ⟨*options*⟩ est une table dont les champs définissent les paramètres d'affichage, qui sont (avec leur valeur par défaut) :
  - `pos="center"` qui indique la position de l'image par rapport au point d'ancrage, sur le même principe que pour les labels. Les valeurs possibles sont : `"center"`, `"N"`, `"NE"`, `"E"`, `"SE"`, `"S"`, `"SW"`, `"W"`, `"NW"`.
  - `name=""` qui permet éventuellement de donner un nom au node qui va être créé (nom qui pourra être utilisé par TikZ ensuite),
  - `graphics_options=""` qui est une chaîne contenant des options pour la commande `\includegraphics` (aucune option par défaut),
  - `matrix=nil` qui est une table représentant une matrice de transformation affine, celle-ci sera appliquée localement au node créé (pour le node, l'origine est le point d'ancrage). Il est à noter que la matrice 2D de transformation globale du graphique en cours est également prise en compte par cette méthode,
  - `node_options=""` qui est une chaîne contenant des options pour la commande `\node`, celles-ci viendront s'ajouter à la suite des options par défaut comme ceci :

    `\node[line width=0.3pt,inner sep=-0.15pt,cm=<matrice>,<position>,<node_options>].`

```
\begin{luadraw}{name=Dimage}
local ld = luadraw
local Z, i = ld.cpx.Z, ld.cpx.I
local g = ld.graph:new{margin={0,0,0,0}, size={10,10},bg="green!30"}
local filename = ld.cachedir.."flower.jpg"
local g_options = "width=4.5cm,height=4.5cm"
g:Dimage(filename,Z(-5,5),{pos="SE", graphics_options=g_options})
g:Dimage(filename,Z(5,5),{pos="SE", matrix={0,-1,i}, graphics_options=g_options})
g:Rotate(180)
```

```
g:Dimage(filename,Z(-5,5),{pos="SE", graphics_options=g_options})
g:IDmatrix()
g:Dimage(filename,Z(-5,-5),{pos="SE", matrix={0,1,-i}, graphics_options=g_options})
g:Dlabel( "Originale", Z(-5,5),{pos="SE"},
"Symétrie verticale", Z(5,5), {pos="SW"},
"Rotation (globale)", Z(5,-5), {pos="NW"},
"Symétrie horizontale", Z(-5,-5), {pos="NE"} )
g:Show()
\end{luadraw}
```

FIGURE 25 : Importer une image



Mapper une image sur un parallélogramme

Ceci est possible avec la méthode : **g:Dmapimage(filename, parallelo, options)** (qui fait appel à la méthode **g:Dimage()**), où :

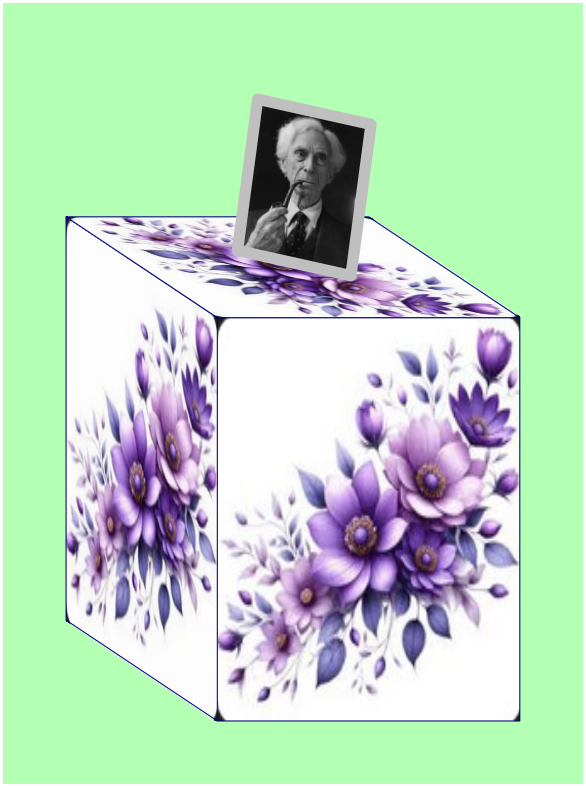
- ⟨*filename*⟩ est le nom complet du fichier image,
- ⟨*parallelo*⟩ est une liste de la forme $\{a,u,v\}$ constituée de trois nombres complexes, cette liste représente le parallélogramme de sommets $\{a, a+u, a+u+v, a+v\}$.
- ⟨*options*⟩ est une table dont les champs définissent les paramètres, qui sont (avec leur valeur par défaut) :
  - `name=""` qui permet éventuellement de donner un nom au node qui va être créé (nom qui pourra être utilisé par TikZ ensuite).
  - `clip=false` qui est un booléen indiquant si l'image doit être clippée avec le parallélogramme, mais normalement cela n'est pas nécessaire.
  - `border_options=nil`, lorsque cette option n'est pas égale à `nil`, ce doit être une chaîne de caractères contenant les options de dessin le contour du parallélogramme, `border_options` sera alors transmise directement à l'instruction *\draw* pour dessiner le contour,
  - `graphics_options=""` qui est une chaîne des options pour la commande `\includegraphics`, celles-ci viendront s'ajouter à la suite des options par défaut comme ceci :

    `\includegraphics[width=1cm,height=1cm, <graphics_options>]`
  - `node_options=""` qui est une chaîne contenant des options pour la commande `\node`, celles-ci viendront s'ajouter à la suite des options par défaut comme pour **g:Dimage()**.

```
\begin{luadraw}{name=Dmapimage}
local ld = luadraw
```

```
local g = ld.graph:new{window={-2,3.5,-0.5,7}, margin={0,0,0,0}, size={10,10},bg="green!30"}
local filename = ld.cachedir.."flower.jpg"
local i = ld.cpx.I
local a, u, v, w = 0, 3, -1.5+i, 4*i
local facets = {{a,u,w}, {a+v,-v,w}, {a+v+w,-v,u} }
local portrait = {a+w+(2*v+u)/3+0.2, (u-v)/4, 1.5*i+0.2}
for _,f in ipairs(facets) do
    g:Dmapimage(filename,f,{border_options="Navy"})
end
g:Dmapimage(ld.cachedir.."russell.jpg", portrait, {border_options="lightgray,line width=1.5mm"})
g:Show()
\end{luadraw}
```

FIGURE 26 : Mapper une image



13) Les couleurs

Dans l'environnement *luadraw* les couleurs sont des chaînes de caractères qui doivent correspondre à des couleurs connues de TikZ. Le package *xcolor* est fortement conseillé pour ne pas être limité aux couleurs de bases.

Calculs sur les couleurs

Afin de pouvoir faire des manipulations sur les couleurs, celles-ci ont été définies (dans le module *luadraw_colors.lua*) sous la forme de tables de trois composantes : rouge, vert, bleu, chaque composante étant un nombre entre 0 et 1, et avec leur nom au format *svgnames* du package *xcolor*, par exemple on y trouve (entre autres) les déclarations :

```
local ld = luadraw
ld.AliceBlue = {0.9412, 0.9725, 1}
ld.AntiqueWhite = {0.9804, 0.9216, 0.8431}
ld.Aqua = {0.0, 1.0, 1.0}
ld.Aquamarine = {0.498, 1.0, 0.8314}
```

On pourra se référer à la documentation de *xcolor* pour avoir la liste de ces couleurs.

Pour utiliser celles-ci dans l'environnement *luadraw*, on peut :

- soit les utiliser avec leur nom si on a déclaré dans le préambule : `\usepackage[svgnames]{xcolor}`, par exemple : `g:Linecolor("AliceBlue")`,

- soit les utiliser avec la fonction **ld.rgb()** de *luadraw*, par exemple : `g:Linecolor(ld.rgb(AliceBlue))`. Par contre, avec cette fonction **ld.rgb()**, pour changer localement de couleur il faut faire comme ceci (exemple) : `g:Dpolyline(L,"color="..ld.rgb(AliceBlue))`, ou `g:Dpolyline(L,"fill="..ld.rgb(AliceBlue))`. Car la fonction **rgb()** ne renvoie pas un nom de couleur, mais une définition de couleur.

Fonctions pour la gestion des couleurs :

- La fonction **ld.rgb(r, g, b)** ou **ld.rgb({r, g, b})**, renvoie la couleur sous forme d'une chaîne de caractères compréhensible par TikZ dans les options `color=...` et `fill=...`. Les valeurs de r, g et b doivent être entre 0 et 1.
- La fonction **ld.hsb(h, s, b [, table])** renvoie la couleur sous forme d'une chaîne de caractères compréhensible par TikZ. L'argument ⟨*h*⟩ (hue) doit être un nombre enter 0 et 360, l'argument ⟨*s*⟩ (saturation) doit être entre 0 et 1, et l'argument ⟨*b*⟩ (brightness) doit être aussi entre 0 et 1. L'argument (facultatif) ⟨*table*⟩ est un booléen (`false` par défaut) qui indique si le résultat doit être renvoyé sous forme de table $\{r,g,b\}$ ou non (par défaut c'est sous forme d'une chaîne).
- La fonction **ld.mixcolor(color1, proportion1, color2, proportion2, ..., colorN, proportionN)** mélange les couleurs ⟨*color1*⟩, ..., ⟨*colorN*⟩ dans les proportions demandées et renvoie la couleur qui en résulte sous forme d'une chaîne de caractères compréhensible par TikZ, suivie de cette même couleur sous forme de table $\{r,g,b\}$. Chacune des couleurs doit être une table de trois composantes $\{r,g,b\}$.
- La fonction **ld.mixpalette(pal, percent, color)** renvoie une nouvelle palette après avoir mixer chaque couleur de la palette ⟨*pal*⟩ avec ⟨*color*⟩. L'argument ⟨*pal*⟩ est une liste de couleurs, chacune d'elles étant une table de trois composantes $\{r,g,b\}$, l'argument ⟨*percent*⟩ est un nombre entre 0 et 100, et l'argument ⟨*color*⟩ est une couleur sous forme de table $\{r,g,b\}$.
- La fonction **ld.palette(colors, pos [, table])** : l'argument ⟨*colors*⟩ est une liste (table) de couleurs au format $\{r,g,b\}$, l'argument ⟨*pos*⟩ est un nombre entre 0 et 1, la valeur 0 correspond à la première couleur de la liste et la valeur 1 à la dernière. La fonction calcule et renvoie la couleur correspondant à la position ⟨*pos*⟩ dans la liste par interpolation linéaire. L'argument (facultatif) ⟨*table*⟩ est un booléen (`false` par défaut) qui indique si le résultat doit être renvoyé sous forme de table $\{r,g,b\}$ ou non (par défaut c'est sous forme d'une chaîne).
- La fonction **ld.getpalette(colors, nb [, table])** : l'argument ⟨*colors*⟩ est une liste (table) de couleurs au format $\{r,g,b\}$, l'argument ⟨*nb*⟩ indique le nombre de couleurs souhaité. La fonction renvoie une liste de ⟨*nb*⟩ couleurs régulièrement réparties dans ⟨*colors*⟩. L'argument (facultatif) ⟨*table*⟩ est un booléen (`false` par défaut) qui indique si les couleurs sont renvoyées sous forme de tables $\{r,g,b\}$ ou non (par défaut c'est sous forme de chaînes).
- La méthode **g:Newcolor(name, color)** permet de définir dans l'export TikZ au format rgb une nouvelle couleur dont le nom sera ⟨*name*⟩ (chaîne), l'argument ⟨*color*⟩ peut être soit une table de trois composantes : rouge, vert, bleu (entre 0 et 1) définissant cette couleur, soit une chaîne de caractères représentant une couleur. Dans le premier cas la méthode utilise un `\definecolor` et dans le second cas elle utilise un `\colorlet`.

On peut également utiliser toutes les possibilités habituelles de TikZ pour la gestion des couleurs.

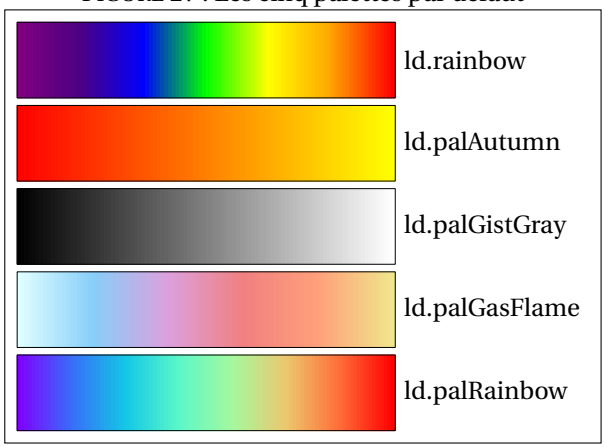
Par défaut, il y a cinq palettes de couleurs[2].

```
\begin{luadraw}{name=palettes}
local ld = luadraw
local Z = ld.cpx.Z
local g = ld.graph:new{window={-5,5,-5,5},bbox=false, border=true}
g:Linewidth(1)
local Dpalette = function(pal,A,h,L,N,name)
    local dl = L/N
    for k = 1, N do
        local color = ld.palette(pal,(k-1)/(N-1))
        g:Drectangle(A,A+h,A+h+dl,"color="..color..",fill="..color)
        A = A+dl
    end
    g:Drectangle(A,A+h,A+h-L); g:Dlabel(name,A+h/2,{pos="E"})
end
local A, h, dh, L, N = Z(-5,4), Z(0,-1), Z(0,-1.1), 5, 100
Dpalette(ld.rainbow,A,h,L,N,"ld.rainbow"); A = A+dh
Dpalette(ld.palAutumn,A,h,L,N,"ld.palAutumn"); A = A+dh
```

2. Une palette est une table de couleurs, celles-ci sont elle-mêmes des tables de nombres entre 0 et 1 représentant les composantes rouge, vert, bleu.

```
Dpalette(ld.palGistGray,A,h,L,N,"ld.palGistGray"); A = A+dh
Dpalette(ld.palGasFlame,A,h,L,N,"ld.palGasFlame"); A = A+dh
Dpalette(ld.palRainbow,A,h,L,N,"ld.palRainbow")
g:Show()
\end{luadraw}
```

FIGURE 27 : Les cinq palettes par défaut



III Constructions géométriques

Dans cette section sont regroupées les fonctions construisant des figures géométriques sans méthode graphique dédiée correspondante.

1) circumcircle(), incircle()

- La fonction **ld.circumcircle(a, b, c)** (ou **ld.circumcircle({a, b, c})**), où $\langle a\rangle$, $\langle b\rangle$ et $\langle c\rangle$ sont trois points (trois nombres complexes), renvoie le cercle circonscrit au triangle formé par ces trois points, sous la forme d'une séquence : C, r, où C est le centre du cercle (nombre complexe), r son rayon.
- La fonction **ld.incircle(a, b, c)** (ou **ld.incircle({a, b, c})**), où $\langle a\rangle$, $\langle b\rangle$, et $\langle c\rangle$ sont trois points (trois nombres complexes), renvoie le cercle inscrit dans triangle formé par ces trois points, sous la forme d'une séquence : C, r, où C est le centre du cercle (nombre complexe), r son rayon.

2) cvx_hull2d()

La fonction **ld.cvx_hull2d(L)** où $\langle L\rangle$ est une liste de complexes, calcule et renvoie une liste de complexes représentant l'enveloppe convexe de L.

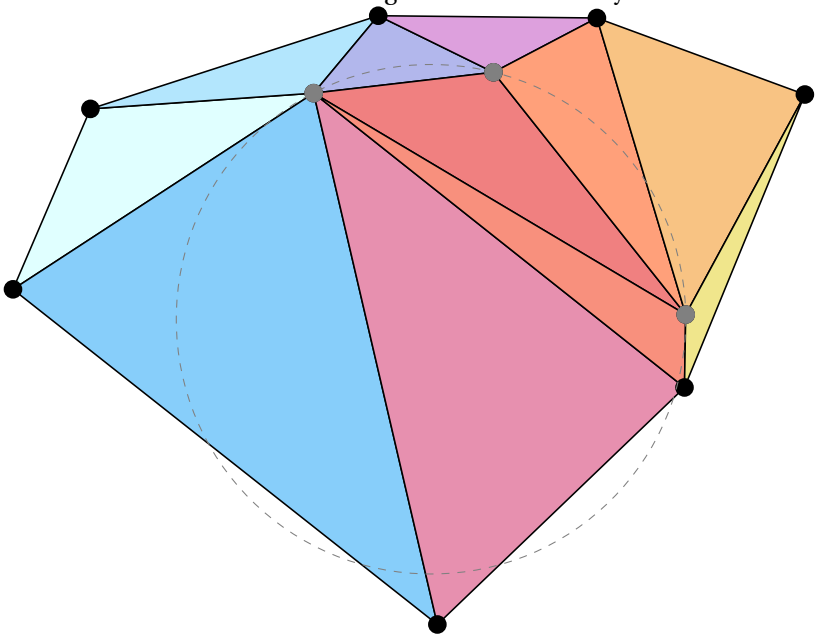
3) delaunay()

La fonction **ld.delaunay(L)** où $\langle L\rangle$ est une liste de nombres complexes **distincts**, renvoie une liste de triangles (un triangle étant une liste de trois nombres complexes) obtenus par triangulation de Delaunay des points de $\langle L\rangle$ (le cercle circonscrit de chacun des triangles ne contient aucun des autres points).

```
\begin{luadraw}{name=delaunay}
local ld = luadraw
local g = ld.graph:new{bbox=false, pictureoptions="scale=2"}
local i = ld.cpx.I; g:Linewidth(6)
local L = {0.285+1.46*i,1.556-0.142*i,2.344+1.313*i,-2.38+1.218*i,1.548-0.624*i,
0.969+1.819*i, -0.086-2.191*i,-0.477+1.834*i,-0.904+1.322*i,-2.892+0.025*i}
local T = ld.delaunay(L) -- liste de triangles
local n = #T
local num = 7 --on choisit un triangle
local colors = ld.getpalette(ld.palGasFlame,n)
for k = 1, n do
    g:Dpolyline(T[k],true,'fill='..colors[k])
```

```
    end
g:Ddots(L)
g:Dcircle( {ld.circumcircle(T[num])}, "line width=0.4pt,gray,dashed" )
g:Ddots(T[num],"gray")
g:Show()
\end{luadraw}
```

FIGURE 28 : Triangulation de Delaunay

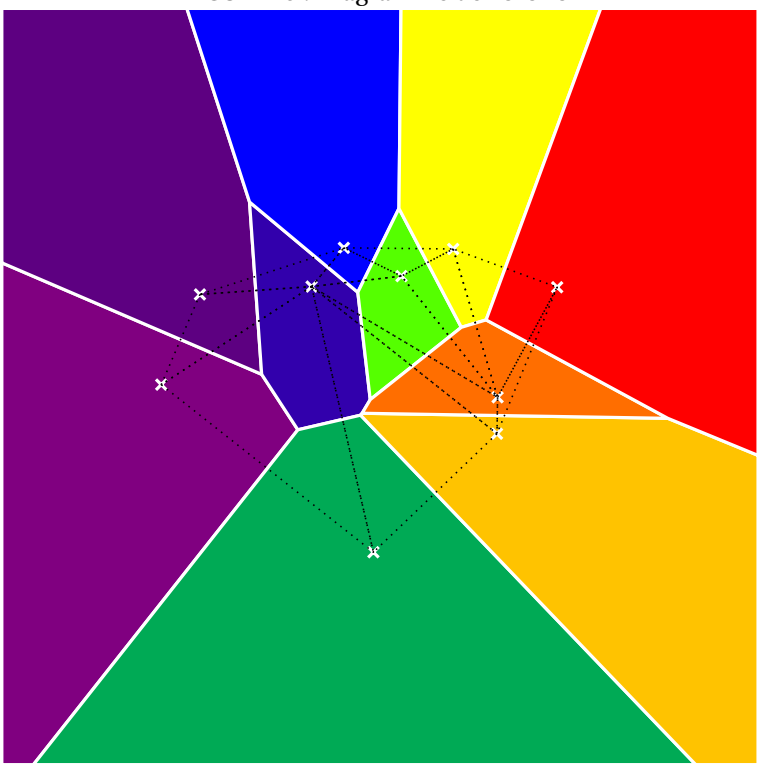


4) voronoi()

La fonction **ld.voronoi(L [, window])** où ⟨*L*⟩ est une liste de nombres complexes **distincts**, détermine le diagramme de Voronoï des points de la liste ⟨*L*⟩. Cette fonction renvoie une liste d'éléments de la forme `{A,polygone}` où `A` est un point la liste ⟨*L*⟩, et `polygone` une liste de nombres complexes représentant les sommets de la cellule associée à `A`. Il y a ainsi une cellule par point de ⟨*L*⟩. La cellule du point `A` contient les points du plan qui sont plus proches de `A` que des autres points de ⟨*L*⟩. Cette fonction utilise la triangulation de Delaunay. L'argument optionnel ⟨*window*⟩, qui vaut par défaut `{-5,5,-5,5}`, est utilisée pour clipper les cellules de Voronoï qui sont non bornées, cette fenêtre est automatiquement agrandie si nécessaire, pour contenir tous les points de ⟨*L*⟩ ainsi que tous les centes des cercles circonscrits au triangles de Delaunay (attention : cela ne change pas la fenêtre 2D du graphique en cours).

```
\begin{luadraw}{name=voronoi}
local ld = luadraw
local g = ld.graph:new{ bbox=true, margin={0,0,0,0}, size={10,10}}
local i = ld.cpx.I
local S = {0.285+1.46*i,1.556-0.142*i,2.344+1.313*i,-2.38+1.218*i,1.548-0.624*i,
0.969+1.819*i,-0.086-2.191*i,-0.477+1.834*i,-0.904+1.322*i,-2.892+0.025*i}
local V = ld.voronoi(S)
local colors = ld.getpalette(ld.rainbow,#V)
for k,T in ipairs(V) do
    local A, polygon = table.unpack(T)
    g:Dpolyline(polygon,true,"color=white, line width=1.2pt,fill="..colors[k])
    g:Ddots(A,"mark=x,white,scale=2,line width=1.2pt")-- A est un des points de S
end
g:Dpolyline(ld.delaunay(S),true,"dotted,line width=0.6pt") -- triangles de Delaunay
g:Show()
\end{luadraw}
```

FIGURE 29 : Diagramme de Voronoï



5) line2strip()

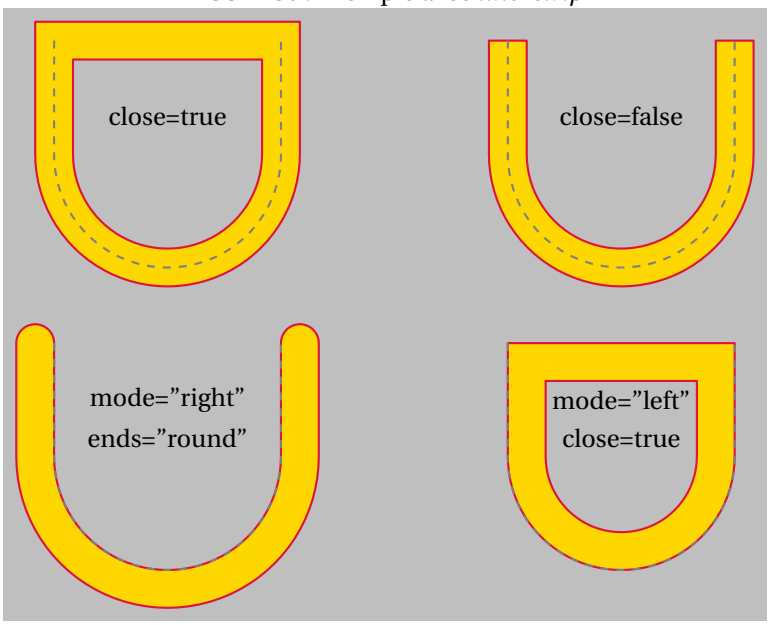
La fonction **ld.line2strip(L, width [, close, ends, mode])** où ⟨*L*⟩ est une liste de nombres complexes, ou une liste de listes de nombres complexes, renvoie un chemin représentant une "bande" calculée sur ⟨*L*⟩ et de largeur ⟨*width*⟩. L'argument optionnel ⟨*close*⟩ est un booléen qui indique si ⟨*L*⟩ doit être refermée (`false` par défaut).

L'argument optionnel ⟨*ends*⟩ indique la façon dont les deux extrémités de la bande sont dessinées, les valeurs possibles sont : `"none"`, `"butt"` (valeur par défaut) ou `"round"`. Pour rester compatible avec l'ancienne version, cet argument peut également prendre les valeurs `true` (équivalent à `"butt"`) ou `false` (équivalent à `"none"`).

L'argument ⟨*mode*⟩ peut valoir soit `"center"` (valeur par défaut), soit `"left"`, soit `"right"`, dans le premier cas la bande est centrée sur ⟨*L*⟩, dans le second cas elle est sur la gauche de ⟨*L*⟩, et dans le troisième cas elle est sur la droite de ⟨*L*⟩.

```
\begin{luadraw}{name=line2strip}
local ld = luadraw
local g = ld.graph:new{bbox=false, bg="lightgray"}
local i = ld.cpx.I; g:Linewidth(8)
local p = {-3+3*i,-3,"l",0,3,3,1,"ca", 3+3*i,"l"}
local P = ld.path(p) -- p est converti en ligne polygonale
g:Setmatrix({-3+3*i,0.5,0.5*i})
local L = ld.line2strip(P,1,true)
g:Dpath(L,"Crimson,fill=Gold"); g:Dpath(p,"gray,dashed")
g:Dlabel("close=true",i,{})
g:Setmatrix({3+3*i,0.5,0.5*i})
L = ld.line2strip(P,1,false)
g:Dpath(L,"Crimson,fill=Gold"); g:Dpath(p,"gray,dashed")
g:Dlabel("close=false",i,{})
g:Setmatrix({-3-i,0.5,0.5*i})
L = ld.line2strip(P,1,false,"round","right")
g:Dpath(L,"Crimson,fill=Gold"); g:Dpath(p,"gray,dashed")
g:Dlabel('mode="right"',1.5*i,{}); g:Dlabel('ends="round"',0.5*i,{})
g:Setmatrix({3-i,0.5,0.5*i})
L = ld.line2strip(P,1,true,false,"left")
g:Dpath(L,"Crimson,fill=Gold"); g:Dpath(p,"gray,dashed")
g:Dlabel('mode="left"',1.5*i,{}); g:Dlabel("close=true",0.5*i,{})
g:Show()
\end{luadraw}
```

FIGURE 30 : Exemple avec *line2strip*



6) parallel_polyline()

La fonction **ld.parallel_polyline(L, width [, close])** où ⟨*L*⟩ est une liste de nombres complexes, ou une liste de listes de nombres complexes, renvoie une ligne polygonale parallèle à ⟨*L*⟩ et située à une "distance" égale à ⟨*width*⟩. L'argument ⟨*width*⟩ peut être positif ou négatif pour être d'un coté ou de l'autre de ⟨*L*⟩ (cela dépend du sens de parcours de ⟨*L*⟩). L'argument optionnel ⟨*close*⟩ est un booléen qui indique si ⟨*L*⟩ doit être refermée (`false` par défaut).

7) sss_triangle()

La fonction **ld.sss_triangle(ab, bc, ca)** où ⟨*ab*⟩, ⟨*bc*⟩ et ⟨*ca*⟩ sont trois longueurs, calcule et renvoie une liste de trois points (3 complexes) {A, B, C} formant les sommets d'un triangle direct dont les longueurs des côtés sont les arguments, c'est-à-dire AB = ab, BC = bc et CA = ca, lorsque cela est possible. Le sommet A est toujours le complexe 0 et le sommet B est toujours le complexe ab. Ce triangle peut être dessiné avec la méthode **g:Dpolyline**.

8) sas_triangle()

La fonction **ld.sas_triangle(ab, alpha, ca)** où ⟨*ab*⟩ et ⟨*ca*⟩ sont deux longueurs, ⟨*alpha*⟩ un angle en degrés, calcule et renvoie une liste de trois points (3 complexes) {A, B, C} formant les sommets d'un triangle tel que AB = ab, CA = ca, et tel que l'angle $(\vec{AB}, \vec{AC})$ a pour mesure ⟨*alpha*⟩, lorsque cela est possible. Le sommet A est toujours le complexe 0 et le sommet B est toujours le complexe ab. Ce triangle peut être dessiné avec la méthode **g:Dpolyline**.

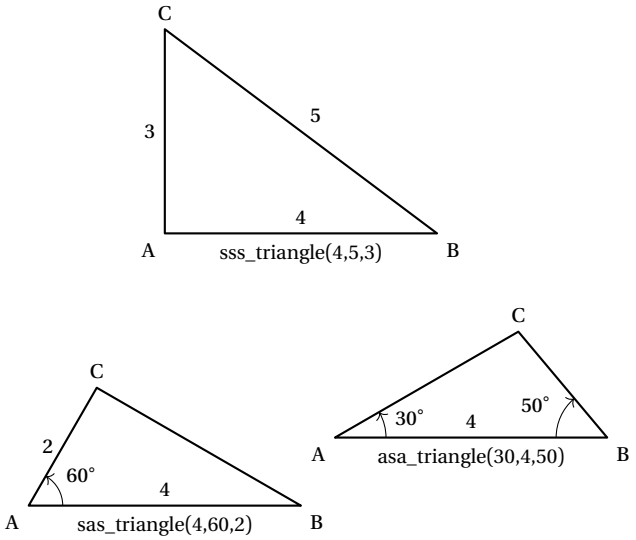
9) asa_triangle()

La fonction **ld.asa_triangle(alpha, ab, beta)** où ⟨*ab*⟩ est une longueur, ⟨*alpha*⟩ et ⟨*beta*⟩ deux angles en degrés, calcule et renvoie une liste de trois points (3 complexes) {A, B, C} formant les sommets d'un triangle tel que AB = ab, tel que l'angle $(\vec{AB}, \vec{AC})$ a pour mesure ⟨*alpha*⟩, et tel que l'angle $(\vec{BA}, \vec{BC})$ a pour mesure ⟨*beta*⟩, lorsque cela est possible. Le sommet A est toujours le complexe 0 et le sommet B est toujours le complexe ab. Ce triangle peut être dessiné avec la méthode **g:Dpolyline**.

```
\begin{luadraw}{name=sss_triangles_and_co}
local ld = luadraw
local g = ld.graph:new{window={-5,5,-3,5},size={10,10}}
g:Labelsize("footnotesize"); g:Linewidth(8)
local Zp, i, deg = ld.cpx.Zp, ld.cpx.I, ld.deg
local T1 = ld.shift( ld.sss_triangle(4,5,3), 2*i-2)
local T2 = ld.shift( ld.sas_triangle(4,60,2), -4-2*i)
local T3 = ld.shift( ld.asa_triangle(30,4,50), 0.5-i)
g:Dpolyline({T1,T2,T3}, true)
```

```
g:Linewidth(4)
g:Darc(T2[2],T2[1],T2[3],0.5,1,"->")
g:Darc(T3[2],T3[1],T3[3],0.75,1,"->")
g:Darc(T3[1],T3[2],T3[3],0.75,-1,"->")
g:Dlabel(
  "$4$",(T1[1]+T1[2])/2,{pos="N"}, "$5$",(T1[2]+T1[3])/2,{pos="NE"},"$3$",(T1[1]+T1[3])/2,{pos="W"},
  "$4$",(T2[1]+T2[2])/2,{pos="N"},"$60^\\circ$", T2[1]+Zp(0.9,30*deg),{pos="center"},
  "$2$",(T2[1]+T2[3])/2,{pos="W"},
  "$4$",(T3[1]+T3[2])/2,{pos="N"}, "$30^\\circ$",T3[1]+Zp(1.15,15*deg),{pos="center"},
  "$50^\\circ$",T3[2]+Zp(1.15,155*deg),{pos="center"},
  "sss\\_triangle(4,5,3)",(T1[1]+T1[2])/2,{pos="S"}, "sas\\_triangle(4,60,2)",(T2[1]+T2[2])/2,{},
  "asa\\_triangle(30,4,50)",(T3[1]+T3[2])/2,{})
for _,T in ipairs({T1,T2,T3}) do
    g:Dlabel("$A$",T[1],{pos="SW"}, "$B$",T[2],{pos="SE"},"$C$",T[3],{pos="N"})
end
g:Show()
\end{luadraw}
```

FIGURE 31 : sss_triangle, sas_triangle et asa_triangle



IV Calculs sur les listes

1) concat

La fonction **ld.concat(table1, table2, ...)** concatène toutes les tables passées en argument, et renvoie la table qui en résulte.

- Chaque argument peut être un réel, un complexe ou une table.
- Exemple : l'instruction `ld.concat(1,2,3,4,5,6,7)` renvoie la table `{1,2,3,4,5,6,7}`.

2) cut

La fonction **ld.cut(L, A [, before])** permet de couper ⟨*L*⟩ au point ⟨*A*⟩ qui est supposé être situé sur la ligne ⟨*L*⟩ (qui est soit une liste de complexes, soit une ligne polygonale c'est-à-dire une liste de listes de complexes). Si l'argument ⟨*before*⟩ vaut `false` (valeur par défaut), alors la fonction renvoie la partie située avant ⟨*A*⟩, suivie de la partie située après ⟨*A*⟩, sinon c'est l'inverse.

3) cutpolyline

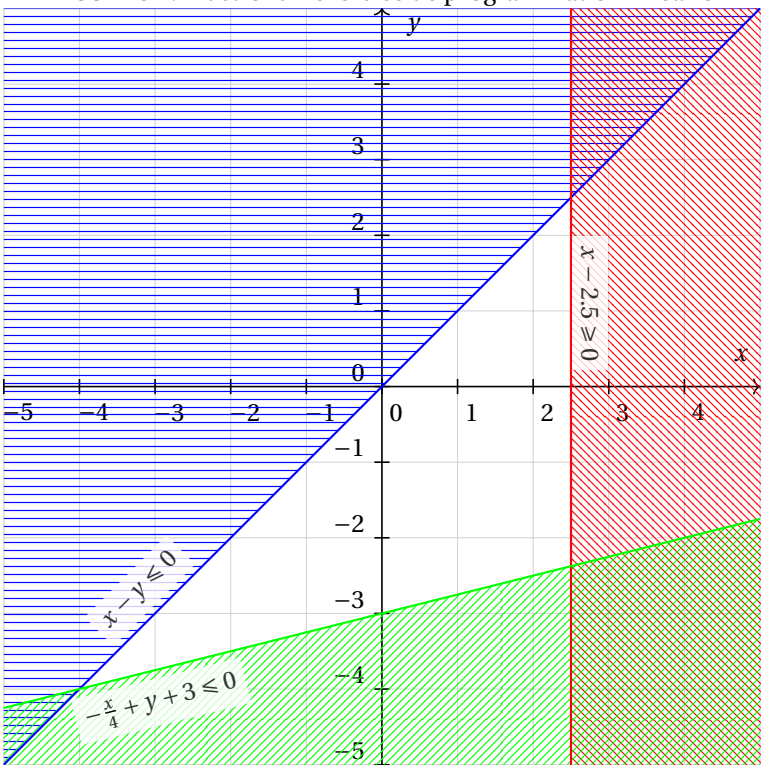
La fonction **ld.cutpolyline(L, D [, close])** permet de couper la ligne polygonale ⟨*L*⟩ avec la droite ⟨*D*⟩. L'argument ⟨*L*⟩ doit être une liste de complexes ou une liste de listes de complexes, l'argument ⟨*D*⟩ est une liste de la forme `{A,u}` où A est

un complexe (point de la droite) et u un complexe non nul (vecteur directeur de la droite). L'argument ⟨*close*⟩ indique si la ligne ⟨L⟩ doit être refermée (`false` par défaut). La fonction renvoie trois choses :

- La partie de ⟨L⟩ qui est dans le demi-plan défini par la droite à "gauche" de u (c'est à dire contenant le point A + iu) (c'est une ligne polygonale),
- suivi de la partie de ⟨L⟩ qui est dans l'autre demi-plan (ligne polygonale),
- suivi de la liste des points d'intersection entre ⟨L⟩ et la droite ⟨D⟩.

```
\begin{luadraw}{name=cutpolyline}
local ld = luadraw
local g = ld.graph:new{window={-5,5,-5,5}, size={10,10},margin={0,0,0,0}}
g:Linewidth(6)
local i = ld.cpx.I
local P = g:Box2d() -- polygone représentant la fenêtre 2D
local D1 = ld.lineEq(1,-1,0,'<')     -- x-y < 0
local D2 = ld.lineEq(1,0,-2.5,'>')   -- x-2 > 0
local D3 = ld.lineEq(-1/4,1,3,'<')   -- x/4+y+3 < 0
local P1 = ld.cutpolyline(P,D1,true)
local P2 = ld.cutpolyline(P,D2,true)
local P3 = ld.cutpolyline(P,D3,true)
g:Daxes({0,1,1},{grid=true,gridcolor="LightGray",arrows="->",legend={"$x$","$y$"}})
g:Filloptions("horizontal","blue"); g:Dpolyline(P1,true,"draw=none")
g:Filloptions("fdiag","red"); g:Dpolyline(P2,true,"draw=none")
g:Filloptions("bdiag","green"); g:Dpolyline(P3,true,"draw=none")
g:Filloptions("none","black",1); g:Linewidth(8)
g:Dline(D1,"blue"); g:Dline(D2,"red"); g:Dline(D3,"green")
g:Dlabel(
"$x-y\\leqslant 0$",-3-3*i,{pos="N",dir={1+i,-1+i},dist=0.1,node_options="fill=white,fill opacity=0.8"},
"$x-2.5\\geqslant0$", 2.5+i,{dir={-i,1}},
"$-\\frac{x}{4}+y+3\\leqslant0$", -3-15/4*i,{pos="S",dir={1+i/4,i-1/4}})
g:Show()
\end{luadraw}
```

FIGURE 32 : Illustrer un exercice de programmation linéaire



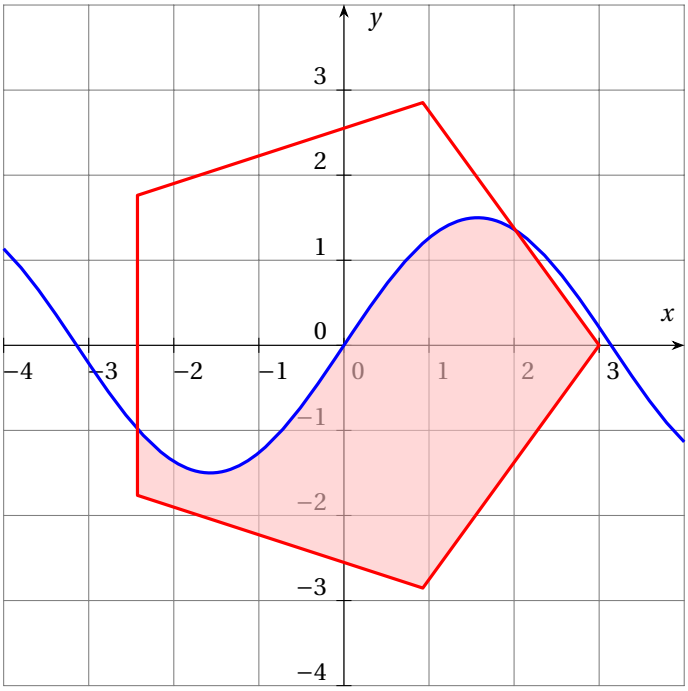
4) cutpolyline2

La fonction **ld.cutpolyline2(P, f, sg, [, close])** permet de couper le polygone ⟨P⟩ avec la courbe représentative de la fonction ⟨f⟩. L'argument ⟨P⟩ doit être une liste de nombres complexes, l'argument ⟨f⟩ est une fonction ⟨f⟩ : $x \mapsto f(x) \in \mathbf{R}$.

L'argument ⟨*sg*⟩ est soit `">"`, soit `"<"`, dans le premier cas, la fonction renvoie la partie du polygone ⟨*P*⟩ dont les points vérifient $y > f(x)$, dans le second cas ce sont les points vérifiant $y < f(x)$. L'argument facultatif ⟨*close*⟩ indique si ⟨*P*⟩ doit être refermé.

```
\begin{luadraw}{name=cutpolyline2}
local ld = luadraw
local g = ld.graph:new{window = {-4, 4, -4, 4}, size ={10,10}}
g:Daxes({0,1,1}, {grid=true, arrows="-Stealth", legend={"$x$","$y$"}})
local f = function(x) return 1.5*math.sin(x) end
local P = ld.polyreg(0,3,5) -- pentagone
local L = ld.cutpolyline2(P,f,'<',true) -- partie du polygone située sous la courbe de f
g:Dpolyline(L,true,"draw=none,fill=pink,fill opacity=0.6")
g:Linewidth(12)
g:Dcartesian(f, {draw_options="blue"})
g:Dpolyline(P,true,"red")
g:Show()
\end{luadraw}
```

FIGURE 33 : cutpolyline2



5) getbounds

- La fonction **ld.getbounds(L)** renvoie les bornes `xmin,xmax,ymin,ymax` de la ligne polygonale ⟨*L*⟩.
- Exemple : `local xmin,xmax,ymin,ymax=ld.getbounds(L)` (où ⟨*L*⟩ désigne une ligne polygonale).

6) getdot

La fonction **ld.getdot(x, L)** renvoie le point d'abscisse ⟨*x*⟩ (réel entre 0 et 1) le long de la composante connexe ⟨*L*⟩ (liste de complexes). L'abscisse 0 correspond au premier point et l'abscisse 1 au dernier, plus généralement, ⟨*x*⟩ correspond à un pourcentage de la longueur de ⟨*L*⟩.

7) insert

La fonction **ld.insert(table1, table2 [, pos])** insère les éléments de ⟨*table2*⟩ dans ⟨*table1*⟩ à la position ⟨*pos*⟩.

- L'argument ⟨*table2*⟩ peut être un réel, un complexe ou une table.
- L'argument ⟨*table1*⟩ doit être une variable qui désigne une table, celle-ci sera modifiée par la fonction.
- Si l'argument ⟨*pos*⟩ vaut `nil`, l'insertion se fait à la fin de ⟨*table1*⟩.

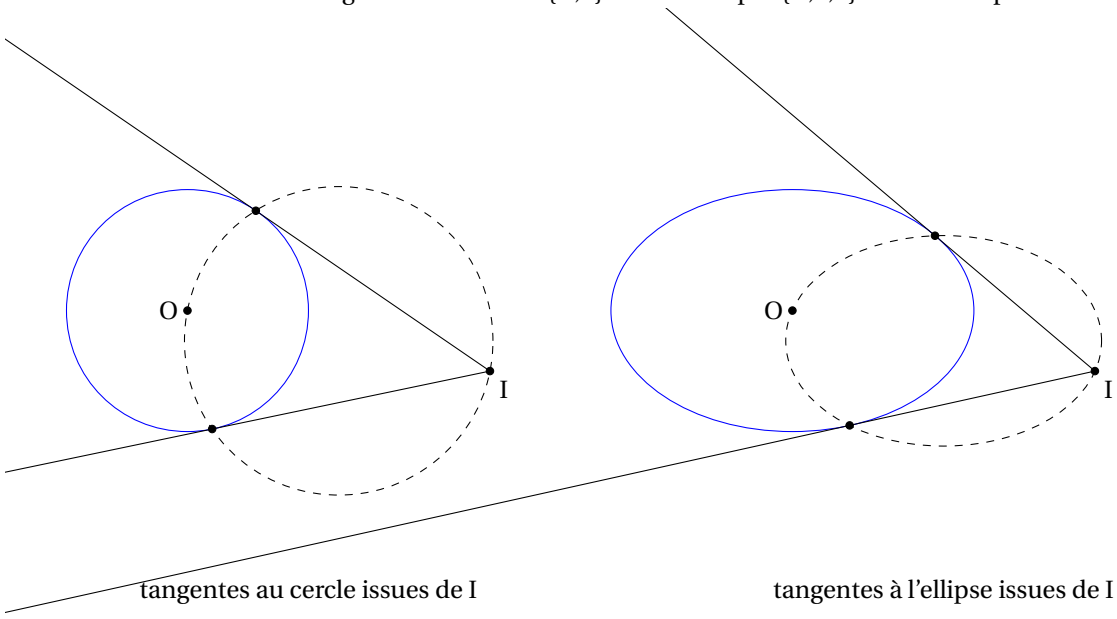
- Exemple : si une variable L vaut `{1,2,6}`, alors après l'instruction `ld.insert(L,3,4,5,3)`, la variable L sera égale à `{1,2,3,4,5,6}`.

8) interCC

La fonction **ld.interCC(C1, C2)** renvoie l'intersection cercle ⟨*C1*⟩ avec le cercle ⟨*C2*⟩, où `C1={O1,r1}` (cercle de centre O_1 et de rayon r_1), et `C2={O2,r2}` (cercle de centre O_2 et de rayon r_2). La fonction renvoie une liste contenant 1 ou 2 points, ou bien le cercle en entier si l'intersection n'est pas vide, sinon elle renvoie `nil`.

```
\begin{luadraw}{name=interCC}
local ld = luadraw
local cpx = ld.cpx
local g = ld.graph:new{window={-10,10,-5,5}, margin={0,0,0,0},size={16,8}}
local i = cpx.I
-- pour le cercle {O,2}
g:Saveattr(); g:Viewport(-10,0,-5,5); g:Coordsystem(-4,6,-5,5)
local O = -1
local C1, I = {O, 2}, 4-i
local C2 = {(O+I)/2,cpx.abs(I-O)/2}
local rep = ld.interCC(C1,C2) -- points de tangence
g:Dcircle(C1,"blue"); g:Dcircle(C2,"dashed")
g:Dhline(I,rep[1],"red"); g:Dhline(I,rep[2],"red") --demi-tangentes
g:Ddots(rep); g:Ddots({O,I}); g:Dlabel("$I$",I,{pos="SE"},"$O$",O,{pos="W"},
  "tangentes au cercle issues de $I$",1-5*i,{pos="N"})
g:Restoreattr()
-- pour l'ellipse (E) : {O,3,2}
g:Saveattr(); g:Viewport(0,10,-5,5); g:Coordsystem(-4,6,-5,5)
local mat = {0,1.5,i} -- cette matrice transforme un cercle {O1,2} en l'ellipse (E)
local inv_mat = ld.invmatrix(mat) -- matrice inverse
local O1, I1 = table.unpack( ld.mtransform({O,I},inv_mat) ) -- antécédents de O et de I
C1 = {O1, 2}
C2 = {(O1+I1)/2,cpx.abs(I1-O1)/2}
rep = ld.interCC(C1,C2) -- points de tangence (tangentes issues de I1)
g:Composematrix(mat) -- on applique la matrice pour retrouver l'ellipse, la tangence est conservée
g:Dcircle(C1,"blue"); g:Dcircle(C2,"dashed")
g:Dhline(I1,rep[1],'red'); g:Dhline(I1,rep[2],"red")
g:Ddots(rep); g:Ddots({O1,I1}); g:Dlabel("$I$",I1,{pos="SE"},"$O$",O1,{pos="W"},
  "tangentes à l'ellipse issues de $I$",1-5*i,{pos="N"})
g:Restoreattr()
g:Show()
\end{luadraw}
```

FIGURE 34 : Tangentes à un cercle {O,2} et à une ellipse {O,3,2} issues d'un point



9) interD

La fonction **ld.interD(d1, d2)** renvoie le point d'intersection des droites ⟨*d1*⟩ et ⟨*d2*⟩, une droite est une liste de deux complexes : un point de la droite et un vecteur directeur.

10) interDC

La fonction **ld.interDC(d, C)** renvoie l'intersection de la droite ⟨*d*⟩ avec le cercle ⟨*C*⟩, où `d={A,u}` (droite passant par A et dirigée par u), et `C={O,r}` (cercle de centre O et de rayon r). La fonction renvoie une liste contenant 1 ou 2 points si l'intersection n'est pas vide, elle renvoie `nil` sinon.

11) interDL

La fonction **ld.interDL(d, L)** renvoie la liste des points d'intersection entre la droite ⟨*d*⟩ et la ligne polygonale ⟨*L*⟩.

12) interL

La fonction **ld.interL(L1, L2)** renvoie la liste des points d'intersection des lignes polygonales définies par *L1* et *L2*, ces deux arguments sont deux listes de complexes ou deux listes de listes de complexes).

13) interP

La fonction **ld.interP(P1, P2)** renvoie la liste des points d'intersection des chemins définis par ⟨*P1*⟩ et ⟨*P2*⟩, ces deux arguments sont deux listes de complexes et d'instructions (voir *Dpath* page 40).

14) linspace

La fonction **ld.linspace(a, b [, nbdots])** renvoie une liste de ⟨*nbdots*⟩ nombres équirépartis de ⟨*a*⟩ jusqu'à ⟨*b*⟩. Par défaut ⟨*nbdots*⟩ vaut 50.

Autre syntaxe possible : **ld.linspace(a1, b1, n1, b2, n2, ..., bp, np)** renvoie une liste de ⟨*n1*⟩ nombres équirépartis de ⟨*a1*⟩ jusqu'à ⟨*b1*⟩, suivis de ⟨*n2*⟩ nombres équirépartis de ⟨*b1*⟩ jusqu'à ⟨*b2*⟩ (mais ⟨*b1*⟩ n'est pas répété), etc.

15) map

La fonction **ld.map(f, list)** applique la fonction ⟨*f*⟩ à chaque élément de la ⟨*list*⟩ et renvoie la table des résultats. Lorsqu'un résultat vaut `nil`, c'est le complexe `cpx.Jump` qui est inséré dans la liste.

16) merge

La fonction **ld.merge(L [, epsilon])** recolle si c'est possible, les composantes connexes de ⟨*L*⟩ qui doit être une liste de listes de complexes, la fonction renvoie une nouvelle ligne polygonale. Les comparaisons se font à ⟨*epsilon*⟩ près, par défaut on a ⟨*epsilon*⟩$= 10^{-10}$.

17) range

La fonction **ld.range(a, b [, step])** renvoie la liste des nombres de ⟨*a*⟩ jusqu'à ⟨*b*⟩ avec un pas égal à ⟨*step*⟩, celui-ci vaut 1 par défaut.

18) read_csv_file

La fonction **ld.read_csv_file(file, options)** permet la lecture d'un fichier *csv*. L'argument ⟨*file*⟩ est une chaîne de caractères représentant le fichier avec extension : `"<name>.csv"`. L'argument ⟨*options*⟩ est une table dont les champs représentent les options, celles-ci sont (avec leur valeur par défaut) :

- `header=true`, avec la valeur `true` cela signifie que la première ligne du fichier contient les noms des colonnes.

- `dic=false`, avec la valeur `true` cela signifie que le résultat renvoyé sera une liste de dictionnaires (un par ligne), les clés de ces dictionnaires étant les noms des colonnes. Avec la valeur `false` (valeur par défaut) le résultat est une liste de listes de valeurs (une liste de valeurs par ligne). Lorsque l'option `header` a la valeur `false`, l'option `dic` reste automatiquement à `false`.
- `sep=","`, chaîne de caractères indiquant le séparateur des valeurs sur chaque ligne.
- `num=true`, booléen indiquant si les valeurs doivent automatiquement être converties en valeurs numériques (lorsque cette conversion échoue, la valeur reste une chaîne de caractères).
- `comment="%%"`, chaîne de caractères annonçant un commentaire (une ligne commençant par ces caractères est ignorée). **Attention** : la recherche utilise la méthode *match* de Lua, méthode pour laquelle le caractère % est un caractère spécial, il doit être doublé ("%%") pour être considéré comme un caractère.

Le résultat renvoyé par cette fonction est une séquence constituée dans cet ordre par :

1. Une liste de listes de résultats (une liste de résultats par ligne).
2. La liste des valeurs de la première ligne lorsque l'option `header` a la valeur `true`.

19) Fonctions de clipping

- La fonction **ld.clipseg(A, B, xmin, xmax, ymin, ymax)** clippe le segment [A;B] (nombres complexes) avec la fenêtre $[x_{\min};x_{\max}]\times[y_{\min};y_{\max}]$ et renvoie le résultat.
- La fonction **ld.clipline(d, xmin, xmax, ymin, ymax)** clippe la droite $\langle d\rangle$ avec la fenêtre $[x_{\min};x_{\max}]\times[y_{\min};y_{\max}]$ et renvoie le résultat. La droite $\langle d\rangle$ est une liste de deux complexes : un point et un vecteur directeur.
- La fonction **ld.clippolyline(L, xmin, xmax, ymin, ymax [, close])** clippe la ligne polygonale $\langle L\rangle$ avec la fenêtre $[x_{\min};x_{\max}]\times[y_{\min};y_{\max}]$ et renvoie le résultat. L'argument $\langle L\rangle$ est une liste de complexes ou une liste de listes de complexes. L'argument facultatif $\langle close\rangle$ (`false` par défaut) indique si la ligne polygonale doit être refermée.
- La fonction **ld.clipdots(L, xmin, xmax, ymin, ymax)** clippe la liste de points (nombres complexes) $\langle L\rangle$ avec la fenêtre $[x_{\min};x_{\max}]\times[y_{\min};y_{\max}]$ et renvoie le résultat (les points extérieurs sont simplement exclus). L'argument $\langle L\rangle$ est une liste de complexes ou une liste de listes de complexes.

20) Ajout de fonctions mathématiques

Outre les fonctions associées aux méthodes graphiques qui font des calculs et renvoient une ligne polygonale (comme **ld.cartesian**, **ld.periodic**, **ld.implicit**, **ld.odesolve**, etc), le paquet *luadraw* ajoute quelques fonctions mathématiques qui ne sont pas proposées nativement dans le module *math*.

Évaluation protégée : evalf

La fonction **ld.evalf(f, ...)** permet d'évaluer `f(...)` et de renvoyer le résultat s'il n'y a pas d'erreur d'exécution par Lua, dans le cas contraire, la fonction renvoie `nil`. Exemple, l'exécution de :

```
local Z = cpx.Z
local f = function(a,b)
    return 2*Z(a,1/b)
end
print(f(1,0))
```

provoque l'erreur d'exécution `attempt to perform arithmetic on a nil value` (dans la console), car ici `Z(1,1/0)` renvoie `nil`, et Lua n'accepte pas un argument égal à `nil` dans un calcul. Par contre, l'exécution de :

```
local Z = cpx.Z
local f = function(a,b)
    return 2*Z(a,1/b)
end
print(ld.evalf(f,1,0))
```

ne provoque pas d'erreur de la part de Lua, et il n'y a pas d'affichage non plus dans la console puisque la valeur à afficher est `nil`.

int

La fonction **ld.int(f, a, b)** renvoie une valeur approchée de l'intégrale de la fonction $\langle f \rangle$ sur l'intervalle $[a;b]$ (deux nombres complexes). La fonction $\langle f \rangle$ est à variable réelle et à valeurs réelles ou complexes. La méthode utilisée est la méthode de Simpson accélérée deux fois avec la méthode de Romberg.

Exemple :

```
$\int_0^1 e^{t^2}\mathrm d t \approx \directlua{tex.sprint(int(function(t) return math.exp(t^2) end, 0, 1))}$
```

Résultat : $\int_0^1 e^{t^2}\,\mathrm{d}t \approx 1.4626517459589$.

gcd

La fonction **ld.gcd(a, b)** renvoie le plus grand diviseur commun entre $\langle a \rangle$ et $\langle b \rangle$ deux entiers.

lcm

La fonction **ld.lcm(a, b)** renvoie le plus petit diviseur commun strictement positif entre $\langle a \rangle$ et $\langle b \rangle$ deux entiers.

nth_root

La fonction **ld.nth_root(n, x)** renvoie la racine énième du réel$\langle x \rangle$, l'argument $\langle n \rangle$ doit être un entier.

solve

La fonction **ld.solve(f, a, b [, n, df])** fait une résolution numérique de l'équation $f(x) = 0$ dans l'intervalle $[a;b]$, l'argument $\langle f \rangle$ est la fonction, $\langle a \rangle$, $\langle b \rangle$ sont deux nombres réels. L'intervalle est subdivisé en n morceaux (25 par défaut). L'argument facultatif $\langle df \rangle$ peut être :

- Soit la chaîne `"bisection"`, dans ce cas la méthode utilisée est une recherche dichotomique.
- Soit une fonction qui représente la dérivée de $\langle f \rangle$, dans ce cas la méthode utilisée est la méthode de Newton.
- Soit `nil` (valeur par défaut), dans ce cas la méthode utilisée est une variante de la méthode de Newton.

La fonction renvoie une liste de nombres ou bien `nil`.

Exemple 1 :

```
\begin{luacode}
local ld = luadraw
resol = function(f,a,b)
    local y = ld.solve(f,a,b)
    if y == nil then tex.sprint("\\emptyset")
    else
        local str = y[1]
        for k = 2, #y do
            str = str..", ".. y[k]
        end
        tex.sprint(str)
    end
end
\end{luacode}
\def\solve#1#2#3{\directlua{resol(#1,#2,#3)}}%
\begin{luacode}
f1 = function(x) return math.cos(x)-x end
f2 = function(x) return x^3-2*x^2+1/2 end
\end{luacode}
La résolution de l'équation $\cos(x)=x$ dans $[0;\frac{\pi}2]$ donne $\solve{f1}{0}{math.pi/2}$.\par
La résolution de l'équation $\cos(x)=x$ dans $[\frac{\pi}2;\pi]$ donne $\solve{f1}{math.pi/2}{math.pi}$.\par
La résolution de l'équation $x^3-2x^2+\frac12=0$ dans $[-1;2]$ donne : $\{\solve{f2}{-1}{2}\}$.
```

Résultat :

La résolution de l'équation $\cos(x) = x$ dans $[0;\frac{\pi}{2}]$ donne 0.73908513321516.
La résolution de l'équation $\cos(x) = x$ dans $[\frac{\pi}{2};\pi]$ donne $\emptyset$.
La résolution de l'équation $x^3-2x^2+\frac{1}{2} = 0$ dans $[-1;2]$ donne : $\{-0.45160596295578, 0.59696828323732, 1.8546376797185\}$.

Exemple 2 : on souhaite tracer la courbe de la fonction f définie par la condition :

$$\forall x \in \mathbf{R},\ \int_x^{f(x)} \exp(t^2)\mathrm{d}t = 1.$$

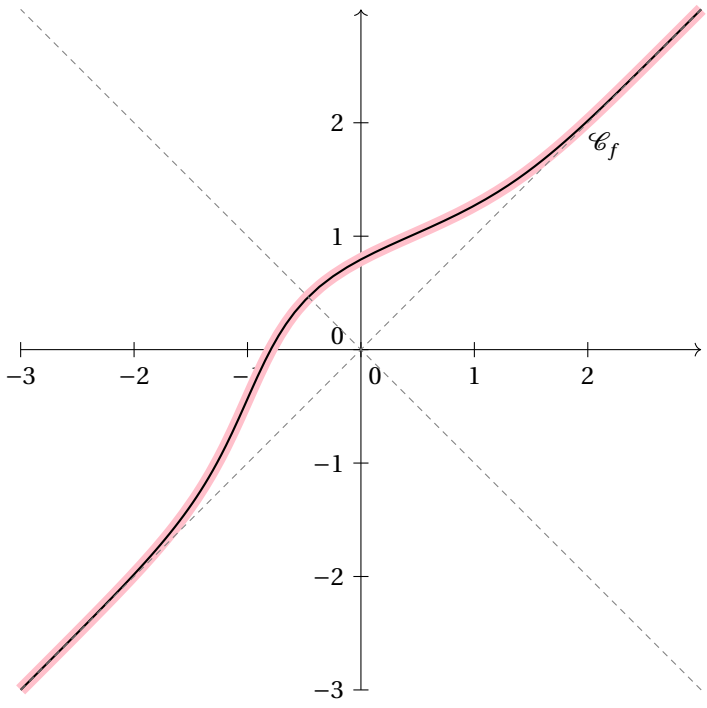
On a deux méthodes possibles :

1. On considère la fonction G: $(x,y) \mapsto \int_x^y \exp(t^2)\mathrm{d}t - 1$, et on dessine la courbe implicite d'équation $G(x,y) = 0$.
2. On détermine un réel y_0 tel que $\int_0^{y_0} \exp(t^2)\mathrm{d}t = 1$ et on dessine la solution de l'équation différentielle $y' = e^{x^2-y^2}$ vérifiant la condition initiale $y(0) = y_0$.

Dessinons les deux :

```
\begin{luadraw}{name=int_solve}
local ld = luadraw
local Z = ld.cpx.Z
local g = ld.graph:new{window={-3,3,-3,3},size={10,10}}
local h = function(t) return math.exp(t^2) end
local G = function(x,y) return ld.int(h,x,y)-1 end
local H = function(y) return G(0,y) end
local F = function(x,y) return math.exp(x^2-y^2) end
local y0 = ld.solve(H,0,1)[1] -- solution de H(x)=0
g:Daxes({0,1,1}, {arrows="->"})
g:Dimplicit(G, {draw_options="line width=4.8pt,Pink"})
g:Dodesolve(F,0,y0,{draw_options="line width=0.8pt"})
g:Lineoptions("dashed","gray",4); g:DlineEq(1,-1,0); g:DlineEq(1,1,0) -- bissectrices
g:Dlabel("${\\mathcal C}_f$",Z(2.15,2),{pos="S"})
g:Show()
\end{luadraw}
```

FIGURE 35 : Fonction f définie par $\int_x^{f(x)} \exp(t^2)\mathrm{d}t = 1$.



On voit que les deux courbes se superposent bien, cependant la première méthode (courbe implicite) est beaucoup plus gourmande en calculs, la méthode 2 est donc préférable.

V Transformations

Dans ce qui suit :

- l'argument ⟨*L*⟩ est soit un complexe, soit une liste de complexes soit une liste de listes de complexes,
- la droite ⟨*d*⟩ est une liste de deux complexes : un point de la droite et un vecteur directeur.

1) affin

La fonction **ld.affin(L, d, v, k)** renvoie l'image de ⟨*L*⟩ par l'affinité de base la droite ⟨*d*⟩, parallèlement au vecteur ⟨*v*⟩ et de rapport ⟨*k*⟩.

2) ftransform

La fonction **ld.ftransform(L, f)** renvoie l'image de ⟨*L*⟩ par la fonction ⟨*f*⟩ qui doit être une fonction de la variable complexe. Si un des éléments de ⟨*L*⟩ est le complexe `cpx.Jump` alors celui-ci est renvoyé tel quel dans le résultat.

3) hom

La fonction **ld.hom(L, factor [, center])** renvoie l'image de ⟨*L*⟩ par l'homothétie de centre ⟨*center*⟩ et de rapport ⟨*factor*⟩. Par défaut, l'argument ⟨*center*⟩ vaut 0.

4) inv

La fonction **ld.inv(L, radius [, center])** renvoie l'image de ⟨*L*⟩ par l'inversion par rapport au cercle de centre ⟨*center*⟩ et de rayon ⟨*radius*⟩. Par défaut, l'argument ⟨*center*⟩ vaut 0.

5) proj

La fonction **ld.proj(L, d)** renvoie l'image de ⟨*L*⟩ par la projection orthogonale sur la droite ⟨*d*⟩.

6) projO

La fonction **ld.projO(L, d, v)** renvoie l'image de ⟨*L*⟩ par la projection sur la droite ⟨*d*⟩ parallèlement au vecteur ⟨*v*⟩.

7) rotate

La fonction **ld.rotate(L, angle [, center])** renvoie l'image de ⟨*L*⟩ par la rotation de centre ⟨*center*⟩ et d'angle ⟨*angle*⟩ (en degrés). Par défaut, l'argument ⟨*center*⟩ vaut 0.

8) shift

La fonction **ld.shift(L, u)** renvoie l'image de ⟨*L*⟩ par la translation de vecteur ⟨*u*⟩.

9) simil

La fonction **ld.simil(L, factor, angle [, center])** renvoie l'image de ⟨*L*⟩ par la similitude de centre ⟨*center*⟩, de rapport *factor* et d'angle *angle* (en degrés). Par défaut, l'argument *center* vaut arguargu0.

10) sym

La fonction **ld.sym(L, d)** renvoie l'image de ⟨*L*⟩ par la symétrie orthogonale d'axe la droite ⟨*d*⟩.

11) symG

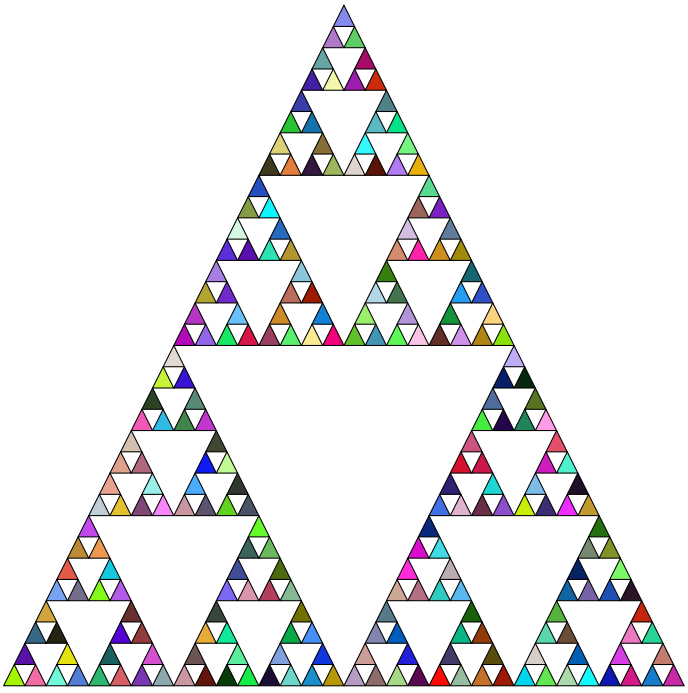
La fonction **ld.symG(L, d, v)** renvoie l'image de ⟨*L*⟩ par la symétrie par rapport à la droite ⟨*d*⟩ suivie de la translation de vecteur ⟨*v*⟩ (symétrie glissée).

12) symO

La fonction **ld.symO(L, d, v)** renvoie l'image de $\langle L \rangle$ par la symétrie par rapport à la droite $\langle d \rangle$ et parallèlement au vecteur $\langle v \rangle$ (symétrie oblique).

```
\begin{luadraw}{name=Sierpinski}
local ld = luadraw
local g = ld.graph:new{window={-5,5,-5,5},size={10,10}}
local i, hom = ld.cpx.I, ld.hom
local rand = math.random
local A, B, C = 5*i, -5-5*i, 5-5*i -- triangle initial
local T, niv = {{A,B,C}}, 5
for k = 1, niv do
    T = ld.concat( hom(T,0.5,A), hom(T,0.5,B), hom(T,0.5,C) )
end
for _,cp in ipairs(T) do
    g:Filloptions("full", ld.rgb(rand(),rand(),rand()))
    g:Dpolyline(cp,true)
end
g:Show()
\end{luadraw}
```

FIGURE 36 : Utilisation de transformations



VI Calcul matriciel

Si f est une application affine du plan complexe, on appellera matrice de f la liste (table) :

`{f(0),Lf(1),Lf(i)}`

où Lf désigne la partie linéaire de f (on a L$f(1) = f(1) - f(0)$ et L$f(i) = f(i) - f(0)$). La matrice identité est la variable globale notée `ID` dans le paquet *luadraw*, elle correspond simplement à la liste $\{0, 1, i\}$.

1) Calculs sur les matrices

applymatrix et applyLmatrix

- La fonction **ld.applymatrix(z, M)** applique la matrice $\langle M \rangle$ au complexe $\langle z \rangle$ et renvoie le résultat (ce qui revient à calculer $f(z)$ si $\langle M \rangle$ est la matrice de f). Lorsque $\langle z \rangle$ est le complexe `cpx.Jump` alors le résultat est `cpx.Jump`. Lorsque $\langle z \rangle$ est une chaîne de caractères alors la fonction renvoie $\langle z \rangle$.

- La fonction **ld.applyLmatrix(z, M)** applique la partie linéaire la matrice $\langle M \rangle$ au complexe $\langle z \rangle$ et renvoie le résultat (ce qui revient à calculer $Lf(z)$ si $\langle M \rangle$ est la matrice de f). Lorsque $\langle z \rangle$ est le complexe `cpx.Jump` alors le résultat est `cpx.Jump`.

composematrix

La fonction **ld.composematrix(M1, M2)** effectue le produit matriciel $\langle M1 \rangle \times \langle M2 \rangle$ et renvoie le résultat.

getmatrix

La fonction **ld.getmatrix(T1, T2)** calcule et renvoie la matrice qui transforme le triangle $\langle T1 \rangle$ en le triangle $\langle T2 \rangle$, avec $\langle T1 \rangle = \{a_1, b_1, c_1\}$ (liste de trois points 2D non alignés), et $\langle T2 \rangle = \{a_2, b_2, c_2\}$ (liste de trois points 2D).

invmatrix

La fonction **ld.invmatrix(M)** calcule et renvoie l'inverse de la matrice $\langle M \rangle$ lorsque cela est possible.

matrixof

- La fonction **ld.matrixof(f)** calcule et renvoie la matrice de $\langle f \rangle$ (qui doit être une application affine du plan complexe).
- Exemple : `ld.matrixof(function(z) return ld.proj(z,0,Z(1,-1)) end)` renvoie `{0,Z(0.5,-0.5),Z(-0.5,0.5)}` (matrice de la projection orthogonale sur la deuxième bissectrice).

mtransform et mLtransform

- La fonction **ld.mtransform(L, M)** applique la matrice $\langle M \rangle$ à la liste $\langle L \rangle$ et renvoie le résultat. $\langle L \rangle$ doit être une liste de complexes ou une liste de listes de complexes, si l'un d'eux est le complexe `cpx.Jump` ou une chaîne de caractères alors il est renvoyé tel quel.
- La fonction **ld.mLtransform(L, M)** applique la partie linéaire la matrice $\langle M \rangle$ à la liste $\langle L \rangle$ et renvoie le résultat. $\langle L \rangle$ doit être une liste de complexes ou une liste de listes de complexes, si l'un d'eux est le complexe `cpx.Jump` alors il est renvoyé tel quel.

2) Matrice associée au graphe

Lorsque l'on crée un graphe dans l'environnement *luadraw*, par exemple :

```
local ld = luadraw
local g = ld.graph:new{window={-5,5,-5,5},size={10,10}}
```

l'objet g créé possède une matrice de transformation qui est initialement l'identité. Toutes les méthodes graphiques utilisées appliquent automatiquement la matrice de transformation du graphe. Cette matrice est désignée par `g.matrix`, mais pour manipuler celle-ci, on dispose des méthodes qui suivent.

g:Composematrix()

La méthode **g:Composematrix(M)** multiplie la matrice du graphe g par la matrice $\langle M \rangle$ (avec $\langle M \rangle$ à droite) et le résultat est affecté à la matrice du graphe. L'argument $\langle M \rangle$ doit donc être une matrice.

g:Det2d()

La méthode **g:Det2d()** envoie 1 lorsque la matrice de transformation a un déterminant positif, et -1 dans le cas contraire. Cette information est utile lorsqu'on a besoin de savoir si l'orientation du plan a été changée ou non.

g:IDmatrix()

La méthode **g:IDmatrix()** réaffecte l'identité à la matrice du graphe g.

g:Mtransform()

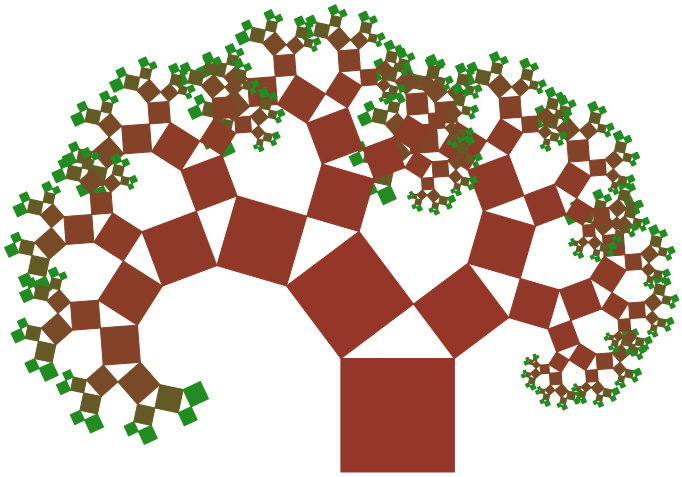
La méthode **g:Mtransform(L)** applique la matrice du graphe g à $\langle L \rangle$ et renvoie le résultat, l'argument $\langle L \rangle$ doit être une liste de complexes, ou une liste de listes de complexes.

g:MLtransform()

La méthode **g:MLtransform(L)** applique la partie linéaire de la matrice du graphe g à $\langle L \rangle$ et renvoie le résultat, l'argument $\langle L \rangle$ doit être une liste de complexes, ou une liste de listes de complexes.

```
\begin{luadraw}{name=Pythagore}
local ld = luadraw
local g = ld.graph:new{window={-15,15,0,22},size={10,10}}
local a, b, c = 3, 4, 5 -- un triplet de Pythagore
local i, arccos, exp = ld.cpx.I, math.acos, ld.cpx.exp
local f1 = function(z)
    return (z-c)*a/c*exp(-i*arccos(a/c))+c+i*c end
local M1 = ld.matrixof(f1)
local f2 = function(z)
    return z*b/c*exp(i*arccos(b/c))+i*c end
local M2 = ld.matrixof(f2)
local arbre
arbre = function(n)
    local color = ld.mixcolor(ld.ForestGreen,1,ld.Brown,n)
    g:Linecolor(color); g:Dsquare(0,c,1,"fill=".color)
    if n > 0 then
        g:Savematrix(); g:Composematrix(M1); arbre(n-1)
        g:Restorematrix(); g:Savematrix(); g:Composematrix(M2)
        arbre(n-1); g:Restorematrix()
    end
end
arbre(8)
g:Show()
\end{luadraw}
```

FIGURE 37 : Utilisation de la matrice du graphe



g:Rotate()

La méthode **g:Rotate(angle [, center])** modifie la matrice de transformation du graphe g en la composant avec la matrice de la rotation d'angle $\langle angle \rangle$ (en degrés) et de centre $\langle center \rangle$. L'argument $\langle center \rangle$ est un complexe qui vaut 0 par défaut.

g:Scale()

La méthode **g:Scale(factor [, center])** modifie la matrice de transformation du graphe *g* en la composant avec la matrice de l'homothétie de rapport ⟨*factor*⟩ et de centre ⟨*center*⟩. L'argument ⟨*center*⟩ est un complexe qui vaut 0 par défaut.

g:Savematrix() et g:Restorematrix()

- La méthode **g:Savematrix()** permet de sauvegarder dans une pile la matrice de transformation du graphe *g*.
- La méthode **g:Restorematrix()** permet de restaurer la matrice de transformation du graphe *g* à sa dernière valeur sauvegardée.

g:Setmatrix()

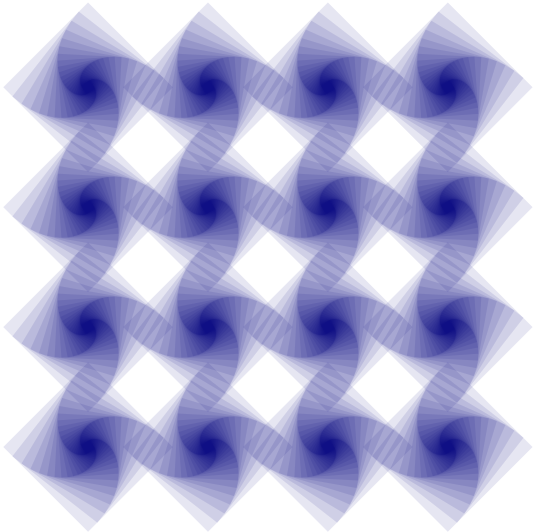
La méthode **g:Setmatrix(M)** permet d'affecter la matrice ⟨*M*⟩ à la matrice de transformation du graphe *g*.

g:Shift()

La méthode **g:Shift(v)** modifie la matrice de transformation du graphe *g* en la composant avec la matrice de la translation de vecteur ⟨*v*⟩ qui doit être un complexe.

```
\begin{luadraw}{name=free_art}
local ld = luadraw
local du = math.sqrt(2)/2
local g = ld.graph:new{window={1-du,4+du,1-du,4+du},
    margin={0,0,0,0},size={7,7}}
local i = ld.cpx.I
g:Linestyle("noline")
g:Filloptions("full","Navy",0.1)
for X = 1, 4 do
    for Y = 1, 4 do
        g:Savematrix()
        g:Shift(X+i*Y); g:Rotate(45)
        for k = 1, 25 do
            g:Dsquare((1-i)/2,(1+i)/2,1)
            g:Rotate(7); g:Scale(0.9)
        end
        g:Restorematrix()
    end
end
g:Show()
\end{luadraw}
```

FIGURE 38 : Utilisation de Shift, Rotate et Scale



3) Changement de vue. Changement de repère

Changement de vue : lors de la création d'un nouveau graphique, par exemple :

```
local ld = luadraw
local g = ld.graph:new{window={-5,5,-5,5},size={10,10}}
```

L'option `window={xmin,xmax,ymin,ymax}` fixe la vue pour le graphique g, ce sera le pavé $[x_{\min};x_{\max}]\times[y_{\min};y_{\max}]$ de $\mathbf{R}^2$, et tous les tracés vont être clippés par cette fenêtre (sauf les labels qui peuvent débordés dans les marges, mais pas au-delà). Il est possible, à l'intérieur de ce pavé, de définir un autre pavé pour faire une nouvelle vue, avec la méthode **g:Viewport(x1, x2, y1, y2)**. Les valeurs de $\langle x1\rangle$, $\langle x2\rangle$, $\langle y1\rangle$, $\langle y2\rangle$ se réfèrent la fenêtre initiale définie par l'option `window`. À partir de là, tout ce qui sort de cette nouvelle zone va être clippé, et la matrice du graphe est réinitialisée à l'identité, par conséquent il faut sauvegarder auparavant les paramètres graphiques courants :

```
g:Saveattr()
g:Viewport(x1,x2,y1,y2)
```

Pour revenir à la vue précédente avec la matrice précédente, il suffit d'effectuer une restauration des paramètres graphiques avec la méthode **g:Restoreattr()**.

Attention : à chaque instruction **g:Saveattr()** doit correspondre une instruction **g:Restoreattr()**, sinon il y aura une erreur à la compilation.

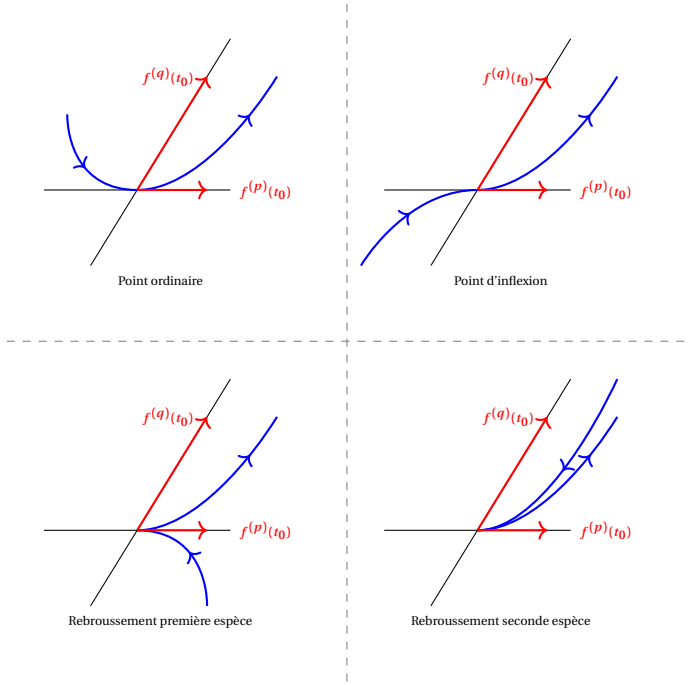
Changement de repère : on peut changer le système de coordonnées de la vue courante avec la méthode

g:Coordsystem(x1, x2, y1, y2 [, ortho]).

Cette méthode va modifier la matrice du graphe de sorte que tout se passe comme si la vue courante correspondait au pavé $[x_1;x_2]\times[y_1;y_2]$, l'argument booléen facultatif $\langle ortho\rangle$ indique si le nouveau repère doit être orthonormé ou non (`false` par défaut). Comme la matrice du graphe est modifiée il est préférable de sauvegarder les paramètres graphiques avant, et de les restaurer ensuite. Cela peut servir par exemple à faire plusieurs figures dans le graphique en cours.

```
\begin{luadraw}{name=viewport_changewin}
local ld = luadraw
local g = ld.graph:new{window={-5,5,-5,5},size={10,10}}
local i, Z = ld.cpx.I, ld.cpx.Z
g:Labelsize("tiny")
local styleA = "\\tikzset{->-/.style={decoration={markings, mark="
local styleB = "at position #1 with {\\arrow{>}}}, postaction={decorate}}}"
g:Writeln(styleA..styleB)
g:Dline({0,1},"dashed,gray"); g:Dline({0,i},"dashed,gray")
local legende = {"Point ordinaire", "Point d'inflexion", "Rebroussement première espèce",
"Rebroussement seconde espèce"}
local A, B, C =(1+i)*0.75, 0.75, 0
local A2, B2 ={-1.25+i*0.5,-0.75-i*0.5,1.25-0.5*i, 0.5+i}, {-0.75,-0.75,0.75,0.75}
local u = {Z(-5,0),Z(0,0),-5-5*i,-5*i}
for k = 1, 4 do
    g:Saveattr(); g:Viewport(u[k].re,u[k].re+5,u[k].im,u[k].im+5)
    g:Coordsystem(-1.4,2.25,-1,1.25)
    g:Composematrix({0,1,1+i}) -- pour pencher l'axe Oy
    g:Dpolyline({{-1,1},{-i*0.5,i}}) -- axes
    g:Lineoptions(nil,"blue",8)
    g:Dpath({A2[k],(B2[k]+2*A2[k])/3,(C+5*B2[k])/6, C,"b"},"->-=0.5")
    g:Dpath({C,(C+5*B)/6,(B+2*A)/3,A,"b"},"->-=0.75")
    g:Dpolyline({{0,0.75},{0,0.75*i}},false,"->,red")
    g:Dlabel(
    legende[k],0.75-0.5*i, {pos="S"},
        "$f^{(p)}(t_0)$",1,{pos="E",node_options="red"},
        "$f^{(q)}(t_0)$",0.75*i,{pos="W",dist=0.05})
    g:Restoreattr()
end
g:Show()
\end{luadraw}
```

FIGURE 39 : Classification des points d'une courbe paramétrée



VII Ajouter ses propres méthodes à la classe ld.graph

Sans avoir à modifier les fichiers sources Lua associés au paquet *luadraw*, on peut ajouter ses propres méthodes à la classe *ld.graph*, ou modifier une méthode existante. Ceci n'a d'intérêt que si ces modifications doivent être utilisées dans différents graphiques et/ou différents documents (sinon il suffit d'écrire localement une fonction dans le graphique où on en a besoin).

1) Un exemple

Dans le graphique de la page 18, nous avons dessiné un champ de vecteurs, pour cela on a écrit une fonction qui calcule les vecteurs avant de faire le dessin, mais cette fonction est locale. On pourrait en faire une fonction globale dans l'espace de noms *luadraw*, elle serait alors utilisable dans tout le document, mais pas dans un autre document!

Pour généraliser cette fonction, on va devoir créer un fichier Lua qui pourra ensuite être importé dans des documents en cas de besoin. Pour rendre l'exemple un peu consistant, on va créer un fichier qui va définir une fonction qui calcule les vecteurs d'un champ, et qui va ajouter à la classe *ld.graph* deux nouvelles méthodes : une pour dessiner un champ de vecteurs d'une fonction $f:(x,y)\to f(x,y)\in\mathbf{R}^2$, on la nommera **ld.graph:Dvectorfield**, et une autre pour dessiner un champ de gradient d'une fonction $f:(x,y)\to f(x,y)\in\mathbf{R}$, on la nommera **ld.graph:Dgradientfield**. Du coup nous appellerons ce fichier : *luadraw_fields.lua*.

Contenu du fichier :

```
local ld = luadraw
local graph = ld.graph
local cpx = ld.cpx
local Z = cpx.Z

function ld.field(f,x1,x2,y1,y2,grid,long)
-- champ de vecteurs dans le pavé [x1,x2]x[y1,y2]
-- f fonction de deux variables à valeurs dans R^2
-- grid = {nbx, nby} : nombre de vecteurs suivant x et suivant y
-- long = longueur d'un vecteur
    if grid == nil then grid = {25,25} end
    local deltax, deltay = (x2-x1)/(grid[1]-1), (y2-y1)/(grid[2]-1) -- pas suivant x et y
    if long == nil then long = math.min(deltax,deltay) end -- longueur par défaut
```

```
    local vectors = {} -- contiendra la liste des vecteurs
    local x, y, v = x1
    for _ = 1, grid[1] do -- parcours suivant x
        y = y1
        for _ = 1, grid[2] do -- parcours suivant y
            v = f(x,y) -- on suppose que v est bien défini
            v = Z(v[1],v[2]) -- passage en complexe
            v = cpx.normalize(v)
            if v ~= nil then
                table.insert(vectors, {Z(x,y)-long/2*v, Z(x,y)+long/2*v} ) -- on ajoute le vecteur
            end
            y = y+deltay
        end
        x = x+deltax
    end
    return vectors -- on renvoie le résultat (ligne polygonale)
end

function graph:Dvectorfield(f,args)
-- dessine un champ de vecteurs
-- f fonction de deux variables à valeurs dans R^2
-- args table à 4 champs :
-- { view={x1,x2,y1,y2}, grid={nbx,nby}, length=, draw_options=""}
    args = args or {}
    local view = args.view or {self:Xinf(),self:Xsup(),self:Yinf(),self:Ysup()} -- repère utilisateur par défaut
    local vectors = ld.field(f,view[1],view[2],view[3],view[4],args.grid,args.length) -- calcul du champ
    self:Dpolyline(vectors,false,args.draw_options,view) -- le dessin (ligne polygonale non fermée)
end

function graph:Dgradientfield(f,args)
-- dessine un champ de gradient
-- f fonction de deux variables à valeurs dans R
-- args table à 4 champs :
-- { view={x1,x2,y1,y2}, grid={nbx,nby}, length=, draw_options=""}
    local h = 1e-6
    local grad_f = function(x,y) -- fonction gradient de f
        return { (f(x+h,y)-f(x-h,y))/(2*h), (f(x,y+h)-f(x,y-h))/(2*h) }
    end
    self:Dvectorfield(grad_f,args) -- on utilise la méthode précédente
end
```

2) Comment importer le fichier

Il y a deux méthodes pour cela :

1. Avec l'instruction Lua *dofile*. On peut l'écrire par exemple dans le préambule après la déclaration du paquet :

```
\usepackage[]{luadraw}
\directlua{dofile("<chemin>/luadraw_fields.lua")}
```

   Bien entendu, il faudra remplacer `<chemin>` par le chemin d'accès à ce fichier.

   L'instruction `\directlua{dofile("<chemin>/luadraw_fields.lua")}` peut être placée ailleurs dans le document pourvu que ce soit après le chargement du paquet (sinon la classe *ld.graph* ne sera pas reconnue lors de la lecture du fichier). On peut aussi placer l'instruction `dofile("<chemin>/luadraw_fields.lua")` dans un environnement *luacode*, et donc en particulier dans un environnement *luadraw*.

   Dès que le fichier est importé, les nouvelles méthodes sont disponibles pour la suite du document.

   Cette façon de procéder a au moins deux inconvénients : il faut se souvenir du chemin à chaque utilisation, et d'autre part l'instruction *dofile* ne vérifie par si le fichier a déjà été lu. Pour ces raisons, on préférera la méthode suivante.

2. Avec l'instruction Lua *require*. On peut l'écrire par exemple dans le préambule après la déclaration du paquet :

```
\usepackage[]{luadraw}
\directlua{require "luadraw_fields"}
```

On remarquera l'absence du chemin (et l'extension lua est inutile).

L'instruction `\directlua{require "luadraw_fields"}` peut être placée ailleurs dans le document pourvu que ce soit après le chargement du paquet (sinon la classe *graph* ne sera pas reconnue lors de la lecture du fichier). On peut aussi placer l'instruction `require "luadraw_fields"` dans un environnement *luacode*, et donc en particulier dans un environnement *luadraw*.

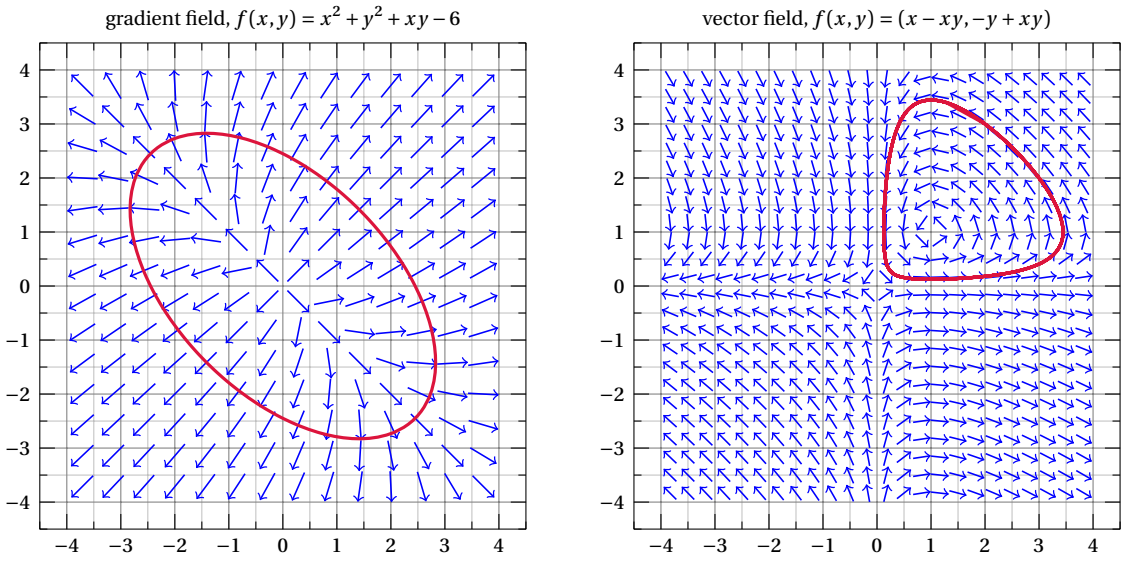
L'instruction *require* vérifie si le fichier a déjà été chargé ou non, ce qui est préférable. Mais il faut cependant que Lua soit capable de trouver ce fichier, et le plus simple pour cela est qu'il soit quelque part dans une arborescence connue de TeX. On peut par exemple créer dans son *texmf* local le chemin suivant :

```
texmf/tex/lualatex/myluafiles/
```

puis copier le fichier *luadraw_fields.lua* dans le dossier *myluafiles*.

```
\begin{luadraw}{name=fields}
require "luadraw_fields" -- import des nouvelles méthodes
local ld = luadraw
local g = ld.graph:new{window={0,21,0,10},size={16,10}}
local i = ld.cpx.I
g:Labelsize("footnotesize")
local f = function(x,y) return {x-x*y,-y+x*y} end -- Volterra
local F = function(x,y) return x^2+y^2+x*y-6 end
local H = function(t,Y) return f(Y[1],Y[2]) end
-- graphique du haut
g:Saveattr();g:Viewport(0,10,0,10);g:Coordsystem(-5,5,-5,5)
g:Dgradbox({-4.5-4.5*i,4.5+4.5*i,1,1}, {originloc=0,originnum={0,0},grid=true,
    title="gradient field, $f(x,y)=x^2+y^2+xy-6$"})
g:Arrows("->"); g:Lineoptions(nil,"blue",6)
g:Dgradientfield(F,{view={-4,4,-4,4},grid={15,15},long=0.5})
g:Arrows("-"); g:Lineoptions(nil,"Crimson",12); g:Dimplicit(F, {view={-4,4,-4,4}})
g:Restoreattr()
-- graphique du bas
g:Saveattr();g:Viewport(11,21,0,10);g:Coordsystem(-5,5,-5,5)
g:Dgradbox({-4.5-4.5*i,4.5+4.5*i,1,1}, {originloc=0,originnum={0,0},grid=true,
    title="vector field, $f(x,y)=(x-xy,-y+xy)$"})
g:Arrows("->"); g:Lineoptions(nil,"blue",6); g:Dvectorfield(f,{view={-4,4,-4,4}})
g:Arrows("-");g:Lineoptions(nil,"Crimson",12)
g:Dodesolve(H,0,{2,3},{t={0,50},out={2,3},nbdots=250})
g:Restoreattr()
g:Show()
\end{luadraw}
```

FIGURE 40 : Utilisation des nouvelles méthodes



3) Modifier une méthode existante

Prenons par exemple la méthode **g:DplotXY(X, Y [, draw_options])** qui prend comme arguments deux listes (tables) de réels et dessine la ligne polygonale formée par les points de coordonnées (X[k],Y[k]). Nous allons la modifier afin qu'elle prenne en compte le cas où $\langle X\rangle$ est une liste de noms (chaînes), dans ce cas, on affichera les noms sous l'axe des abscisses (avec l'abscisse k pour le k^{e} nom) et on dessinera la ligne polygonale formée par les points de coordonnées (k,Y[k]), sinon on fera comme l'ancienne méthode. Il suffit pour cela de réécrire la méthode (dans un fichier Lua pour pouvoir ensuite l'importer) :

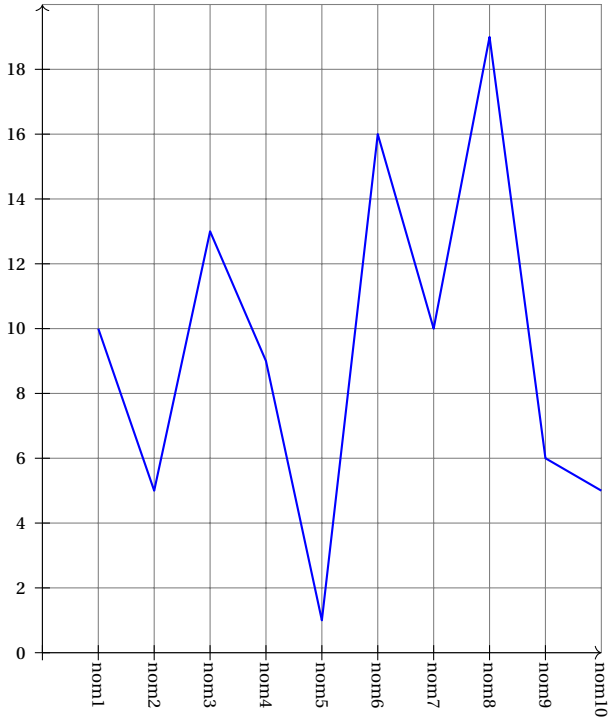
```
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

function ld.graph:DplotXY(X,Y,draw_options)
-- X est une liste de réels ou de chaînes
-- Y est une liste de réels de même longueur que X
    local L = {} -- liste des points à dessiner
    if type(X[1]) == "number" then -- liste de réels
        for k,x in ipairs(X) do
            table.insert(L,Z(x,Y[k]))
        end
    else
        local noms = {} -- liste des labels à placer
        for k = 1, #X do
            table.insert(L,Z(k,Y[k]))
            insert(noms,{X[k],k,{pos="E",node_options="rotate=-90"}})
        end
        self:Dlabel(table.unpack(noms)) --dessin des labels
    end
    self:Dpolyline(L,draw_options) -- dessin de la courbe
end
```

Dès que le fichier sera importé, cette nouvelle définition va écraser l'ancienne (pour toute la suite du document). Bien entendu on pourrait imaginer ajouter d'autres options sur le style de tracé par exemple (ligne, bâtons, points ...).

```
\begin{luadraw}{name=newDplotXY}
require "luadraw_fields" -- import de la méthode modifiée
local ld = luadraw
local g = ld.graph:new{window={-0.5,11,-1,20}, margin={0.5,0.5,0.5,1}, size={10,10,0}}
g:Labelsize("scriptsize")
local X, Y = {}, {} -- on définit deux listes X et Y, on pourrait aussi les lire dans un fichier
for k = 1, 10 do
    table.insert(X,"nom"..k)
    table.insert(Y,math.random(1,20))
end
g:Daxes({0,1,2},{limits={{0,10},{0,20}}, labelpos={"none","left"},
    labelshift={0,0}, originpos={"none","center"}, arrows="->", grid=true})
g:DplotXY(X,Y,"line width=0.8pt, blue")
g:Show()
\end{luadraw}
```

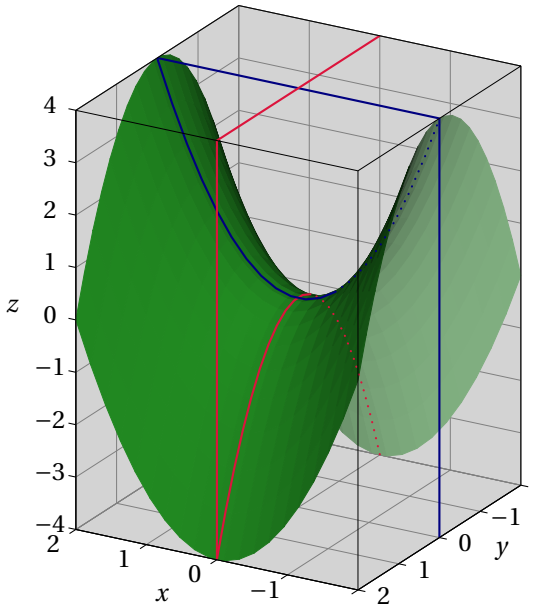
FIGURE 41 : Modification d'une méthode existante



Chapitre 2

Dessin 3D

FIGURE 1 : Point col en M$(0,0,0)$ ($z = x^2 - y^2$)



I Introduction

Rappel, nous utilisons les raccourcis suivants :

```
local ld = luadraw -- alias sur l'espace de nommage
local cpx = ld.cpx -- raccourci pour la classe cpx
local pt3d = ld.pt3d -- raccourci pour la classe pt3d
```

1) Prérequis

- Ce chapitre présente l'utilisation du package *luadraw* avec l'option globale *3d* : `\usepackage[3d]{luadraw}`.
- Le paquet charge le module *luadraw_graph2d.lua* qui définit la classe *ld.graph*, et fournit l'environnement *luadraw* qui permet de faire des graphiques en Lua. Tout ce qui est dit dans le précédent chapitre (Dessin 2D) s'applique donc, et est supposé connu ici.
- L'option globale *3d* permet en plus le chargement du module *luadraw_graph3d.lua*. Celui-ci définit en plus la classe *ld.graph3d* (qui s'appuie sur la classe *ld.graph*) pour des dessins en 3D.

2) Quelques rappels

- Autre option globale du paquet : `noexec`. Lorsque cette option globale est mentionnée la valeur par défaut de l'option `exec` pour l'environnement *luadraw* sera `false` (et non plus `true`).

- Lorsqu'un graphique est terminé il est exporté au format TikZ, donc ce paquet charge également le paquet *tikz* ainsi que les librairies :
  - *patterns, patterns.meta*
  - *plotmarks*
  - *arrows.meta, bending*
  - *decorations.markings*
  - *backgrounds*
- Les graphiques sont créés dans un environnement *luadraw*, celui-ci appelle *luacode*, c'est donc du **langage Lua** qu'il faut utiliser dans cet environnement.
- Sauvegarde du fichier **.tkz* : le graphique est exporté au format TikZ dans un fichier (avec l'extension *tkz*), par défaut celui-ci est sauvegardé dans le dossier *_luadraw* qui est un sous-dossier du dossier courant (contenant le document maître), mais il est possible d'imposer un chemin vers un autre sous-dossier avec l'option globale `cachedir=...`.
- Les options de l'environnement sont :
  - `name=...` : permet de donner un nom au fichier TikZ produit, on donne un nom sans extension (celle-ci sera automatiquement ajoutée, c'est *.tkz*). Si cette option est omise, alors il y a un nom par défaut, qui est le nom du fichier maître suivi d'un numéro.
  - `exec=true/false` : permet d'exécuter ou non le code Lua compris dans l'environnement. Par défaut cette option vaut `true`, **SAUF** si l'option globale `noexec` a été mentionnée dans le préambule avec la déclaration du paquet. Lorsqu'un graphique complexe qui demande beaucoup de calculs est au point, il peut être intéressant de lui ajouter l'option `exec=false`, cela évitera les recalculs de ce même graphique pour les compilations à venir.
  - `auto=true/false` : permet d'inclure ou non automatiquement le fichier TikZ en lieu et place de l'environnement *luadraw* lorsque l'option `exec` est à `false`. Par défaut l'option `auto` vaut `true`.

3) Création d'un graphe 3D

```
\begin{luadraw}{ name=<filename>, exec=true/false, auto=true/false }
local ld = luadraw
-- création d'un nouveau graphique en lui donnant un nom local
local g = ld.graph3d:new{ window3d={x1,x2,y1,y2,z1,z2 [,xscale,yscale,zscale]}, adjust2d=true/false, viewdir={30,60},
 ↪  window={x1,x2,y1,y2 [,xscale,yscale]}, margin={top,right,bottom,left}, size={largeur,hauteur,ratio}, bg="color",
 ↪  border=true/false }
-- construction du graphique g
    instructions graphiques en langage Lua ...
-- affichage du graphique g et sauvegarde dans le fichier <filename>.tkz
g:Show()
-- ou bien sauvegarde uniquement dans le fichier <filename>.tkz
g:Save()
\end{luadraw}
```

Important : dans toute la suite de ce chapitre, les points de l'espace sont appelés *points 3D*, ce sont des triplets de $\mathbf{R}^3$. Dans l'environnement *luadraw*, le point 3D (x,y,z) sera noté **pt3d.M(x, y, z)** (**pt3d.M** est une fonction de la classe *pt3d* qui crée et renvoie un point 3D).

La création se fait dans un environnement *luadraw*, c'est à la première ligne à l'intérieur de l'environnement qu'est faite cette création en nommant le graphique :

```
local ld = luadraw
local g = ld.graph3d:new{ window3d={x1,x2,y1,y2,z1,z2 [,xscale,yscale,zscale]}, adjust2d=true/false, viewdir={30,60},
 ↪  window={x1,x2,y1,y2 [,xscale,yscale]}, margin={left,right,top,bottom}, size={largeur,hauteur,ratio}, bg="color",
 ↪  border=true/false }
```

La classe *ld.graph3d* est définie dans le paquet *luadraw* grâce à l'option globale `3d`. On instancie cette classe en invoquant son constructeur et en donnant un nom (ici c'est *g*), on le fait en local de sorte que le graphique *g* ainsi créé, n'existera plus une fois sorti de l'environnement (sinon *g* resterait en mémoire jusqu'à la fin du document).

- L'option `window3d={x1,x2,y1,y2,z1,z2 [,xscale,yscale,zscale]}` définit le pavé de $\mathbf{R}^3$ correspondant au graphique : c'est $[x_1;x_2]\times[y_1;y_2]\times[z_1;z_2]$, ainsi que l'échelle sur les trois axes : *xscale*, *yscale* et *zscale*, celles-ci sont facultatives et valent 1 par défaut. Le pavé par défaut est $[-5;5]\times[-5;5]\times[-5;5]$.

Attention : les trois échelles déterminent la matrice 3D initiale du graphe, lorsque l'une d'elle est différente de 1, cette matrice n'est pas l'identité. Si vous devez changer la matrice 3D du graphe par la suite, il faut utiliser la méthode **g:Composematrix3d()** et non pas **g:Setmatrix3d()**, et penser à sauvegarder auparavant la matrice initiale avec la méthode **g:Savematrix()**, on peut ensuite la restaurer avec la méthode **g:Restorematrix()**.

- L'option `adjust2d` indique si la fenêtre 2D qui va contenir la projection du dessin 3D, doit être déterminée automatiquement (`false` par défaut). Cette fenêtre 2D correspond à l'option `window`.
- L'option `viewdir` est une table qui définit le mode de projection et les deux angles de vue (en degrés). Par défaut `viewdir={"ortho",30,60}` (projection orthographique). La figure suivante montre à quoi correspondent ces deux angles.

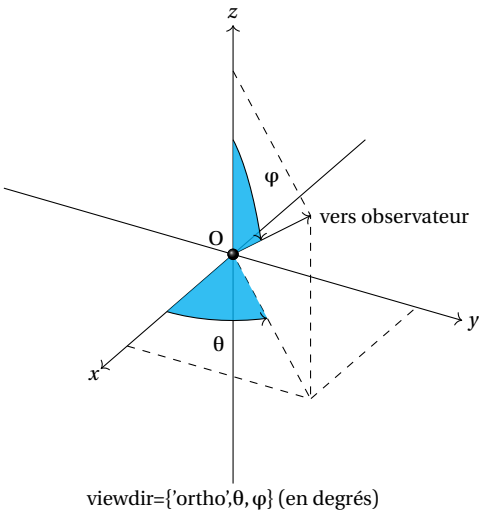


FIGURE 2 : Angles de vue

- Les autres options sont celles de la classe *ld.graph*, elles ont été décrites dans le chapitre 1.

Construction du graphique.

- L'objet instancié possède toutes les méthodes de la classe *ld.graph*, plus des méthodes spécifiques à la 3D.
- La classe *ld.graph3d* amène aussi un certain nombre de fonctions mathématiques propres à la 3D.

4) Modes de projection affine

- Projections orthogonales :
  - `viewdir={"ortho",theta,phi}`, ou `viewdir={theta,phi}` : projection orthographique (projection orthogonale sur l'écran), c'est la projection par défaut avec theta=30 et phi=60 (degrés),
  - `viewdir="xOy"` : projection sur le plan xy (observateur du côté des z positifs),
  - `viewdir="yOx"` : projection sur le plan xy (observateur du côté des z négatifs),
  - `viewdir="xOz"` : projection sur le plan xz (observateur du côté des y positifs),
  - `viewdir="zOx"` : projection sur le plan xz (observateur du côté des y négatifs),
  - `viewdir="yOz"` : projection sur le plan yz (observateur du côté des x positifs),
  - `viewdir="zOy"` : projection sur le plan yz (observateur du côté des x négatifs),
- Projections non orthogonales :
  - `viewdir={"yz",k,alpha}` : perspective cavalière sur le plan yz (observateur du côté des x positifs),
  - `viewdir={"zy",k,alpha}` : perspective cavalière sur le plan yz (observateur du côté des x négatifs),
  - `viewdir={"xz",k,alpha}` : perspective cavalière sur le plan xz (observateur du côté des y positifs),
  - `viewdir={"zx",k,alpha}` : perspective cavalière sur le plan xz (observateur du côté des y négatifs),
  - `viewdir={"xy",k,alpha}` : perspective cavalière sur le plan xy (observateur du côté des z positifs),
  - `viewdir={"yx",k,alpha}` : perspective cavalière sur le plan xy (observateur du côté des z négatifs),
  - `viewdir="iso"` : perspective isométrique (observateur du côté des x et y positifs),
  - `viewdir="osi"` : perspective isométrique (observateur du côté des x et y négatifs).

Les perspectives cavalières sont définies à l'aide de deux paramètres : un nombre positif *k* et un angle en degré *alpha*, qui sont mis en évidence dans la figure suivante.

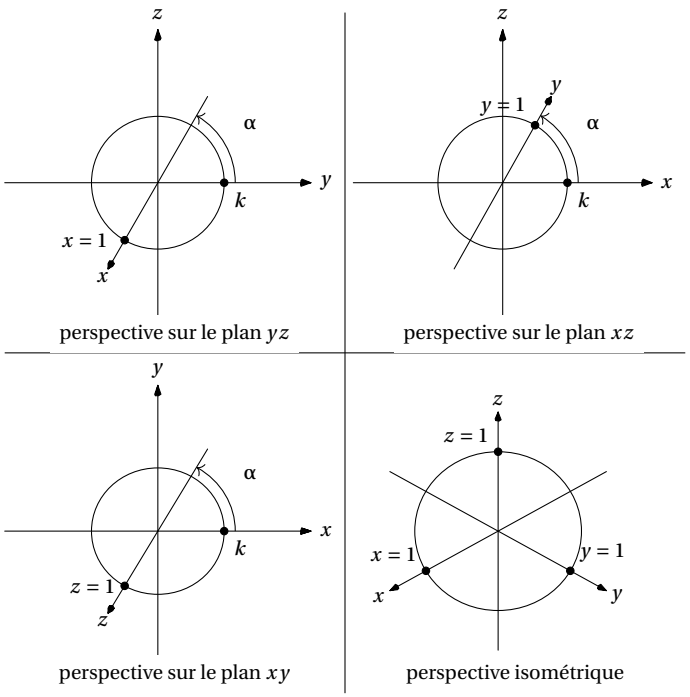


FIGURE 3 : Modes de projection affine

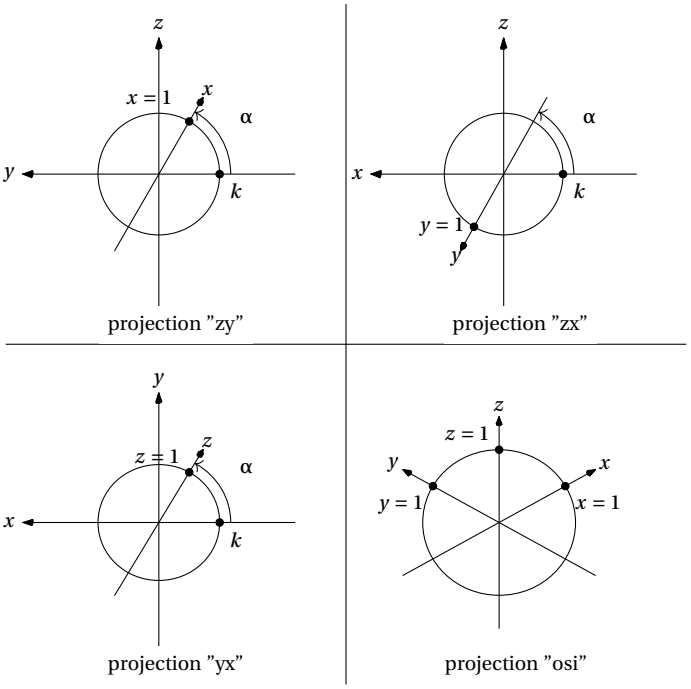


FIGURE 4 : Modes "symétriques" des précédents

Toutes les valeurs qui peuvent être transmises à l'option `viewdir` lors la création, peuvent également être utilisées dans la méthode **g:Setviewdir()** en cours de graphique.

5) Projection centrale

Depuis la version 2.4, *luadraw* propose également la projection centrale. À la différence des modes précédents, **cette projection n'est pas affine**, et d'autre part elle n'est pas définie pour tous les points de l'espace, ce qui peut conduire à des erreurs, cela demande donc de la réflexion et des ajustements. Cette projection est définie par :

- Une caméra, qui est un point de l'espace mémorisé dans une variable globale appelée `ld.camera` et qui ne doit pas être modifiée directement.
- Une cible, qui est un point de l'espace mémorisé dans une variable globale appelée `ld.target` et qui ne doit pas être modifiée directement.

Le plan passant par le point `ld.target` et orthogonal à l'axe `ld.target - ld.camera` est le plan de la projection, il représente l'écran. Comme pour les modes précédents, la projection centrale est accessible par l'option `viewdir` ou la méthode **g:Setviewdir** :

viewdir = {"central" [, camera, target]}

ou bien

viewdir = {"central" [, theta, phi, d, target]}

Dans le premier cas on donne les valeurs de ⟨*camera*⟩ et ⟨*target*⟩ (points 3D, par défaut ⟨*target*⟩ est l'origine). Dans le second cas, les trois arguments ⟨*theta*⟩, ⟨*phi*⟩, et ⟨*d*⟩ servent à positionner la caméra conformément au schéma suivant :

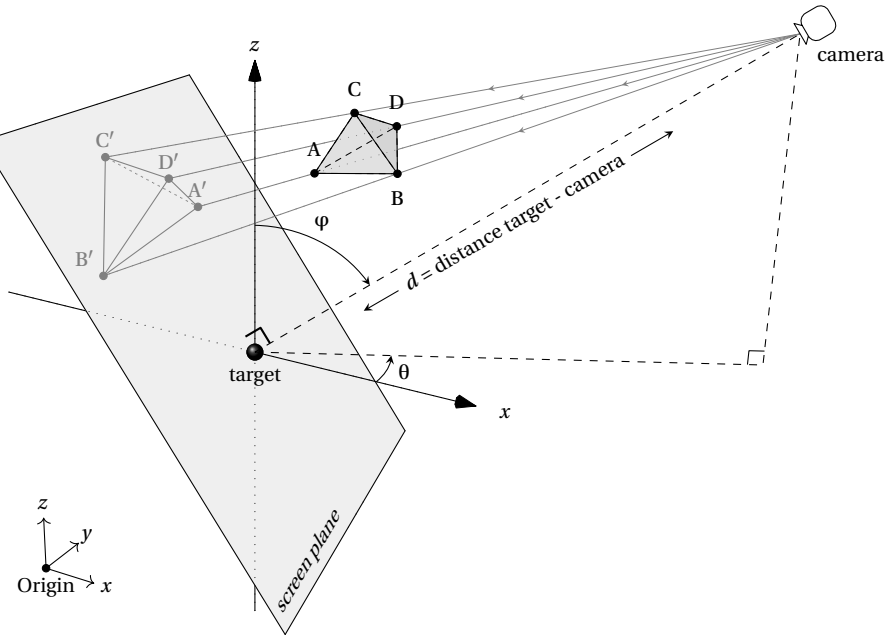


FIGURE 5 : Projection centrale

Les valeurs par défaut sont : ⟨*theta*⟩=30 (degrés), ⟨*phi*⟩=60, ⟨*d*⟩=15, ⟨*target*⟩=`pt3d.Origin` (la variable globale `pt3d.Origin` est le point 3D $(0,0,0)$).

II La classe pt3d

1) Représentation des points et vecteurs

L'espace usuel est $\mathbf{R}^3$, les points et les vecteurs sont donc des triplets de réels, on les appellera : points 3D. La classe *pt3d* (qui est automatiquement chargée) permet de gérer les points 3D, les opérations possibles, et un certain nombre de méthodes et de constantes.

- Quatre triplets portent un nom spécifique (variables globales), il s'agit de :
  - `pt3d.Origin`, qui représente le triplet $(0,0,0)$.
  - `pt3d.vecI`, qui représente le triplet $(1,0,0)$.
  - `pt3d.vecJ`, qui représente le triplet $(0,1,0)$.
  - `pt3d.vecK`, qui représente le triplet $(0,0,1)$.
- Pour créer un point 3D, il y a trois méthodes :
  - Définition en cartésien : la fonction **pt3d.M(x, y, z)** renvoie le triplet (x,y,z). On peut également obtenir ce triplet en faisant : `x*vecI+y*vecJ+z*vecK`.
  - Définition en cylindrique : la fonction **pt3d.Mc(r, theta, z)** (angle exprimé en radians) renvoie le triplet $(r\cos(\theta), r\sin(\theta), z)$.
  - Définition en sphérique : la fonction **pt3d.Ms(r, theta, phi)** renvoie le triplet $(r\cos(\theta)\sin(\varphi), r\sin(\theta)\sin(\varphi), r\cos(\varphi))$ (angles exprimés en radians).

Accès aux composantes d'un point 3D : si une variable A désigne un point 3D, alors ses trois composantes sont A.x, A.y et A.z.

Pour tester si une variable A désigne un point 3D, on dispose de la fonction **pt3d.isPoint3d()** qui renvoie un booléen.

Conversion : pour convertir un réel ou un complexe en point 3D, on dispose de la fonction **pt3d.toPoint3d()**.

2) Opérations sur les points 3D

Ces opérations sont les opérations usuelles avec les symboles usuels :

- L'addition (+), la différence (-), l'opposé (-).
- Le produit par un scalaire, si k et un réel, `k*M(x,y,z)` renvoie `M(k*x,k*y,k*z)`.
- On peut diviser un point 3D par un scalaire, par exemple, si A et B sont deux points 3D, alors le milieu s'écrit simplement (A + B)/2.
- On peut tester l'égalité de deux points 3D avec le symbole =.

3) Méthodes de la classe *pt3d*

Celles-ci sont :

- **pt3d.abs(u)** : renvoie la norme euclidienne du point 3D ⟨u⟩.
- **pt3d.abs2(u)** : renvoie la norme euclidienne au carré du point 3D ⟨u⟩.
- **pt3d.N1(u)** : renvoie la norme 1 du point 3D u. Si $u = \mathrm{M}(x,y,z)$, alors `pt3d.N1(u)` renvoie $|x|+|y|+|z|$.
- **pt3d.dot(u, v)** : renvoie le produit scalaire entre les vecteurs (points 3D) ⟨u⟩ et ⟨v⟩.
- **pt3d.det(u, v, w)** : renvoie le déterminant entre les vecteurs (points 3D) ⟨u⟩, ⟨v⟩ et ⟨w⟩.
- **pt3d.prod(u, v)** : renvoie le produit vectoriel entre les vecteurs (points 3D) ⟨u⟩ et ⟨v⟩.
- **pt3d.angle3d(u, v [, epsilon])** : renvoie l'écart angulaire (en radians) entre les vecteurs (points 3D) ⟨u⟩ et ⟨v⟩. supposés non nuls. L'argument (facultatif) ⟨*epsilon*⟩ vaut 0 par défaut, il indique à combien près se fait un certain test d'égalité sur un flottant.
- **pt3d.normalize(u)** : renvoie le vecteur (point 3D) ⟨u⟩ normalisé (renvoie `nil` si ⟨u⟩ est nul).
- **pt3d.round(u, nbDeci)** : renvoie un point 3D dont les composantes sont celles du point 3D ⟨u⟩ arrondies avec ⟨*nbDeci*⟩ décimales.
- **pt3d.isobar3d(L)** : renvoie l'isobarycentre des points 3D de la liste (table) ⟨L⟩ (les éléments de ⟨L⟩ qui ne sont pas des points 3D sont ignorés).
- **pt3d.insert3d(L, A [, epsilon])** : cette fonction insère le point 3D ⟨A⟩ dans la liste ⟨L⟩ qui doit être une **variable** (et qui sera donc modifiée). Le point ⟨A⟩ est inséré **sans doublon** et la fonction renvoie sa position (indice) dans la liste ⟨L⟩ après insertion. L'argument (facultatif) ⟨*epsilon*⟩ vaut 0 par défaut, il indique à combien près se font les comparaisons.

4) Afficher une variable dans le terminal

L'instruction **ld.whatis(variable [, msg])** affiche dans le terminal lors de la compilation, le type de la ⟨*variable*⟩ ainsi que son contenu. Les types reconnus sont, les types prédéfinis plus : *complex number*, *list of (complex) numbers*, *list of lists of (complex) numbers*, *3D point*, *list of 3D points*, *list of lists of 3D points*. L'argument ⟨*msg*⟩ est une chaîne optionnelle (vide par défaut) qui est affichée avec le type pour repérer la variable dans le terminal.

III Méthodes graphiques élémentaires

Toutes les méthodes graphiques 2D s'appliquent. À cela s'ajoute la possibilité de dessiner dans l'espace des lignes polygonales, des segments, droites, courbes, chemins, points, labels, plans, solides. Avec les solides vient également la notion de facettes que l'on ne trouvait pas en 2D.

Les méthodes graphiques 3D vont calculer automatiquement la projection sur le plan de l'écran, après avoir appliquer aux objets la matrice de transformation 3D associée au graphique (qui est l'identité par défaut), ce sont ensuite les méthodes graphiques 2D qui prendront le relai.

La méthode qui applique la matrice 3D et fait la projection sur l'écran (plan passant par l'origine et normal au vecteur unitaire dirigé vers l'observateur et défini par les angles de vue), est : **g:Proj3d(L)** où ⟨*L*⟩ est soit un point 3D, soit une liste de points 3D, soit une liste de listes de points 3D. Cette fonction renvoie des complexes (affixes des projetés sur l'écran).

Attention : lorsque la matrice 3D du graphe est une transformation affine non linéaire, le projeté sur l'écran d'un vecteur *u* de l'espace n'est pas `g:Proj3d(u)`, mais `g:Proj3d(A+u)-g:Proj3d(A)` où A désigne un point quelconque de l'espace. Pour éviter ces calculs, la méthode **g:Proj3dV()** a été introduite, elle fait la projection des **vecteurs** sur l'écran, et renvoie des complexes (affixes des projetés sur l'écran).

Mise en garde pour les couleurs : dans un certain nombre de situations la couleur donnée par l'utilisateur est légèrement modifiée (par exemple pour les facettes en fonction de leur inclinaison), de ce fait la couleur doit être soit un nom de couleur, soit une table au format {r,g,b}, soit une chaîne contenant une couleur déjà modifiée mais à condition que la syntaxe soit complète, par exemple la syntaxe `"orange!50"` n'est pas complète et risque de provoquer une erreur, la syntaxe correcte est `"orange!50!white"` (qui ne provoquera pas d'erreur).

1) Dessin aux traits

Ligne polygonale : Dpolyline3d

La méthode **g:Dpolyline3d(L [, close, draw_options, clip])** (où *g* désigne le graphique en cours de création), ⟨*L*⟩ est une ligne polygonale 3D (liste de listes de points 3D), ⟨*close*⟩ un argument facultatif qui vaut `true` ou `false`, indiquant si la ligne doit être refermée ou non (`false` par défaut), et ⟨*draw_options*⟩ est une chaîne de caractères qui sera passée directement à l'instruction *\draw* dans l'export. L'argument ⟨*clip*⟩ vaut `false` par défaut, il indique si la ligne ⟨*L*⟩ doit être clippée avec la fenêtre 3D courante.

Angle droit : Dangle3d

La méthode **g:Dangle3d(B, A, C [, r, draw_options, clip])** dessine l'angle BAC avec un parallélogramme (deux côtés seulement sont dessinés), l'argument facultatif ⟨*r*⟩ précise la longueur d'un côté (0.25 par défaut). Le parallélogramme est dans le plan défini par les points ⟨*A*⟩, ⟨*B*⟩ et ⟨*C*⟩, ceux-ci ne doivent donc pas être alignés. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*. L'argument ⟨*clip*⟩ vaut `false` par défaut, il indique si le tracé doit être clippé avec la fenêtre 3D courante.

Segment : Dseg3d, Dmarkseg3d

La méthode **g:Dseg3d(seg [, scale, draw_options, clip])** dessine le segment défini par l'argument ⟨*seg*⟩ qui doit être une liste de deux points 3D. L'argument facultatif ⟨*scale*⟩ (1 par défaut) est un nombre qui permet d'augmenter ou réduire la longueur du segment (la longueur naturelle est multipliée par ⟨*scale*⟩). L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*. L'argument ⟨*clip*⟩ vaut `false` par défaut, il indique si le tracé doit être clippé avec la fenêtre 3D courante.

La méthode **g:Dmarkseg3d(A, B, V, n, [long, espace, angle, draw_options])** permet de marquer le segment défini par les points 3D ⟨*A*⟩ et ⟨*B*⟩ avec ⟨*n*⟩ petits segments penchés de ⟨*angle*⟩ degrés (45° par défaut). Le tracé est effectue dans le plan contenant les points ⟨*A*⟩, ⟨*B*⟩ et le vecteur 3D ⟨*V*⟩ (qui ne doit donc pas être colinéaire au vecteur $\vec{AB}$). Par défaut la longueur (argument ⟨*long*⟩) est de 0.25, et l'espacement (argument ⟨*espace*⟩) et de 0.125. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

Droite : Dline3d

La méthode **g:Dline3d(d [, draw_options, clip])** trace la droite ⟨*d*⟩, celle-ci est une liste du type ⟨*d*⟩={A, *u*} où A représente un point de la droite (point 3D) et *u* un vecteur directeur (un point 3D non nul).

Variante : la méthode **g:Dline3d(A, B [, draw_options, clip])** trace la droite passant par les points ⟨*A*⟩ et ⟨*B*⟩ (deux points 3D). L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*. L'argument ⟨*clip*⟩ vaut `false` par défaut, il indique si le tracé doit être clippé avec la fenêtre 3D courante.

La méthode **g:Line3d2seg(d [, scale])** renvoie une table constituée de deux points 3D représentant un segment, ce segment est la partie de la droite ⟨*d*⟩ à l'intérieur la fenêtre 3D courante. L'argument ⟨*scale*⟩ (1 par défaut) permet de faire varier la taille de ce segment. Lorsque la fenêtre est trop petite l'intersection peut être vide.

Arc de cercle : Darc3d

- La méthode **g:Darc3d(B, A, C, r, sens [, normal, draw_options, clip])** dessine un arc de cercle de centre ⟨*A*⟩ (point 3D), de rayon ⟨*r*⟩, allant de ⟨*B*⟩ (point 3D) vers ⟨*C*⟩ (point 3D) dans le sens direct si l'argument ⟨*sens*⟩ vaut 1, le sens inverse sinon. Cet arc est tracé dans le plan contenant les trois points ⟨*A*⟩, ⟨*B*⟩ et ⟨*C*⟩, lorsque ces trois points sont alignés il faut préciser l'argument ⟨*normal*⟩ (point 3D non nul) qui représente un vecteur normal au plan. Ce plan est orienté par le produit vectoriel $\vec{AB} \wedge \vec{AC}$ ou bien par le vecteur ⟨*normal*⟩ si celui-ci est précisé. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*. L'argument ⟨*clip*⟩ vaut `false` par défaut, il indique si le tracé doit être clippé avec la fenêtre 3D courante.
- La fonction **ld.arc3d(B, A, C, r, sens [, normal])** renvoie la liste des points de cet arc (ligne polygonale 3D).
- La fonction **ld.arc3db(B, A, C, r, sens [, normal])** renvoie cet arc sous forme d'un chemin 3D (voir *Dpath3d* page 90) utilisant des courbes de Bézier.

Cercle : Dcircle3d

- La méthode **g:Dcircle3d(I, R, normal [, draw_options, clip])** trace le cercle de centre ⟨*I*⟩ (point 3D) et de rayon ⟨*R*⟩, dans le plan contenant ⟨*I*⟩ et normal au vecteur défini par l'argument ⟨*normal*⟩ (point 3D non nul). L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*. L'argument ⟨*clip*⟩ vaut `false` par défaut, il indique si le tracé doit être clippé avec la fenêtre 3D courante.

  Autre syntaxe possible : **g:Dcircle3d(C [, draw_options, clip])** où ⟨*C*⟩={⟨*I*⟩,⟨*R*⟩,⟨*normal*⟩}.
- La fonction **ld.circle3d(I, R, normal)** renvoie la liste des points de ce cercle (ligne polygonale 3D).
- La fonction **ld.circle3db(I, R, normal)** renvoie ce cercle sous forme d'un chemin 3D (voir *Dpath3d* page 90) utilisant des courbes de Bézier.

Arc d'ellipse : Dellipticarc3d

- La méthode **g:Dellipticarc3d(B, A, C, r1, r2, sens, dir1, [, normal, draw_options, clip])** dessine un arc d'ellipse de centre ⟨*A*⟩ (point 3D), de rayons ⟨*r1*⟩ et ⟨*r2*⟩, allant de ⟨*B*⟩ (point 3D) vers ⟨*C*⟩ (point 3D) dans le sens direct si l'argument ⟨*sens*⟩ vaut 1, le sens inverse sinon. Cet arc est tracé dans le plan contenant les trois points ⟨*A*⟩, ⟨*B*⟩ et ⟨*C*⟩, lorsque ces trois points sont alignés il faut préciser l'argument ⟨*normal*⟩ (point 3D non nul) qui représente un vecteur normal au plan. Ce plan est orienté par le produit vectoriel $\vec{AB} \wedge \vec{AC}$ ou bien par le vecteur ⟨*normal*⟩ si celui-ci est précisé. L'argument ⟨*dir1*⟩ est un vecteur directeur de l'axe portant le rayon ⟨*r1*⟩ (il doit être orthogonal au vecteur ⟨*normal*⟩), le deuxième axe est déterminé par le produit vectoriel entre ⟨*dir1*⟩ et ⟨*normal*⟩.

  L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*. L'argument ⟨*clip*⟩ vaut `false` par défaut, il indique si le tracé doit être clippé avec la fenêtre 3D courante.
- La fonction **ld.ellipticarc3d(B, A, C, r1, r2, sens, dir1, [, normal])** renvoie la liste des points de cet arc (ligne polygonale 3D).
- La fonction **ld.ellipticarc3db(B, A, C, r1, r2, sens, dir1, [, normal])** renvoie cet arc sous forme d'un chemin 3D (voir *Dpath3d* page 90) utilisant des courbes de Bézier.

Ellipse : Dellipse3d

- La méthode **g:Dellipse3d(C, r1, r2, dir1, normal [, draw_options, clip])** dessine l'ellipse de centre ⟨*C*⟩ (point 3D), de rayons ⟨*r1*⟩ et ⟨*r2*⟩. Cette ellipse est tracée dans le plan contenant ⟨*C*⟩ et orthogonal au vecteur ⟨*normal*⟩ (point 3D non nul). L'argument ⟨*dir1*⟩ est un vecteur directeur de l'axe portant le rayon ⟨*r1*⟩ (il doit être orthogonal au vecteur ⟨*normal*⟩), le deuxième axe est déterminé par le produit vectoriel entre ⟨*dir1*⟩ et ⟨*normal*⟩.

  L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*. L'argument ⟨*clip*⟩ vaut `false` par défaut, il indique si le tracé doit être clippé avec la fenêtre 3D courante.
- La fonction **ld.ellipse3d(C, r1, r2, dir1, normal)** renvoie la liste des points de cette ellipse (ligne polygonale 3D).
- La fonction **ld.ellipse3db(C, r1, r2, dir1, normal)** renvoie cette ellipse sous forme d'un chemin 3D (voir *Dpath3d* page 90) utilisant des courbes de Bézier.

Chemin 3D : Dpath3d

La méthode **g:Dpath3d(chemin [, draw_options, clip])** fait le dessin du ⟨*chemin*⟩. L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *draw*. L'argument ⟨*clip*⟩ vaut `false` par défaut, il indique si le tracé doit être clippé avec la fenêtre 3D courante. L'argument ⟨*chemin*⟩ est une liste de points 3D suivis d'instructions (chaînes) fonctionnant **sur le même principe qu'en 2D**. Instructions disponibles et leur syntaxe, le mot *last* représente le dernier point du morceau précédent :

- `p1,"m"` (moveto), permet de commencer une nouvelle composante du chemin au point 3D p_1.
- `p1,...,pn,"l"` (lineto), dessine la ligne polygonale 3D `{last,p1,...,pn}`.
- `c1,c2,p2,"b"` (bézier) dessine la courbe de Bézier `{last,c1,c2,p2}`, où c_1 et c_2 sont les deux points 3D de contrôle.
- `p1,n,"c"` (cercle), dessine le cercle de centre p_1 et passant par le point *last*, et normal au vecteur 3D n.
- `p1,p2,r,sens,n,"ca"` (arc de cercle), dessine un arc de cercle de centre p_1, de rayon r, allant de *last* vers p_2, dans le sens direct lorsque *sens*=1 (et donc dans le sens inverse si *sens*=−1). Le vecteur 3D n est optionnel, il indique un vecteur normal au plan du cercle lorsque les points *last*, p_1 et p_2 sont alignés (et dans ce cas, le vecteur n est obligatoire).
- `p1,r1,r2,dir1,n,"e"` (ellipse), dessine l'ellipse de centre p_1, passant par le point *last*, et normal au vecteur 3D n. Les deux rayons sont r_1 et r_2, et *dir1* est un vecteur directeur de l'axe portant le rayon r_1. Si le point *last* n'est pas exactement sur l'ellipse, alors un segment est tracé entre ce point et l'ellipse.
- `p1,p2,r1,r2,sens,dir1,n,"ea"` (arc d'ellipse), dessine nu arc d'ellipse de centre p_1, allant du point *last* au point p_2, et normal au vecteur 3D n. Les deux rayons sont r_1 et r_2, et *dir1* est un vecteur directeur de l'axe portant le rayon r_1. Le vecteur 3D n est optionnel, il indique un vecteur normal au plan de l'ellipse et oriente celui-ci ; lorsque les points *last*, p_1 et p_2 sont alignés, ce vecteur n est obligatoire.
- `"cl"` (closepath), cette instruction s'utilise seule, elle permet de refermer la composante courante en traçant un segment reliant le dernier point au premier point (de la composante courante).

Voici par exemple le code de la figure 2.

```
\begin{luadraw}{name=viewdir}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK
local g = ld.graph3d:new{ size={8,8} }
local i, M = ld.cpx.I, ld.pt3d.M
local O, A = Origin, M(4,4,4)
local B, C, D, E = ld.pxy(A), ld.px(A), ld.py(A), ld.pz(A) --projeté de A sur le plan xOy et sur les axes
g:Dpolyline3d( {{O,A},{-5*vecI,5*vecI},{-5*vecJ,5*vecJ},{-5*vecK,5*vecK}}, "->") -- axes
g:Dpolyline3d( {{E,A,B,O}, {C,B,D}}, "dashed")
g:Dpath3d( {C,O,B,2.5,1,"ca",O,"l","cl"}, "draw=none,fill=cyan,fill opacity=0.8") --secteur angulaire
g:Darc3d(C,O,B,2.5,1,"->") -- arc de cercle pour theta
g:Dpath3d( {E,O,A,2.5,1,"ca",O,"l","cl"}, "draw=none,fill=cyan,fill opacity=0.8") --secteur angulaire
g:Darc3d(E,O,A,2.5,1,"->") -- arc de cercle pour phi
g:Dballdots3d(O) -- le point origine sous forme d'une petite sphère
g:Labelsize("footnotesize")
g:Dlabel3d(
  "$x$", 5.25*vecI,{}, "$y$", 5.25*vecJ,{}, "$z$", 5.25*vecK,{},
  "vers observateur", A, {pos="E"},
  "$O$", O, {pos="NW"},
  "$\\theta$", (B+C)/2, {pos="N", dist=0.15},
  "$\\varphi$", (A+E)/2, {pos="S",dist=0.25})
g:Dlabel('viewdir=\\{"ortho",$\\theta,\\varphi$\\} (en degrés)',-5*i,{pos="N"}) -- label 2D
g:Show()
\end{luadraw}
```

Fonctions sur les chemins 3D

- La fonction **ld.path3d(chemin [, nbdots])** renvoie une ligne polygonale 3D contenant les points constituant le ⟨*chemin*⟩. L'argument facultatif, ⟨*nbdots*⟩ est le nombre minimal de points calculés pour chaque courbe de Bézier contenue dans le chemin, sa valeur par défaut est la variable globale `ld.bezier_nbdots` qui est initialisée à 12.
- À l'inverse, la fonction **ld.polyline2path3d(L [, close])** où ⟨*L*⟩ est une ligne polygonale 3D (liste de points 3D ou une

liste de listes de points 3D), renvoie ⟨*L*⟩ sous la forme d'un chemin. L'argument optionnel ⟨*close*⟩ permet de refermer ou non chaque composante du chemin (`false` par défaut).

- La méthode **g:Convpath3d(chemin)** convertit le ⟨*chemin*⟩ en un autre chemin n'utilisant que les instructions "m", "l" et "b" (toutes les autres instructions sont transformées en courbes de Bézier). La fonction renvoie le nouveau chemin, l'intérêt de ce nouveau chemin est qu'on peut lui appliquer une matrice de transformation 3D (fonction **ld.mtransform3d()**).
- La méthode **g:Path3d2path2d(chemin)** convertit le chemin 3D ⟨*chemin*⟩ en un chemin 2D (qui est la projection sur l'écran) et renvoie le résultat.

Plan : Dplane

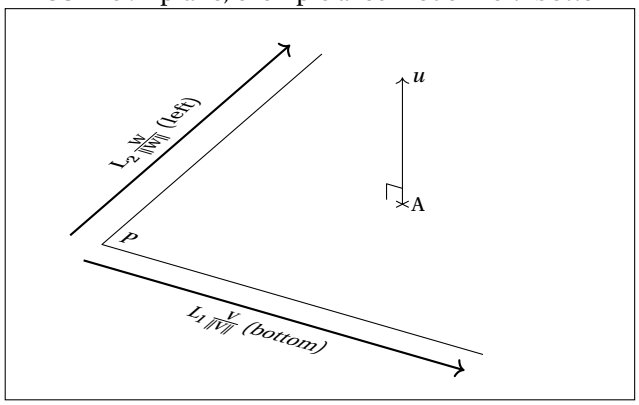
- La méthode **g:Dplane(P, V, L1, L2 [, mode, draw_options])** permet de dessiner les bords du plan ⟨*P*⟩={A, *u*} où A est un point du plan et *u* un vecteur normal au plan (⟨*P*⟩ est donc une table de deux points 3D). L'argument ⟨*V*⟩ doit être un vecteur non nul du plan ⟨*P*⟩, Les arguments ⟨*L1*⟩ et ⟨*L2*⟩ sont deux longueurs. La méthode construit un parallélogramme centré sur A, dont un côté est $L_1 \frac{V}{\|V\|}$ et l'autre $L_2 \frac{W}{\|W\|}$ où $W = u \wedge V$. L'argument ⟨*mode*⟩ est un entier naturel qui indique les bords à tracer. Pour calculer cet entier on utilise les variables prédéfinies : `ld.top` (=8), `ld.right` (=4), `ld.bottom` (=2), `ld.left` (=1) et `ld.all` (=15), que l'on peut ajouter entre elles, par exemple :
  - `mode=ld.bottom+ld.left` : pour les côtés bas et gauche
  - `mode=ld.top+ld.right+ld.bottom` : pour les côtés haut, droit et bas
  - etc.

  Par défaut le mode vaut `ld.all` ce qui correspond à `mode=ld.top+ld.right+ld.bottom+ld.left`.
- La fonction **ld.plane2rectangle(P [, V, L1, L2])** calcule et renvoie le rectangle (liste de points 3D) dessiné par la méthode **g:Dplane()**. Le vecteur ⟨*V*⟩ est facultatif, il permet d'imposer un bord du rectangle (il doit appartenir au plan), s'il est omis la fonction choisira elle-même un vecteur V. Les longueurs ⟨*L1*⟩ et ⟨*L2*⟩ sont facultatives et valent 5 par défaut. Lorsque l'argument ⟨*L2*⟩ et omis, il a implicitement la même valeur que ⟨*L1*⟩. Le résultat peut être dessiné avec la méthode **g:Dpolyline3d()**, ou bien dessiné en tant que facette.

```
\begin{luadraw}{name=Dplane}
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{size={8,8},window={-5.25,3,-2.5,2.5},margin={0,0,0,0},border=true}
local i = cpx.I
g:Labelsize("footnotesize")
local A = Origin
local P = {A, vecK}
g:Dplane(P, vecJ, 6, 6, ld.left + ld.bottom)
g:Dcrossdots3d({A,vecK},nil,0.75)
g:Dseg3d({A,A+2*vecK},"->")
g:Dangle3d(-vecJ,A,vecK,0.25)
g:Dpolyline3d({{M(3.5,-3,0),M(3.5,3,0)},{M(3,-3.5,0), M(-3,-3.5,0)}}, "->,line width=0.8pt")
g:Dlabel3d("$A$",A,{pos="E"},
    "$u$",2*vecK,{},
    "$P$", M(3,-3,0),{pos="NE", dir={vecJ,-vecI}},
    "$L_1\\frac{V}{\\|V\\|}$ (bottom)", M(3.5,0,0), {pos="S"},
    "$L_2\\frac{W}{\\|W\\|}$ (left)", M(0,-3.5,0), {pos="N",dir={-vecI,-vecJ}}
)
g:Show()
\end{luadraw}
```

FIGURE 6 : Dplane, exemple avec mode = left+bottom



Attention : les notions de haut, droite, bas et gauche sont relatives ! Elles dépendent du sens des vecteurs u (vecteur normal au plan) et V (vecteur donné dans le plan). Le troisième vecteur W est le produit vectoriel $u \wedge V$.

Courbe paramétrique : Dparametric3d

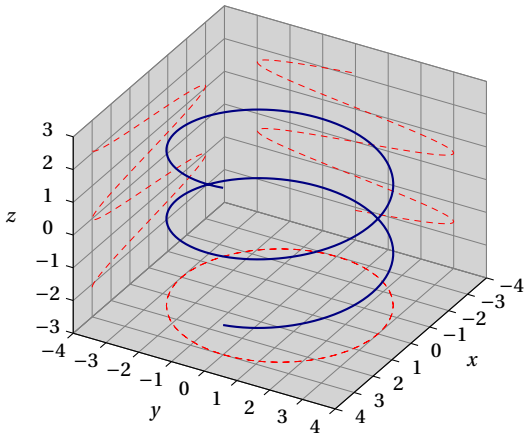
- La fonction **ld.parametric3d(p, t1, t2 [, nbdots, discont, nbdiv])** fait le calcul des points de la courbe et renvoie une ligne polygonale 3D (pas de dessin).
  - L'argument ⟨*p*⟩ est le paramétrage, ce doit être une fonction d'une variable réelle t et à valeurs dans $\mathbf{R}^3$ (les images sont des points 3D), par exemple : `local p=function(t) return Mc(3,t,t/3) end`.
  - Les arguments ⟨*t1*⟩ et ⟨*t2*⟩ sont obligatoires avec $t_1 < t_2$, ils forment les bornes de l'intervalle pour le paramètre.
  - L'argument ⟨*nbdots*⟩ est facultatif, c'est le nombre de points (minimal) à calculer, il vaut 40 par défaut.
  - L'argument ⟨*discont*⟩ est un booléen facultatif qui indique s'il y a des discontinuités ou non, c'est `false` par défaut.
  - L'argument ⟨*nbdiv*⟩ est un entier positif qui vaut 5 par défaut et indique le nombre de fois que l'intervalle entre deux valeurs consécutives du paramètre peut être coupé en deux (par dichotomie) lorsque les points correspondants sont trop éloignés.
- La méthode **g:Dparametric3d(p, options)** fait le calcul des points et le dessin de la courbe paramétrée par ⟨*p*⟩. L'argument ⟨*options*⟩ est une table dont les champs sont les options possibles. Celles-ci sont, avec leur valeur par défaut :
  - `t={g:Xinf(),g:Xsup()}`,
  - `nbdots=40`,
  - `discont=false`,
  - `nbdiv=5`,
  - `clip=false`, il indique si la courbe doit être clippée avec la fenêtre 3D courante.
  - `draw_options=""`, chaîne qui sera transmise telle quelle à l'instruction *\draw*.

```
\begin{luadraw}{name=Dparametric3d}
local ld = luadraw
local pt3d = ld.pt3d
local vecI, vecJ, vecK, M, Mc = pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M, pt3d.Mc

local g = ld.graph3d:new{window3d={-4,4,-4,4,-3,3}, window={-7.5,6.5,-7,6}, size={8,8}}
local pi = math.pi
g:Labelsize("footnotesize")
local p = function(t) return Mc(3,t,t/3) end
local L = ld.parametric3d(p,-2*pi,2*pi,25,false,2)
g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
g:Lineoptions("dashed","red",2)
-- projection sur le plan y=-4
g:Dpolyline3d(ld.proj3d(L,{M(0,-4,0),vecJ}))
-- projection sur le plan x=-4
g:Dpolyline3d(ld.proj3d(L,{M(-4,0,0),vecI}))
-- projection sur le plan z=-3
g:Dpolyline3d(ld.proj3d(L,{M(0,0,-3),vecK}))
```

```
-- dessin de la courbe
g:Lineoptions("solid","Navy",8)
g:Dparametric3d(p,{t={-2*pi,2*pi}})
g:Show()
\end{luadraw}
```

FIGURE 7 : Une courbe et ses projections sur trois plans



Paramétrisation d'une ligne polygonale : *curvilinear_param3d*

Soit L une liste de points 3D représentant une ligne continue, il est possible d'obtenir une paramétrisation de cette ligne en fonction d'un paramètre t entre 0 et 1 (t est l'abscisse curviligne divisée par la longueur totale de L).

La fonction **ld.curvilinear_param3d(L [, close])** renvoie une fonction d'une variable $t \in [0;1]$ et à valeurs sur la ligne $\langle L \rangle$ (points 3D), la valeur en $t = 0$ est le premier point de $\langle L \rangle$, et la valeur en $t = 1$ est le dernier point ; cette fonction est suivie d'un nombre qui représente la longueur total de $\langle L \rangle$. L'argument optionnel $\langle close \rangle$ indique si la ligne doit être refermée (`false` par défaut).

Le repère : Dboxaxes3d

La méthode **g:Dboxaxes3d(options)** permet de dessiner les trois axes, avec un certain nombre d'options définies dans la table $\langle options \rangle$. Ces options sont :

- `xaxe=true`, `yaxe=true` et `zaxe=true` : indique si les axes correspondant doivent être dessinés ou non.
- `drawbox=false` : indique si une boite doit être dessinée avec les axes.
- `grid=false` : indique si une grille doit être dessinée (une pour x, une pour y et une pour z). Lorsque cette option vaut `true`, on peut utiliser aussi les options suivantes :
  - `gridwidth=1` : indique l'épaisseur de trait de la grille en dixième de point,
  - `gridcolor="black"` : indique la couleur de la grille,
  - `fillcolor=""` : couleur pour peindre le fond des grilles.
- `xlimits={x1,x2}`, `ylimits={y1,y2}`, `zlimits={z1,z2}` : permet de définir les trois intervalles utilisés pour les axes. Par défaut ce sont les valeurs fournies à l'argument `window3d` à la création du graphe.
- `xgradlimits={x1,x2}`, `ygradlimits={y1,y2}`, `zgradlimits={z1,z2}` : permet de définir les trois intervalles de graduation sur les axes. Par défaut ces options ont la valeur `"auto"`, ce qui veut dire qu'elles prennent les mêmes valeurs que `xlimits`, `ylimits` et `zlimits`.
- `xyzstep=1` : indique le pas des graduations sur les trois axes.
- `xstep=xyzstep`, `ystep=xyzstep`, `zstep=xyzstep` : indique le pas des graduations sur chaque axe (valeur de `xyzstep` par défaut).
- `xlabels={x1,…,xn}`, `ylabels={y1,…,yn}`, `zlabels={z1,…,zn}` : ces options permettent d'imposer les labels sur les axes. Par défaut ces options ont la valeur `nil`, dans ce cas ce sont les labels par défaut (un par graduation) qui sont affichés.

- `xyzticks=0.2` : indique la longueur des graduations.
- `labels=true` : indique si la valeur des graduations doit être affichée ou non.
- `xlabelsep=0.25`, `ylabelsep=0.25`, `zlabelsep=0.25` : indique la distance entre les labels et les graduations.
- `xlabelstyle=<style courant>`, `ylabelstyle=<style courant>`, `zlabelstyle=<style courant>` : indique le style des labels, c'est à dire la position par rapport au point d'ancrage. Par défaut c'est le style en cours qui s'applique.
- `xlegend="$x$"`, `ylegend="$y$"`, `zlegend="$z$"` : permet de définir une légende pour les axes.
- `xlegendsep=0.5`, `ylegendsep=0.5`, `zlegendsep=0.5` : indique la distance entre les legendes et les graduations.

Dessiner sur un plan

Il est possible d'utiliser les méthodes de dessin 2D sur un plan P de l'espace. Il faut pour cela choisir tout d'abord un repère cartésien (A, u, v) du plan P.

- La méthode **g:BeginOnPlane(coordsystem [, options])** permet de passer en mode "dessin sur un plan". L'argument ⟨*coordsystem*⟩ est une table de la forme {A,u,v} représentant un repère cartésien du plan choisi, donc A est un point 3D appartenant au plan, u et v sont deux vecteurs 3D formant une base de la direction du plan. L'argument ⟨*options*⟩ est une table définissant les options, qui sont (avec leur valeur par défaut) :
  - `labeldir=nil`, cette option définit le sens de l'écriture, avec la valeur `nil` c'est le sens courant, avec la valeur `"auto"` le sens de l'écriture est donnée par les vecteurs u et v. On peut également imposer un autre sens de l'écriture ainsi : `labeldir={dir1, dir2}`, où *dir1* et *dir2* sont deux vecteurs 3D indépendants. Cette option ne donne pas toujours de bons résultats en projection centrale, car celle-ci n'est pas une transformation affine.
  - `view=nil`, avec la valeur `nil` la fenêtre sera la même que la fenêtre 2D du graphique. Avec une valeur de la forme `view={x1,x2,y1,y2}`, le dessin sera clippé par la fenêtre $[x_1;x_2]\times[y_1;y_2]$ **dans le repère** (A, u, v).
  - `border=false` et `border_options=""` : avec la valeur `true` un cadre est dessiné autour de la zone définie par la vue, dans ce cas la chaîne `border_options` est transmise à l'instruction *\draw* pour dessiner ce cadre.
  - `tikzmatrix=true`, avec la valeur `true` (valeur par défaut) la méthode utilise l'option *cm* de TikZ (current matrix), avec la valeur `false` c'est la matrice de *luadraw* qui est utilisée laissant intacte celle de TikZ.
  - `lowlevel=false`, avec la valeur `true` l'option `tikzmatrix` prend également la valeur `true`, et la matrice de transformation de TikZ sera appliquée au plus bas niveau avec la macro `\pgflowlevelsynccm`, cela peut être utile par exemple pour que les flèches soient dessinées dans le plan voulu (et non le plan de l'écran). Cependant, cela a une conséquence, tous les éléments graphiques seront transformés : flèches, texte, épaisseur de ligne, etc.
  - `out=nil`, si l'utilisateur donne à cette option une variable de type table (`out=<var>`), alors il pourra récupérer dans cette variable une matrice 2D qui redéfinit les axes pour correspondre à ceux du plan choisi. Cette matrice est utile notamment pour pouvoir utiliser la méthode **g:Dimage()** sur ce plan.

  Une fois cette méthode exécutée, tout nombre complexe $z=\mathrm{Z}(a,b)$ correspondra en réalité au point 3D $\mathrm{A}+a\cdot u+b\cdot v$.

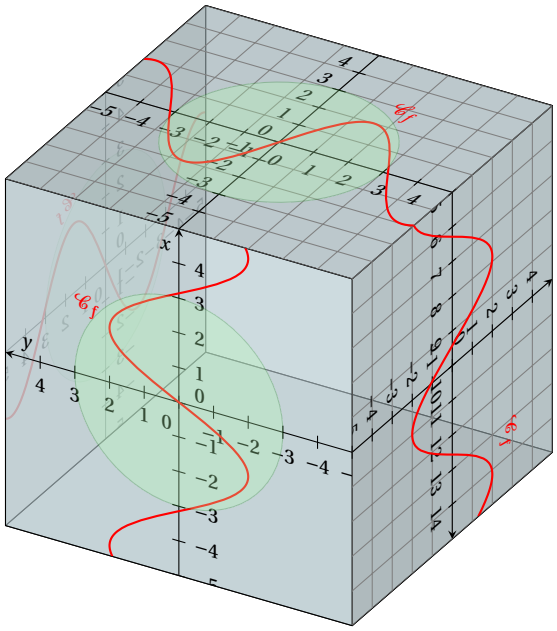
  Attention : cette méthode ne donnera pas le résultat escompté en mode projection centrale, car celle-ci n'est pas une transformation affine.
- La méthode **g:EndOnPlane()**, met fin au mode "dessin sur un plan".

```
\begin{luadraw}{name=draw_on_plane}
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Z, M = cpx.Z, pt3d.M
local Origin, vecI, vecJ, vecK = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK

local g = ld.graph3d:new{ window={-8,8,-9,8}, size={10,10}}
g:Labelsize("footnotesize")
local function f(x) return 2*math.sin(x) end
-- gauche
g:BeginOnPlane( {M(0,-5,0), vecI, vecK}, {labeldir="auto", view={-5,5,-5,5}})
    g:Daxes(nil, {limits={{-5,5},{-5,5}}})
    g:Dcartesian(f, {x={-5,5}, draw_options="red,thick"})
    g:Dcircle(0,3, "ForestGreen, fill=green!30,opacity=0.3")
    g:Dlabel("$\\mathcal{C}_{f}$", Z(2,2),{pos="N",node_options="red"})
g:EndOnPlane()
-- cube
g:Dpoly( g:Box3d(), {color="LightBlue!50!white", opacity=0.6})
```

```
-- dessus
g:BeginOnPlane( {M(0,0,5), vecJ, -vecI}, {labeldir="auto", view={-5,5,-5,5}})
    g:Daxes(nil, {limits={{-5,5},{-5,5}}, grid=true})
    g:Dcartesian(f, {x={-5,5}, draw_options="red,thick"})
    g:Dcircle(0,3, "ForestGreen, fill=green!30,opacity=0.3")
    g:Dlabel("$\\mathcal{C}_{f}$", Z(2,2),{pos="N",node_options="red"})
g:EndOnPlane()
-- devant
g:BeginOnPlane( {M(5,0,0), vecK, -vecJ}, {labeldir={vecJ,vecK}, view={-5,5,-5,5}})
    g:Daxes(nil, {limits={{-5,5},{-5,5}}, labelstyle={"E","S"}, arrows="-stealth",
        legend={"$x$","$y$"}, legendpos={0.9,0.98}})
    g:Dcartesian(f, {x={-5,5}, draw_options="red,thick"})
    g:Dcircle(0,3, "ForestGreen, fill=green!30,opacity=0.3")
    g:Dlabel("$\\mathcal{C}_{f}$", Z(2,2),{pos="W",node_options="red"})
g:EndOnPlane()
-- droite
g:BeginOnPlane( {M(0,5,10), -vecK, -vecI}, {labeldir="auto", view={5,15,-5,5}})
    g:Daxes({10,1,1}, {grid=true, arrows="-stealth", labelshift={0,0}})
    g:Dcartesian(f, {x={5,15}, draw_options="red,thick"})
    g:Dlabel("$\\mathcal{C}_{f}$", Z(14,2),{pos="NW",node_options="red"})
g:EndOnPlane()
g:Show()
\end{luadraw}
```

FIGURE 8 : Dessiner sur un plan



2) Points et labels

Points 3D : Ddots3d, Dballdots3d, Dcrossdots3d

Il y a trois possibilités de dessiner des points 3D. Pour les deux premières, l'argument ⟨*L*⟩ peut être soit un seul point 3D, soit une liste (une table) de points 3D, soit une liste de listes de points 3D :

- La méthode **g:Ddots3d(L [, mark_options, clip])**. Le principe est le même que dans la version 2D, les points sont dessinés dans la couleur courante du tracé de lignes avec le style courant. L'argument ⟨*mark_options*⟩ est une chaîne de caractères facultative qui sera passée telle quelle à l'instruction *\draw* (modifications locales). L'argument ⟨*clip*⟩ vaut `false` par défaut, il indique si le tracé doit être clippé avec la fenêtre 3D courante.
- La méthode **g:Dballdots3d(L [, color, scale, clip])** dessine les points de ⟨*L*⟩ sous forme d'une sphère. L'argument facultatif ⟨*color*⟩ précise la couleur de la sphère (`"black"` par défaut), et l'argument facultatif ⟨*scale*⟩ permet de jouer

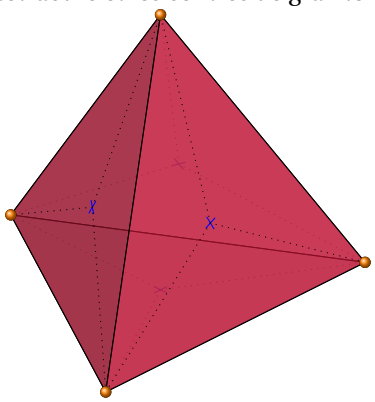
sur la taille de la sphère (1 par défaut).

- La méthode **g:Dcrossdots3d(L [, color, scale, clip])** dessine les points de ⟨*L*⟩ sous forme d'une croix plane. L'argument ⟨*L*⟩ est une liste de la forme {point 3D, vecteur normal} ou { {point3d, vecteur normal}, {point3d, vecteur normal}, ...}. Pour chaque point 3D, le vecteur normal associé permet de déterminer le plan contenant la croix. L'argument facultatif ⟨*color*⟩ précise la couleur de la croix (`"black"` par défaut), et l'argument facultatif ⟨*scale*⟩ permet de jouer sur la taille de la croix (1 par défaut).

```
\begin{luadraw}{name=Ddots3d}
local ld = luadraw
local pt3d = ld.pt3d
local vecI, vecJ, vecK, M = pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{viewdir={15,60},bbox=false,size={8,8}}
local A, B, C, D = 4*M(1,0,-0.5), 4*M(-1/2,math.sqrt(3)/2,-0.5), 4*M(-1/2,-math.sqrt(3)/2,-0.5), 4*M(0,0,1)
local u, v, w = B-A, C-A, D-A
-- centres de gravité faces cachées
for _, F in ipairs({{A,B,C},{B,C,D}}) do
local G, u = pt3d.isobar3d(F), pt3d.prod(F[2]-F[1],F[3]-F[1])
g:Dcrossdots3d({G,u}, "blue",0.75)
g:Dpolyline3d({{F[1],G,F[2]},{G,F[3]}},"dotted")
end
-- dessin du tétraèdre construit sur A, B, C et D
g:Dpoly( ld.tetra(A,u,v,w),{mode=ld.mShaded,opacity=0.7,color="Crimson"})
-- centres de gravité faces visibles
for _, F in ipairs({{A,B,D},{A,C,D}}) do
    local G, u = pt3d.isobar3d(F), pt3d.prod(F[2]-F[1],F[3]-F[1])
    g:Dcrossdots3d({G,u}, "blue",0.75)
    g:Dpolyline3d({{F[1],G,F[2]},{G,F[3]}},"dotted")
end
g:Dballdots3d({A,B,C,D}, "orange") --sommets
g:Show()
\end{luadraw}
```

FIGURE 9 : Un tétraèdre et les centres de gravité de chaque face



Labels 3D : Dlabel3d

La méthode pour placer un label dans l'espace est :
cmdlng:Dlabel3d(text1, anchor1, options1, text2, anchor2, options2, ...).

- Les arguments ⟨*text1*⟩, ⟨*text2*⟩, ..., sont des chaînes de caractères, ce sont les labels.
- Les arguments ⟨*anchor1*⟩, ⟨*anchor2*⟩, ..., sont des points 3D représentant les points d'ancrage des labels.
- Les arguments ⟨*options1*⟩, ⟨*options2*⟩, ..., permettent de définir localement les options des labels, ces options sont (avec leur valeur par défaut) :
  - `pos="center"` : indique la position du label dans le plan de l'écran par rapport au point d'ancrage, il peut valoir `"N"` pour nord, `"NE"` pour nord-est, `"NW"` pour nord-ouest, ou encore `"S"`, `"SE"`, `"SW"`. Par défaut, il vaut `"center"`, et dans ce cas le label est centré sur le point d'ancrage.
  - `dist=0` : c'est la distance (en cm) entre le label et son point d'ancrage lorsque `pos` n'est pas égal a `"center"`.

- `dir={}` : indique la direction de l'écriture dans l'espace, la liste vide indique le sens par défaut. Plus généralement, `dir={dirX,dirY,dep}`, les 3 valeurs *dirX*, *dirY* et *dep* sont trois points 3D représentant 3 vecteurs, les deux premiers indiquent le sens de l'écriture, le troisième un déplacement (translation) du label par rapport au point d'ancrage.
- `node_options=""` : chaîne destinée à recevoir des options qui seront directement passées à TikZ dans l'instruction *\node[]*.
- Les labels sont dessinés dans la couleur courante du texte du document, mais on peut changer de couleur avec l'option `node_options` en mettant par exemple : `node_options="color=blue"`.

  Attention : les options choisies pour un label s'appliquent aussi aux labels suivants si elles sont inchangées.

3) Solides de base (sans facette)

Les quatre méthodes décrites ci-dessous sont sensibles à la variable globale `ld.Hiddenlines` qui vaut `false` par défaut.

Cylindre : Dcylinder

Dessiner un cylindre à base circulaire (droit ou penché). Plusieurs syntaxes possibles :

- Ancienne syntaxe : **g:Dcylinder(A, V, r [, options])** dessine un cylindre droit, où $\langle A \rangle$ est un point 3D représentant le centre d'une des faces circulaires, $\langle V \rangle$ est un point 3D, c'est un vecteur représentant l'axe du cylindre, le centre de la face circulaire opposée est le point A + V (cette face est orthogonale à $\langle V \rangle$), et $\langle r \rangle$ est le rayon de la base circulaire.
- La syntaxe : **g:Dcylinder(A, r, B [, optons])** dessine un cylindre droit, où $\langle A \rangle$ est un point 3D représentant le centre d'une des faces circulaires, $\langle B \rangle$ est le centre de la face opposée, et $\langle r \rangle$ est le rayon. Le cylindre est droit, c'est à dire que les faces circulaires sont orthogonales à l'axe (AB).
- Pour un cylindre penché : **g:Dcylinder(A, r, V, B [, options])**, où $\langle A \rangle$ est un point 3D représentant le centre d'une des faces circulaires, $\langle B \rangle$ est le centre de la face circulaire opposée, $\langle r \rangle$ est le rayon, et $\langle V \rangle$ est un vecteur 3D non nul orthogonal au plan des faces circulaires.

Pour les trois syntaxes, $\langle options \rangle$ est une table dont les champs définissent les options. Ces options sont (avec leur valeur par défaut) :

- `mode=ld.mWireframe` : deux valeurs possibles `ld.mWireframe` ou `ld.mGrid`. En mode `ld.mWireframe` c'est un dessin en fil de fer, en mode `ld.mGrid` c'est un dessin en grille (comme s'il y avait des facettes).
- `hiddenstyle=ld.Hiddenlinestyle` : définit le style de ligne pour les parties cachées (mettre `"noline"` pour ne pas les afficher). Par défaut cette option a la valeur de la variable globale `ld.Hiddenlinestyle` qui est elle même initialisée avec la valeur `"dotted"`.
- `hiddencolor=edgecolor` : définit la couleur des lignes cachées, par défaut c'est la même valeur que l'option `edgecolor`.
- `edgecolor=<couleur courante>`, définit la couleur des lignes.
- `edgestyle=<style courant>`, définit le style de ligne pour les arêtes visibles.
- `edgewidth=<épaisseur courante>`, définit l'épaisseur des des arêtes visible en dixième de point.
- `color=""`, lorsque cette option est une chaîne vide (valeur par défaut) il n'y a pas de remplissage, lorsque c'est une couleur (sous forme de chaîne) il y a un remplissage avec ou sans un gradient linéaire.
- `gradient=true`, booléen indiquant si un remplissage avec gradient doit être utilisé lorsqu'une couleur a été précisée. Avec la valeur `false` il y a un remplissage continu (fill).
- Paramètres pour le gradient : lorsqu'il y a un remplissage avec la couleur et que `gradient=true`, un gradient linéaire est utilisé pour le côté du cylindre et un autre pour la section, dans les deux cas le gradient est défini comme suit :

  `left color=<color>!<l>, right color=<color>!<r>, middlecolor=<color>!<m>`

  où *<color>* désigne la couleur utlisée, et *<l>*, *<m>*, *<r>* sont trois nombres entre 0 et 100, ce sont ces trois nombres qui constituent les paramètres du gradient sous forme d'une table *{<l>,<m>,<r>}* :
  - Pour le côté du cylindre, c'est l'option `gradside`, par défaut : `gradside={50,10,100}`.
  - Pour la section du cylindre, c'est l'option `gradsection`, par défaut : `gradsection={25,18,50}`.
- `opacity=1`, définit la transparence du dessin.

Cône : Dcone

Dessiner un cône à base circulaire (droit ou penché). Plusieurs syntaxes possibles :

- Ancienne syntaxe : **g:Dcone(A, V, r [, options])** dessine un cône droit, où ⟨*A*⟩ est un point 3D représentant le sommet du cône, ⟨*V*⟩ est un point 3D, c'est un vecteur représentant l'axe du cône, le centre de la face circulaire est le point A + V (cette face est orthogonale à ⟨*V*⟩), et ⟨*r*⟩ est le rayon de la base circulaire.
- La syntaxe : **g:Dcone(C, r, A [, options])** dessine un cône droit, où ⟨*A*⟩ est un point 3D représentant le sommet du cône, ⟨*C*⟩ est le centre de la face circulaire, et ⟨*r*⟩ est le rayon. Le cône est droit, c'est à dire que la face circulaire est orthogonale à l'axe (AC).
- Pour un cône penché : **g:Dcone(C, r, V, A [, options])**, où ⟨*A*⟩ est un point 3D représentant le sommet du cône, ⟨*C*⟩ est le centre de la face circulaire, ⟨*r*⟩ est le rayon, et ⟨*V*⟩ est un vecteur 3D non nul orthogonal au plan de la face circulaire.

Pour les trois syntaxes, les ⟨*options*⟩ sont les mêmes que pour le cylindre, plus l'option :

- `apex=true` : avec la valeur `true`, le sommet est matérialisé par un point lorsque le cône est vu de dessus.

Tronc de cône : Dfrustum

Dessiner un tronc de cône à base circulaire (droit ou penché). Deux syntaxes possibles :

- La syntaxe : **g:Dfrustum(A, R, r, B [, options])** pour un tronc de cône droit, ⟨*A*⟩ est un point 3D représentant le centre de la face de rayon ⟨*R*⟩, ⟨*B*⟩ est le centre de la deuxième face circulaire, et son rayon est ⟨*r*⟩, les faces sont orthogonales au vecteur $\vec{AB}$. Lorsque R = *r* on a simplement un cylindre.
- La syntaxe : **g:Dfrustum(A, R, r, V, B [, options])** pour un tronc de cône penché, ⟨*A*⟩ est un point 3D représentant le centre de la face de rayon ⟨*R*⟩, ⟨*V*⟩ est un vecteur 3D représentant un vecteur normal aux faces circulaires, le centre de la deuxième face circulaire est le point ⟨*B*⟩, et son rayon est ⟨*r*⟩. Lorsque R = *r* on a un cylindre penché.

Pour les deux syntaxes, les ⟨*options*⟩ sont les mêmes que pour le cylindre, plus l'option :

- `old=false` : avec la valeur `true`, on peut utiliser l'ancienne syntaxe **g:Dfrustum(A, R, r, V [, options])**, dans ce cas le centre de la seconde face circulaire est A + V.

Sphère : Dsphere

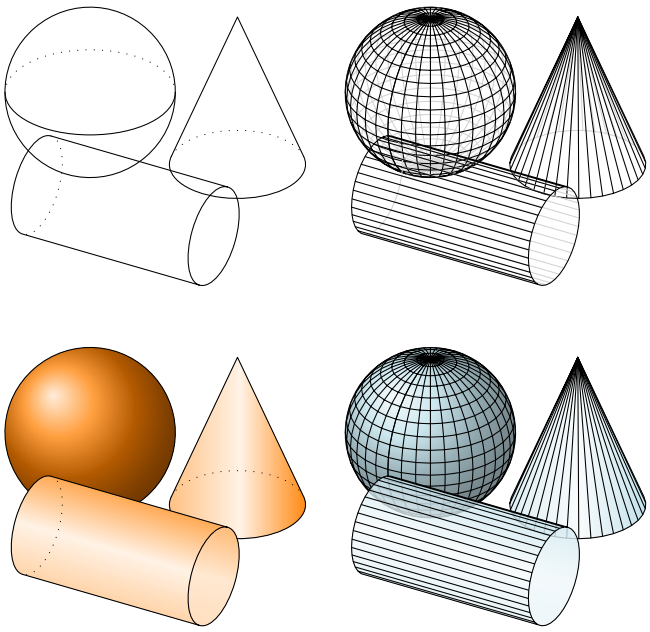
La méthode **g:Dsphere(A, r [, options])** dessine une sphère.

- ⟨*A*⟩ est un point 3D représentant le centre de la sphère.
- ⟨*r*⟩ est le rayon de la sphère
- ⟨*options*⟩ est une table dont les champs définissent les options. Ces options sont (avec leur valeur par défaut) :
  - `mode=ld.mWireframe` : trois valeurs possibles `ld.mWireframe` ou `ld.mGrid` ou `ld.mBorder`. En mode `ld.mWireframe` on dessine le contour (cercle) et l'équateur, en mode `ld.mGrid` c'est le contour avec méridiens et fuseaux (grille), et en mode `ld.mBorder` c'est le contour uniquement.
  - `hiddenstyle=ld.Hiddenlinestyle` : définit le style de ligne pour les parties cachées (mettre `"noline"` pour ne pas les afficher). Par défaut cette option a la valeur de la variable globale `ld.Hiddenlinestyle` qui est elle même initialisée avec la valeur `"dotted"`.
  - `hiddencolor=edgecolor` : définit la couleur des lignes cachées, par défaut c'est la même valeur que l'option `edgecolor`.
  - `edgecolor=<couleur courante>`, définit la couleur des lignes.
  - `edgestyle=<style courant>`, définit le style de ligne pour les arêtes visibles.
  - `edgewidth=<épaisseur courante>`, définit l'épaisseur des des arêtes visible en dixième de point.
  - `color=""` : lorsque cette option est une chaîne vide (valeur par défaut) il n'y a pas de remplissage, lorsque c'est une couleur (sous forme de chaîne) il y a un remplissage avec "ball color".
  - `opacity=1`, définit la transparence du dessin.

```
\begin{luadraw}{name=cylindre_cone_sphere}
local ld = luadraw
local pt3d = ld.pt3d
local vecI, vecJ, vecK, M = pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M
local g = ld.graph3d:new{ size={10,10} }
ld.Hiddenlines = true
local dessin = function(args)
```

```
    g:Dsphere(M(-1,-2.5,1),2.5, args)
    g:Dcone(M(-1,2.5,5),-5*vecK,2, args)
    g:Dcylinder(M(3,-2,0),6*vecJ,1.5, args)
end
-- en haut à gauche, options par défaut
g:Saveattr(); g:Viewport(-5,0,0,5); g:Coordsystem(-5,5,-5,5,true); dessin(); g:Restoreattr()
g:Saveattr(); g:Viewport(0,5,0,5); g:Coordsystem(-5,5,-5,5,true) -- en haut à droite
dessin({mode=ld.mGrid, hiddenstyle="solid", hiddencolor="LightGray"}); g:Restoreattr()
g:Saveattr(); g:Viewport(-5,0,-5,0); g:Coordsystem(-5,5,-5,5,true) -- en bas à gauche
dessin({mode=ld.mBorder, color="orange"}); g:Restoreattr()
g:Saveattr(); g:Viewport(0,5,-5,0); g:Coordsystem(-5,5,-5,5,true) -- en bas à droite
dessin({mode=ld.mGrid,opacity=0.8,hiddenstyle="noline",color="LightBlue"}); g:Restoreattr()
g:Show()
\end{luadraw}
```

FIGURE 10 : Cylindres, cônes et sphères



Sphère coupée : Dcut_sphere

La méthode **g:Dcut_sphere(A, r, P, [, options])** dessine la partie de la sphère de centre $\langle A\rangle$ et de rayon $\langle e\rangle$, coupée par le plan $\langle P\rangle$= {B, n} (plan passant par le point 3D b et de vecteur normal n). C'est la partie de la sphère qui est dans le demi-plan contenant le vecteur n qui est dessiné, seule la partie extérieure de la sphère est dessinée, pas l'intérieur.

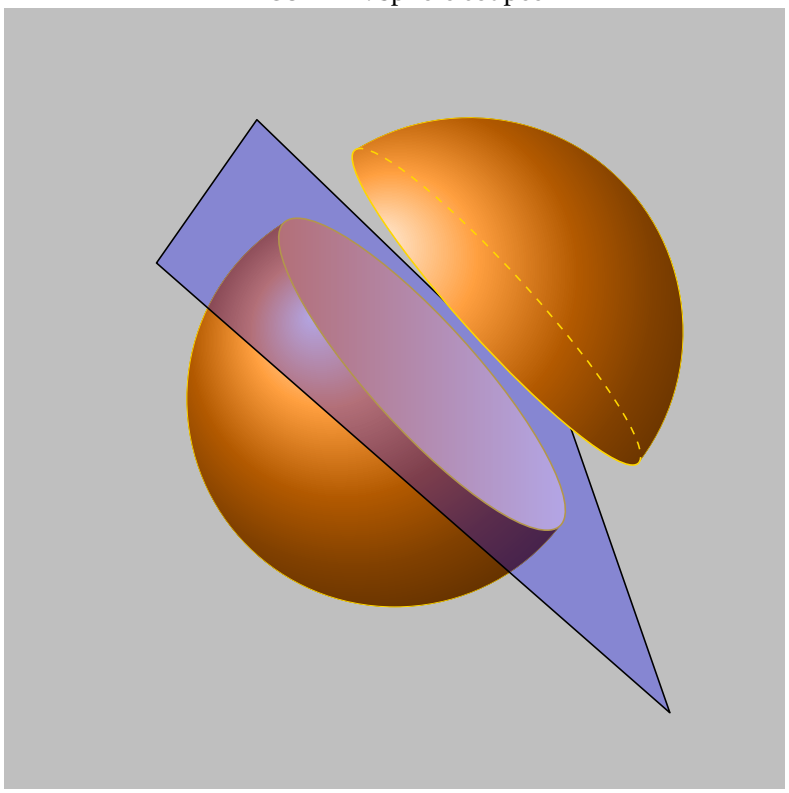
Les $\langle$*options*$\rangle$ sont les mêmes que pour la méthode **g:Dsphere()**, plus les options suivantes :

- `visibletrace=""`, chaîne de caractères définissant les options (passées à la commande *\draw*) pour le dessin de la partie sur la face visible (si elle existe) du cercle intersection entre le plan et la sphère. Lorsque cette chaîne est vide (valeur par défaut) cet arc de cercle n'est pas dessiné.
- `hiddentrace=""`, chaîne de caractères définissant les options (passées à la commande *\draw*) pour le dessin de la partie sur la face non visible (si elle existe) du cercle intersection entre le plan et la sphère. Lorsque cette chaîne est vide (valeur par défaut) cet arc de cercle n'est pas dessiné.
- `outline=false`, avec la valeur `true`, le contour de la partie sphérique est dessinée avec les options contenues dans la chaîne `visibletrace` (ou les options courantes si la chaîne est vide).
- `section=""`, chaîne de caractères définissant les options (passées à la commande *\draw*) pour dessiner la section circulaire lorsque celle-ci est visible. Lorsque cette chaîne est vide (valeur par défaut) la section n'est pas dessinée.

```
\begin{luadraw}{name=cut_sphere}
local ld = luadraw
local pt3d = ld.pt3d
```

```
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M
local g = ld.graph3d:new{ viewdir={"central",0,65}, size={10,10}, bg="lightgray"}
g:Linewidth(6)
local O, P = Origin, {vecK, -vecK-vecJ}
local c, r, n = ld.interPS(P, {O, 3})
g:Dcut_sphere(O, 3, P, {color="orange", mode=ld.mBorder, edgecolor="Gold", section=true,
    section_options="Gold,left color=orange!80, right color=orange!10"})
g:Dplane({c,P[2]}, vecI,8,8,"fill=blue, fill opacity=0.3")
g:Shift3d(-P[2])
g:Dcut_sphere(O, 3, {P[1],-P[2]},
    {color="orange", mode=ld.mBorder, edgecolor="Gold", visibletrace="Gold,line width=0.6",
     hiddentrace="dashed,Gold,line width=0.6"})
g:Show()
\end{luadraw}
```

FIGURE 11 : Sphère coupée



Cone_outline, Cylinder_outline, Frustum_outline, Sphere_outline

Les méthodes suivantes calculent et renvoient une table contenant les éléments caractéristiques relatifs au cône, au cylindre, au cône tronqué ou à la sphère. Ces méthodes prennent en compte le mode de projection ainsi que la matrice de transformation 3D du graphe. Les trois premières fonctions renvoient une table donc les champs sont :

- `side` : chemin 3D représentant le contour de la partie latérale visible du solide.
- `hiddenside` : chemin 3D représentant le contour de la partie latérale non visible du solide.
- `section` : liste de chemins 3D, chacun de ces chemins représente le contour d'une des bases visibles du solide.
- `hiddensection` : liste de chemins 3D, chacun de ces chemins représente le contour d'une des bases non visibles du solide.
- `visible` : chemin 3D représentant le contour visible du solide.
- `hidden` : chemin 3D représentant le contour caché du solide.
- `tangency` : liste de points 3D représentant les points de tangence du solide, ce sont des points pour lesquels le plan tangent au solide contient la direction de l'observateur (vecteur **g.Normal**), ils sont sur les bases circulaires.
- `angle` : angle en degré que fait l'axe du solide par rapport à l'horizontal.

Description des quatre méthodes

- La méthode **g:Cone_outline(C, r, [V,] A)** calcule et renvoie les éléments caractéristiques du cône de sommet $\langle A \rangle$, et dont la base circulaire a pour centre $\langle C \rangle$ et rayon $\langle r \rangle$. Le cône est droit si $\langle V \rangle$ est omis, sinon la base est orthogonale au vecteur $\langle V \rangle$ (cône penché).
- La méthode **g:Cylinder_outline(C, r, [V,] A)** calcule et renvoie les éléments caractéristiques du cylindre de rayon $\langle r \rangle$, et dont les bases circulaires ont pour centre $\langle C \rangle$ et $\langle A \rangle$. Le cylindre est droit si $\langle V \rangle$ est omis, sinon les bases circulaires sont orthogonales au vecteur $\langle V \rangle$ (cylindre penché).
- La méthode **g:Frustum_outline(C, R, r, [V,] A)** calcule et les éléments caractéristiques du cône tronqué dont les bases circulaires ont pour centre $\langle C \rangle$ et $\langle A \rangle$, et pour rayon $\langle R \rangle$ (pour la première) et $\langle r \rangle$. Le cône tronqué est droit si $\langle V \rangle$ est omis, sinon les bases circulaires sont orthogonales au vecteur $\langle V \rangle$ (cône tronqué penché).
- La méthode **g:Sphere_outline(C, R)** calcule et les éléments caractéristiques de la sphère, sous la forme d'une table dont les champs sont :
  - `visible` : chemin 3D représentant le contour apparent de la sphère.
  - `data` : liste de la forme $\{O, r, n\}$ où O est le centre du cercle apparent, r son rayon et n un vecteur normal au plan du cercle apparent.

Cone_tangency, Cylinder_tangency, Frustum_tangency, Sphere_tangency

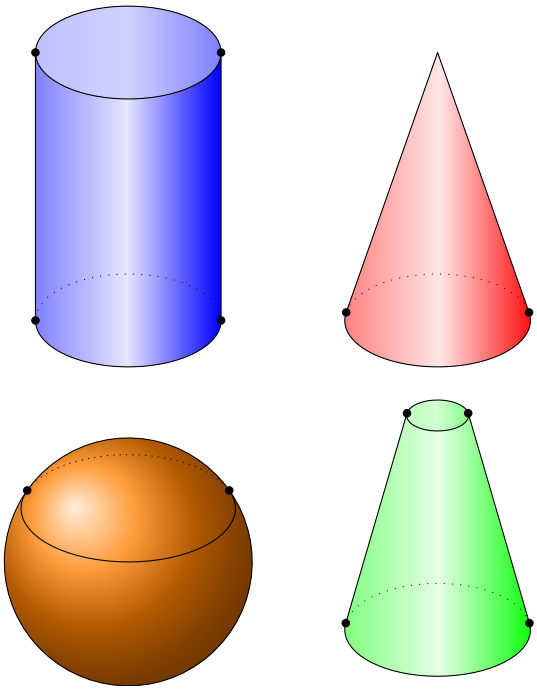
Les méthodes suivantes calculent et renvoient sous forme d'une liste, les points de tangence, lorsqu'ils existent, au cône, au cylindre, au cône tronqué ou à la sphère.

- La méthode **g:Cone_tangency(C, r, [V,] A)** calcule et renvoie les deux points de tangence au cône de sommet $\langle A \rangle$, et dont la base circulaire a pour centre $\langle C \rangle$ et rayon $\langle r \rangle$. Le cône est droit si $\langle V \rangle$ est omis, sinon la base est orthogonale au vecteur $\langle V \rangle$ (cône penché). Les deux points de tangence sont sur la base circulaire, ce sont des points pour lesquels le plan tangent au cône contient la direction de l'observateur (vecteur **g.Normal**).
- La méthode **g:Cylinder_tangency(C, r, [V,] A)** calcule et renvoie les quatre points de tangence au cylindre de rayon $\langle r \rangle$, et dont les bases circulaires ont pour centre $\langle C \rangle$ et $\langle A \rangle$. Le cylindre est droit si $\langle V \rangle$ est omis, sinon les bases circulaires sont orthogonales au vecteur $\langle V \rangle$ (cylindre penché). Les deux premiers points de tangence sont sur la base circulaire de centre $\langle C \rangle$, et les deux suivants sur l'autre base circulaire.
- La méthode **g:Frustum_tangency(C, R, r, [V,] A)** calcule et renvoie les quatre points de tangence au cône tronqué dont les bases circulaires ont pour centre $\langle C \rangle$ et $\langle A \rangle$, et pour rayon $\langle R \rangle$ (pour la première) et $\langle r \rangle$. Le cône tronqué est droit si $\langle V \rangle$ est omis, sinon les bases circulaires sont orthogonales au vecteur $\langle V \rangle$ (cône tronqué penché). Les deux premiers points de tangence sont sur la base circulaire de centre $\langle C \rangle$, et les deux suivants sur l'autre base circulaire.
- La méthode **g:Sphere_tangency(C, R, P)** calcule et renvoie les deux points de tangence à la sphère de centre $\langle C \rangle$ et de rayon $\langle R \rangle$, et au cercle correspondant à l'intersection de cette sphère avec le plan $\langle P \rangle$ (qui est une liste de la forme $\{A, n\}$ où A est un point du plan, et n un vecteur normal au plan).

```
\begin{luadraw}{name=tangency}
local ld = luadraw
local pt3d, cpx = ld.pt3d, ld.cpx
local M, Z = pt3d.M, cpx.Z
local O, I, J, K = M(0,0,0), M(1,0,0), M(0,1,0), M(0,0,1)
local g = ld.graph3d:new{ size={10,10}, window={-5,5,-5,6} }
ld.Hiddenlines = true
g:Shift( Z(-2.5,3.5) ) -- cylinder
local A, B, C, D = table.unpack( g:Cylinder_tangency(-3*K, 1.5, 2*K) )
g:Dcylinder(-3*K, 1.5, 2*K, {color="blue"})
g:Ddots3d({A,B,C,D})
g:Shift( Z(5,0) ) -- cone
local A, B = table.unpack( g:Cone_tangency(-3*K, 1.5, 2*K) )
g:Dcone(-3*K, 1.5, 2*K, {color="red"})
g:Ddots3d({A,B})
g:Shift( Z(0,-5) ) -- frustum
local A, B, C, D = table.unpack( g:Frustum_tangency(-3*K, 1.5,0.5, K) )
g:Dfrustum(-3*K, 1.5, 0.5, K, {color="green"})
g:Ddots3d({A,B,C,D})
g:Shift(Z(-5,-1.5)) -- sphere
local R, P = 2, {K, K}
local I, r, n = ld.interPS(P, {O,R})
```

```
local A, B = table.unpack( g:Sphere_tangency(O, R, P) )
g:Dsphere(O, R, {color="orange", mode=ld.mBorder})
g:Ddots3d({A,B})
g:Darc3d(A,I,B,r,1,n); g:Darc3d(A,I,B,r,-1,n,"dotted")
g:Show()
\end{luadraw}
```

FIGURE 12 : Points de tangence



IV Solides à facettes

1) Définition d'un solide

Il y a deux façons de définir un solide :

1. Sous forme d'une liste (table) de facettes. Une facette est elle-même une liste de points 3D (au moins 3) coplanaires et non alignés, qui sont les sommets. Les facettes sont supposées convexes et elles sont orientées par l'ordre d'apparition des sommets. C'est à dire, si A, B et C sont les trois premiers sommets d'une facette F, alors la facette est orientée avec le vecteur normal $\vec{AB} \wedge \vec{AC}$. Si ce vecteur normal est dirigé vers l'observateur, alors la facette est considérée comme visible. La méthode **g:Cosine_incidence(N [, A])** renvoie le cosinus de l'angle au point $\langle A \rangle$ entre le vecteur $\langle N \rangle$ (qui doit être unitaire) et le vecteur unitaire dirigé vers l'observateur, si ce cosinus est positif alors $\langle N \rangle$ est dirigé vers l'observateur (le point $\langle A \rangle$ est facultatif SAUF en projection centrale).

   Dans la définition d'un solide, les vecteurs normaux aux facettes doivent être dirigés vers **l'extérieur** du solide pour que l'orientation soit correcte.

2. Sous forme de **polyèdre**, c'est à dire une table à deux champs, un premier champ appelé *vertices* qui est la liste des sommets du polyèdre (points 3D), et un deuxième champ appelé *facets* qui la liste des facettes, mais ici, dans la définition des facettes, les sommets sont remplacés par leur indice dans la liste *vertices*. Les facettes sont orientées de la même façon que précédemment. Cette définition correspond au format *obj* mais sans vecteurs normaux.

Par exemple, considérons les quatre points A = M(−2,−2,0), B = M(3,0,0), C = M(−2,2,0) et D = M(0,0,4), alors on peut définir le tétraèdre construit sur ces quatre points :

- soit sous forme d'une liste de facettes : `T={{A,B,D},{B,C,D},{C,A,D},{A,C,B}}` (attention à l'orientation),
- soit sous forme de polyèdre : `T={vertices={A,B,C,D},facets={{1,2,4},{2,3,4},{3,1,4},{1,3,2}}}`.

Fonctions de conversion entre les deux définitions

- La fonction **ld.poly2facet(P)** où ⟨*P*⟩ est un polyèdre, renvoie ce solide sous forme d'une liste de facettes.
- La fonction **ld.facet2poly(L [, epsilon])** renvoie la liste de facettes ⟨*L*⟩ sous forme de polyèdre. L'argument facultatif ⟨*epsilon*⟩ vaut 10^{-8} par défaut, il précise à combien près sont faites les comparaisons entre points 3D. S'il y a beaucoup de facettes, le temps de calcul peut devenir important.

2) Dessin d'un polyèdre : Dpoly

La fonction **g:Dpoly(P, options)** permet de représenter le polyèdre ⟨*P*⟩ (par l'algorithme naïf du peintre). L'argument ⟨*options*⟩ est une table contenant les options, celles-ci sont (avec leur valeur par défaut) :

- `mode=ld.mShaded` : définit le mode de représentation. Il y a six valeurs possibles :
  - `ld.mWireframe` : mode fil de fer, on dessine les arêtes visibles et cachées.
  - `ld.mFlat` : on dessine les faces de couleur unie, ainsi que les arêtes visibles.
  - `ld.mFlatHidden` : on dessine les faces de couleur unie, les arêtes visibles, et les arêtes cachées.
  - `ld.mShaded` : on dessine les faces de couleur nuancée en fonction de leur inclinaison, ainsi que les arêtes visibles. C'est le mode par défaut.
  - `ld.mShadedHidden` : on dessine les faces de couleur nuancée en fonction de leur inclinaison, les arêtes visibles et cachées.
  - `ld.mShadedOnly` : on dessine les faces de couleur nuancée en fonction de leur inclinaison, mais pas les arêtes.
- `contrast=1` : nombre qui permet d'accentuer ou diminuer la nuance des couleurs des facettes dans les modes `ld.mShaded`, `ld.mShadedHidden`, `ld.mShadedOnly`.
- `edgestyle=<style courant>` : chaîne qui définit le style de ligne des arêtes.
- `edgecolor=<couleur courante>` : chaîne qui définit la couleur des arêtes.
- `hiddenstyle=ld.Hiddenlinestyle` : chaîne qui définit le style de ligne des arêtes cachées. Par défaut c'est la valeur contenue dans la variable globale `ld.Hiddenlinestyle` (qui vaut elle-même `"dotted"` par défaut).
- `hiddencolor=<couleur courante>` : chaîne qui définit la couleur des arêtes cachées.
- `edgewidth=<épaisseur courante>` : épaisseur de trait des arêtes en dixième de point.
- `opacity=1` : nombre entre 0 et 1 qui permet de mettre une transparence ou non sur les facettes. La valeur par défaut est 1, ce qui signifie pas de transparence.
- `backcull=false` : avec la valeur `true`, les facettes considérées comme non visibles (vecteur normal non dirigé vers l'observateur) ne sont pas affichées. Cette option est intéressante pour les polyèdres convexes car elle permet de diminuer le nombre de facettes à dessiner.
- `twoside=true` : avec la valeur `true` on distingue les deux côtés des facettes (intérieur et extérieur), les deux côtés n'auront pas exactement la même couleur.
- `reverse=false` : avec la valeur `true` l'orientation des facettes est inversée.
- `color="white"` : chaîne définissant la couleur de remplissage des facettes.
- `usepalette=nil` : cette option permet éventuellement de préciser une palette de couleurs pour peindre les facettes ainsi qu'un mode de calcul, la syntaxe est : `usepalette={palette,mode [, minmax]}`, où ⟨*palette*⟩ désigne une table de couleurs qui sont elles-mêmes des tables de la forme $\{r, g, b\}$ où r, g et b sont des nombres entre 0 et 1. L'argument ⟨*mode*⟩ peut être :
  - soit une des chaînes : `"x"`, `"y"`, `"z"`. Dans le premier cas par exemple, les facettes au centre de gravité d'abscisse minimale ont la première couleur de la palette, les facettes au centre de gravité d'abscisse maximale ont la dernière couleur de la palette, pour les autres, la couleur est calculée en fonction de l'abscisse du centre de gravité par interpolation linéaire.
  - soit une fonction : ⟨*mode*⟩ : $f \mapsto \text{mode}(f) \in \mathbb{R}$, où f désigne une facette (liste de points 3D). Les facettes ayant la valeur minimale ont la première couleur de la palette, celles ayant la valeur maximale ont la dernière couleur de la palette, pur les autres la couleur est calculée par interpolation linéaire.

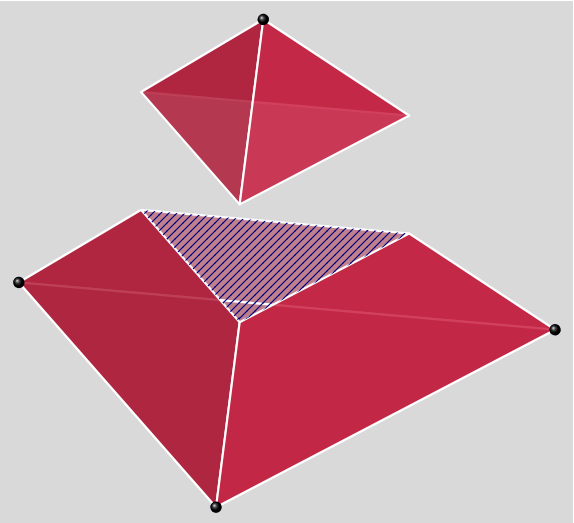
  L'argument facultatif ⟨*minmax*⟩ est une table de deux nombres, ces deux nombres sont utilisés comme valeur minimale et valeur maximale de l'argument ⟨*mode*⟩. En l'absence de ⟨*minmax*⟩, ces deux valeurs sont calculées automatiquement.

```
\begin{luadraw}{name=tetra_coupe}
local ld = luadraw
local pt3d = ld.pt3d
```

```
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{viewdir={10,60},bbox=false, size={10,10}, bg="gray!30"}
local A,B,C,D = M(-2,-4,-2),M(4,0,-2),M(-2,4,-2),M(0,0,2)
local T = ld.tetra(A,B-A,C-A,D-A) -- tétraèdre de sommets A, B, C, D
local plan = {Origin, -vecK} -- plan de coupe
local T1, T2, section = ld.cutpoly(T,plan) -- on coupe le tétraèdre
-- T1 est le polyèdre résultant dans le demi espace contenant -vecK
-- T2 est le polyèdre résultant dans l'autre demi espace
-- section est une facette (c'est la coupe)
g:Dpoly(T1,{color="Crimson", edgecolor="white", opacity=0.8, edgewidth=8})
g:Filloptions("bdiag","Navy"); g:Dpolyline3d(section,true,"draw=none")
g:Dpoly(ld.shift3d(T2,2*vecK), {color="Crimson", edgecolor="white", opacity=0.8, edgewidth=8})
g:Dballdots3d({A,B,C,D+2*vecK}) -- on a dessiné T2 translaté avec le vecteur 2*vecK
g:Show()
\end{luadraw}
```

FIGURE 13 : Section d'un tétraèdre par un plan



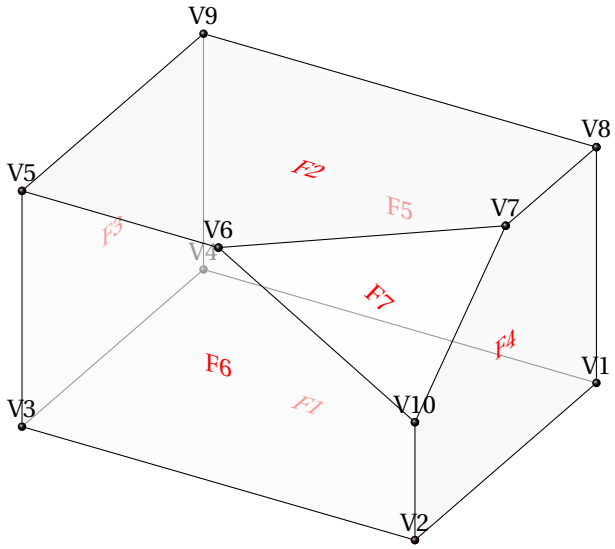
3) Visualiser les numéros des facettes et/ou ceux des sommets d'un polyèdre

La méthode **g:Dpolynames(P [, option, opacity])** affiche le polyèdre ⟨*P*⟩ avec la possibilité d'ajouter sur chaque face son numéro (qui est son rang d'apparition dans la liste *P.facets*) précédé de la lettre *F*, et éventuellement le numéro de chaque sommet (qui est son rang d'apparition dans la liste *P.vertices*) précédé de la lettre *V*. L'argument ⟨*option*⟩ peut prendre les valeurs : `"facet"` ou bien `"vertex"` ou bien `"both"` (qui est la valeur par défaut). L'argument ⟨*opacity*⟩ est un nombre entre 0 et 1 qui vaut 0.6 par défaut.

```
\begin{luadraw}{name=show_facet_number}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{viewdir={30,60}, window={-2.5,5,-3,3},size={10,10}}
P = ld.parallelep(Origin, 4*vecI,5*vecJ,3*vecK)
local A, B, C = M(4,2.5,3), M(2,5,3), M(4,5,1.5)
P = ld.cutpoly(P, ld.plane(A,B,C), true) -- P est coupé avec le plan, la section est ajoutée en tant que facette
g:Dpolynames(P) -- on visualise les numéros des facettes et des sommets de P
g:Show()
\end{luadraw}
```

FIGURE 14 : Visualiser les faces et sommets d'un polyèdre



4) Fonctions de construction de polyèdres

Les fonctions suivantes renvoient un polyèdre, c'est à dire une table à deux champs, un premier champ appelé *vertices* qui est la liste des sommets du polyèdre (points 3D), et un deuxième champ appelé *facets* qui la liste des facettes, mais dans la définition des facettes, les sommets sont remplacés par leur indice dans la liste *vertices*.

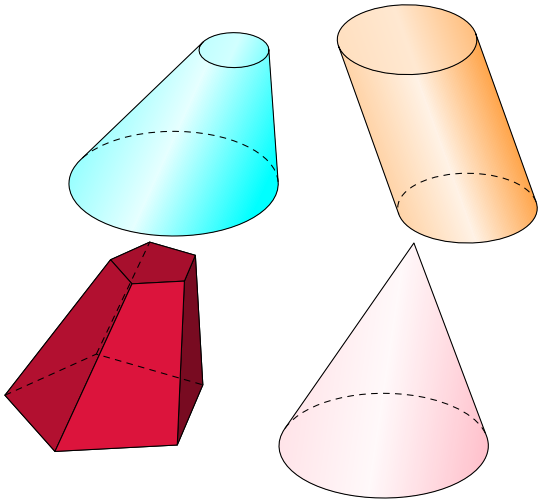
- **ld.tetra(S, v1, v2, v3)** renvoie le tétraèdre construit à partir du sommet ⟨*S*⟩ (point 3D) et des 3 vecteurs ⟨*v1*⟩, ⟨*v2*⟩, ⟨*v3*⟩ (points 3D) supposés dans le sens direct. Les sommets de ce tétraèdre sont S, S + v_1, S + v_2, S + v_3.
- **ld.parallelep(A, v1, v2, v3)** renvoie le parallélépipède construit à partir du sommet ⟨*A*⟩ (point 3D) et de 3 vecteurs ⟨*v1*⟩, ⟨*v2*⟩, ⟨*v3*⟩ (points 3D) supposés dans le sens direct.
- **ld.prism(base, vector [, open])** renvoie un prisme, l'argument ⟨*base*⟩ est une liste de points 3D (une des deux bases du prisme), ⟨*vector*⟩ est le vecteur de translation (point 3D) qui permet d'obtenir la seconde base. L'argument facultatif ⟨*open*⟩ est un booléen indiquant si le prisme est ouvert ou non (`false` par défaut). Dans le cas où il est ouvert, seules les facettes latérales sont renvoyées. La ⟨*base*⟩ doit être orientée par le ⟨*vector*⟩.
- **ld.pyramid(base, vertex [, open])** renvoie une pyramide, l'argument ⟨*base*⟩ est une liste de points 3D, ⟨*vertex*⟩ est le sommet de la pyramide (point 3D). L'argument facultatif ⟨*open*⟩ est un booléen indiquant si la pyramide est ouverte ou non (`false` par défaut). Dans le cas où elle est ouverte, seules les facettes latérales sont renvoyées. La ⟨*base*⟩ doit être orientée par le sommet.
- **ld.regular_pyramid(n, side, height [, open, center, axe])** renvoie une pyramide régulière, ⟨*n*⟩ est le nombre de côtés de la base, l'argument ⟨*side*⟩ est la longueur d'un côté, et ⟨*height*⟩ est la hauteur de la pyramide. L'argument facultatif ⟨*open*⟩ est un booléen indiquant si la pyramide est ouverte ou non (`false` par défaut). Dans le cas où elle est ouverte, seules les facettes latérales sont renvoyées. L'argument facultatif ⟨*center*⟩ est le centre de la base (`Origin` par défaut), et l'argument facultatif ⟨*axe*⟩ est un vecteur directeur de l'axe de la pyramide (`vecK` par défaut).
- **ld.truncated_pyramid(base, vertex, height [, open])** renvoie une pyramide tronquée, l'argument ⟨*base*⟩ est une liste de points 3D, ⟨*vertex*⟩ est le sommet de la pyramide (point 3D). L'argument ⟨*height*⟩ est un nombre indiquant la hauteur par rapport à la base, où s'effectue la troncature, celle-ci est parallèle au plan de la base. L'argument facultatif ⟨*open*⟩ est un booléen indiquant si la pyramide est ouverte ou non (`false` par défaut). Dans le cas où elle est ouverte, seules les facettes latérales sont renvoyées. La ⟨*base*⟩ doit être orientée par le sommet.
- **ld.cylinder(A, V, R [, nbfacet, open])** renvoie un cylindre droit de rayon ⟨*R*⟩, ⟨*A*⟩ (point 3D) est le centre d'une des bases circulaires et ⟨*V*⟩ est vecteur 3D non nul tel que le centre de la seconde base est le point A + V. L'argument facultatif ⟨*nbfacet*⟩ vaut 35 par défaut (nombre de facettes latérales). L'argument facultatif ⟨*open*⟩ est un booléen indiquant si le cylindre est ouvert ou non (`false` par défaut). Dans le cas où il est ouvert, seules les facettes latérales sont renvoyées.
- **ld.cylinder(A, R, B [, nbfacet, open])** renvoie un cylindre droit de rayon ⟨*R*⟩, ⟨*A*⟩ (point 3D) est le centre d'une des bases circulaires et ⟨*B*⟩ est le centre de la seconde base. L'argument facultatif ⟨*nbfacet*⟩ vaut 35 par défaut (nombre

de facettes latérales). L'argument facultatif ⟨*open*⟩ est un booléen indiquant si le cylindre est ouvert ou non (`false` par défaut). Dans le cas où il est ouvert, seules les facettes latérales sont renvoyées.

- **ld.cylinder(A, R, V, B [, nbfacet, open])** renvoie un cylindre de rayon ⟨*R*⟩, ⟨*A*⟩ (point 3D) est le centre d'une des bases circulaires et ⟨*B*⟩ est le centre de la seconde base, et ⟨*V*⟩ est un vecteur 3D normal au plan des bases circulaires (le cylindre peut donc être penché). L'argument facultatif ⟨*nbfacet*⟩ vaut 35 par défaut (nombre de facettes latérales). L'argument facultatif ⟨*open*⟩ est un booléen indiquant si le cylindre est ouvert ou non (`false` par défaut). Dans le cas où il est ouvert, seules les facettes latérales sont renvoyées.
- **ld.cone(A, V, R [, nbfacet, open])** renvoie un cône de sommet ⟨*A*⟩ (point 3D), d'axe dirigé par ⟨*V*⟩ (point 3D), de base le cercle de centre le point A + V de rayon ⟨*R*⟩ (dans un plan orthogonal à ⟨*V*⟩). L'argument facultatif ⟨*nbfacet*⟩ vaut 35 par défaut (nombre de facettes latérales). L'argument facultatif ⟨*open*⟩ est un booléen indiquant si le cône est ouvert ou non (`false` par défaut). Dans le cas où il est ouvert, seules les facettes latérales sont renvoyées.
- **ld.cone(C, R, A [, nbfacet, open])** renvoie un cône de sommet ⟨*A*⟩ (point 3D), ⟨*C*⟩ est le centre de base circulaire et ⟨*R*⟩ son rayon (dans un plan orthogonal à l'axe (AC)). L'argument facultatif ⟨*nbfacet*⟩ vaut 35 par défaut (nombre de facettes latérales). L'argument facultatif ⟨*open*⟩ est un booléen indiquant si le cône est ouvert ou non (`false` par défaut). Dans le cas où il est ouvert, seules les facettes latérales sont renvoyées.
- **ld.cone(C, R, V, A [, nbfacet, open])** renvoie un cône de sommet ⟨*A*⟩ (point 3D), ⟨*C*⟩ est le centre de base circulaire, ⟨*R*⟩ son rayon, la base est dans un plan orthogonal à ⟨*V*⟩ (vecteur 3D). L'axe (AC) n'est donc pas forcément orthogonal à la face circulaire (cône penché). L'argument facultatif ⟨*nbfacet*⟩ vaut 35 par défaut (nombre de facettes latérales). L'argument facultatif ⟨*open*⟩ est un booléen indiquant si le cône est ouvert ou non (`false` par défaut). Dans le cas où il est ouvert, seules les facettes latérales sont renvoyées.
- **ld.frustum(C, R, r, A [, nbfacet, open])** renvoie un tronc de cône droit. Le point ⟨*C*⟩ (point 3D) est le centre de la base circulaire de rayon ⟨*R*⟩, le centre de l'autre base circulaire est le point ⟨*A*⟩, et son rayon est ⟨*r*⟩, les bases sont orthogonales au vecteur $\vec{CA}$. L'argument facultatif ⟨*nbfacet*⟩ vaut 35 par défaut (nombre de facettes latérales). L'argument facultatif ⟨*open*⟩ est un booléen indiquant si le tronc de cône est ouvert ou non (`false` par défaut). Dans le cas où il est ouvert, seules les facettes latérales sont renvoyées.
- **ld.frustum(C, R, r, V, A [, nbfacet, open])** renvoie un tronc de cône. Le point ⟨*C*⟩ (point 3D) est le centre de la base circulaire de rayon ⟨*R*⟩, le centre de l'autre base circulaire est le point ⟨*A*⟩, et son rayon est ⟨*r*⟩, les bases sont orthogonales au vecteur ⟨*V*⟩, mais pas forcément orthogonales à l'axe (AC). L'argument facultatif ⟨*nbfacet*⟩ vaut 35 par défaut (nombre de facettes latérales). L'argument facultatif
  arguCopen est un booléen indiquant si le tronc de cône est ouvert ou non (`false` par défaut). Dans le cas où il est ouvert, seules les facettes latérales sont renvoyées.
- **ld.sphere(A, R [, nbu, nbv])** renvoie la sphère de centre ⟨*A*⟩ (point 3D) et de rayon ⟨*R*⟩. L'argument facultatif ⟨*nbu*⟩ représente le nombre de fuseaux (36 par défaut) et l'argument facultatif ⟨*nbv*⟩ le nombre de parallèles (20 par défaut).

```
\begin{luadraw}{name=frustum_pyramid}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M
local g = ld.graph3d:new{adjust2d=true,bbox=false, size={10,10} }
ld.Hiddenlines = true; ld.Hiddenlinestyle = "dashed"
g:Dfrustum(M(-1,-4,0),3,1,vecK, M(-1,-2,5), {color="cyan"})
g:Dcylinder(M(-4,4,0),2,vecK,M(-4,2,5), {color="orange"})
local base = ld.map(pt3d.toPoint3d, ld.polyreg(0,3,5))
g:Dpoly(ld.truncated_pyramid( ld.shift3d(base,8*vecI-vecJ-2*vecK), M(5,0,5),4),
    {mode=ld.mShadedHidden,color="Crimson"})
g:Dcone(M(6,7,-2),3,vecK,M(6,8,5),{color="Pink"})
g:Show()
\end{luadraw}
```

FIGURE 15 : Cône tronqué, pyramide tronquée, cylindre oblique

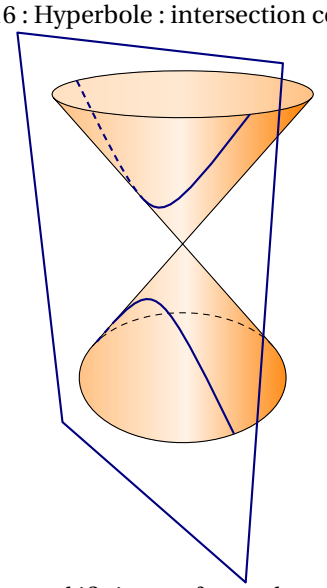


Remarque : nous avons déjà des primitives pour dessiner des cylindres, cônes, et sphères sans passer par des facettes. Un des intérêts de donner une définition de ces objets sous forme de polyèdres est que l'on va pouvoir faire certains calculs sur ces objets comme par exemple des sections planes.

```
\begin{luadraw}{name=hyperbole}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window={-8,6,-9,9},bbox=false,
    viewdir={"central",45,65}, size={10,10}}
ld.Hiddenlinestyle = "dashed"; ld.Hiddenlines = true
local C1 = ld.cone(Origin,4*vecK,3,35,true)
local C2 = ld.cone(Origin, -4*vecK,3,35,true)
local P = {M(1,-1,-2),vecI} -- plan de la section
local I1 = g:Intersection3d(C1,P) -- intersection entre le cône C1 et le plan P
local I2 = g:Intersection3d(C2,P) -- intersection entre le cône C2 et le plan P
-- I1 et I2 sont de type Edges (arêtes)
g:Dcone(Origin,4*vecK,3,{color="orange"}); g:Dcone(Origin,-4*vecK,3,{color="orange"})
g:Lineoptions("solid","Navy",8)
g:Dedges(I1); g:Dedges(I2) -- dessin des arêtes I1 et I2
g:Dplane(P, vecK,14,9)
g:Show()
\end{luadraw}
```

FIGURE 16 : Hyperbole : intersection cône - plan

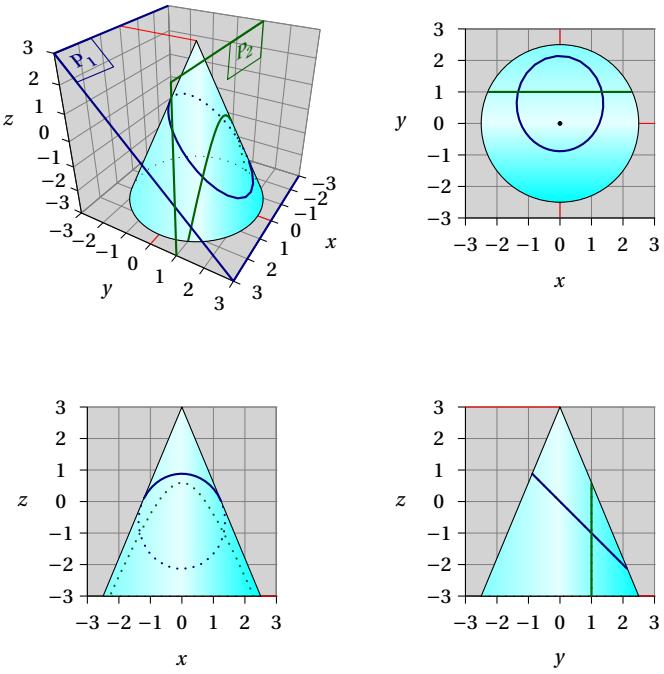


Dans cet exemple, les cônes C_1 et C_2 sont définis sous forme de polyèdres pour déterminer leur intersection avec le plan P, mais pas pour les dessiner. La méthode **g:Intersection3d(C1, P)** renvoie l'intersection du polyèdre C_1 avec le plan P sous

la forme d'une table à deux champs, un champ nommé *visible* qui contient une ligne polygonale 3D représentant les arêtes (segments) visibles de l'intersection (c'est à dire qui sont sur une facette visible de C_1), et un autre champ nommé *hidden* qui contient une ligne polygonale 3D représentant les arêtes cachées de l'intersection (c'est à dire qui sont sur une facette non visible de C_1). La méthode **g:Dedges()** permet de dessiner ce type d'objets.

```
\begin{luadraw}{name=several_views}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M
local g = ld.graph3d:new{window3d={-3,3,-3,3,-3,3}, size={10,10}, margin={0,0,0,0}}
ld.Hiddenlines = true
g:Labelsize("footnotesize")
local y0, R = 1, 2.5
local C = ld.cone(M(0,0,3),-6*vecK,R,35,true) -- cone ouvert
local P1 = {M(0,0,0),vecK+vecJ} -- 1er plan de coupe
local P2 = {M(0,y0,0),vecJ} -- 2ieme plan de coupe
local I, I2
local dessin = function() -- un dessin par vue
    g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
    I1 = g:Intersection3d(C,P1) -- intersection entre le cône C et les plans P1 et P2
    I2 = g:Intersection3d(C,P2) -- I1 et I2 sont de type Edges
    g:Dpolyline3d( {{M(0,-3,3),M(0,0,3),M(0,0,-3),M(3,0,-3)}, {M(0,0,-3),M(0,3,-3)}},"red,line width=0.4pt" )
    g:Dcone( M(0,0,3),-6*vecK,R, {color="cyan"})
    g:Dedges(I1, {hidden=true,color="Navy", width=8})
    g:Dedges(I2, {hidden=true,color="DarkGreen", width=8})
end
-- en haut à gauche, vue dans l'espace, on ajoute les plans au dessin
g:Saveattr(); g:Viewport(-5,0,0,5); g:Coordsystem(-7,6,-6,5,1); g:Setviewdir("central"); dessin()
g:Dpolyline3d( {M(-3,-3,3),M(3,-3,3),M(3,3,-3),M(-3,3,-3)},"Navy,line width=0.8pt")
g:Dpolyline3d( {M(-3,y0,3),M(3,y0,3),M(3,y0,-3)},"DarkGreen,line width=0.8pt")
g:Dlabel3d( "$P_1$",M(3,-3,3),{pos="SE",dir={-vecI,-vecJ+vecK},node_options="Navy, draw"})
g:Dlabel3d( "$P_2$",M(-3,y0,3),{pos="SW",dir={-vecI,vecK},node_options="DarkGreen,draw"})
g:Restoreattr()
-- en haut à droite, projection sur le plan xOy
g:Saveattr(); g:Viewport(0,5,0,5); g:Coordsystem(-6,6,-6,5,1); g:Setviewdir("xOy"); dessin()
g:Restoreattr()
-- en bas à gauche, projection sur le plan xOz
g:Saveattr(); g:Viewport(-5,0,-5,0); g:Coordsystem(-6,6,-6,5,1); g:Setviewdir("xOz"); dessin()
g:Restoreattr()
-- en bas à droite, projection sur le plan yOz
g:Saveattr(); g:Viewport(0,5,-5,0); g:Coordsystem(-6,6,-6,5,1); g:Setviewdir("yOz"); dessin()
g:Restoreattr()
g:Show()
\end{luadraw}
```

FIGURE 17 : Section de cône avec plusieurs vues



5) Conversion d'un polyèdre en graphe

La fonction **ld.poly2graph(P [, mode])** permet de convertir le polyèdre ⟨*P*⟩ en graphe et de renvoyer le résultat. L'argument ⟨*mode*⟩ permet de sélectionner le type de graphe, deux valeurs possibles :

- `mode="facet"` (valeur par défaut) : les sommets du graphe sont les facettes et les sommets voisins dans le graphe sont les facettes adjacentes. Le résultat est une liste de listes, l'élément numéro i du résultat est la liste des numéros des facettes adjacentes à la facette numéro i. Le numéro d'une facette est son indice dans la liste ⟨*P*⟩.facets.
- `mode="vertex"` : les sommets du graphe sont les sommets du polyèdre ⟨*P*⟩, et les sommets voisins dans le graphe sont les sommets voisins dans le polyèdre. Le résultat est une liste de listes, l'élément numéro i du résultat est la liste des numéros des sommets voisins du sommet numéro i. Le numéro d'un sommet est son indice dans la liste ⟨*P*⟩.vertices.

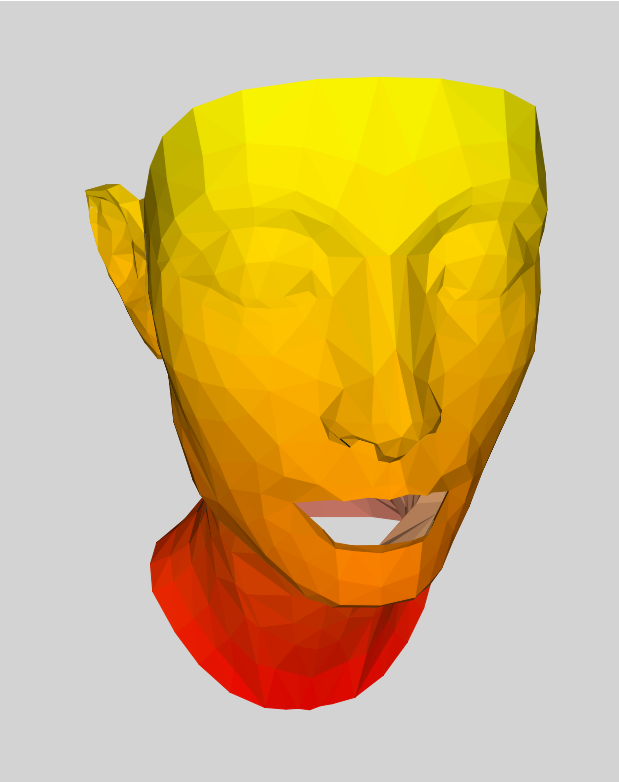
6) Lecture dans un fichier obj

La fonction **ld.read_obj_file(file)** [1] permet de lire le contenu du fichier *obj* désigné par la chaîne de caractères ⟨*file*⟩. La fonction lit les définitions des sommets (lignes commençant par `v `), et les lignes définissant les facettes (lignes commençant par `f `). Les autres lignes sont ignorées. La fonction renvoie une séquence constituée du polyèdre, suivi d'une liste de six réels `{x1,x2,y1,y2,z1,z2}` représentant la boîte 3D englobante (bounding box) du polyèdre.

```
\begin{luadraw}{name=lecture_obj}
local ld = luadraw
local P,bbox = ld.read_obj_file("obj/nefertiti.obj")
local g = ld.graph3d:new{window3d=bbox,window={-6,5,-7,7},viewdir={"central",35,65,20},
margin={0,0,0,0}, size={10,10}, bg="LightGray"}
g:Dpoly(P, {usepalette={ld.palAutumn,"z"},mode=ld.mShadedOnly})
g:Show()
\end{luadraw}
```

1. Cette fonction est une contribution de Christophe BAL.

FIGURE 18 : Masque de Nefertiti



7) Surface au format obj

Deux syntaxes possibles :

1. **ld.obj_surface(f, u1, u2, v1, v2 [, grid])** renvoie au format *obj* la surface paramétrée par la fonction $\langle f\rangle : (u,v) \mapsto f(u,v) \in \mathbf{R}^3$. L'intervalle pour le paramètre u est donné par $\langle u1\rangle$ et $\langle u2\rangle$. L'intervalle pour le paramètre v est donné par $\langle v1\rangle$ et $\langle v2\rangle$. Le paramètre facultatif $\langle grid\rangle$ vaut $\{25,25\}$ par défaut, il définit le nombre de points à calculer pour le paramètre u suivi du nombre de points à calculer pour le paramètre v (les valeurs de u et v sont équiréparties).
2. **ld.obj_surface(f, mesh)** avec $\langle mesh\rangle = \{\{u_1,\dots,u_n\},\{v_1,\dots,v_m\}\}$, renvoie la surface paramétrée par la fonction $\langle f\rangle : (u,v) \mapsto f(u,v) \in \mathbf{R}^3$. Les valeurs des paramètres u et v sont données par l'argument $\langle mesh\rangle$, elles doivent être dans l'ordre strictement croissant, mais elles ne sont pas forcément équiréparties.

Dans les deux cas, le résultat n'est pas une liste de facettes mais une table à trois champs :

`{vertices={…}, normals={…}, facets={{…},{…},…} }.`

- Les champs *vertices* et *facets* sont identiques au cas des polyèdres : listes des sommets (points 3D) et liste des facettes avec numéros des sommets, sauf qu'ici les facettes sont toutes **triangulaires**.
- Le champ *normals* est une liste de vecteurs 3D unitaires représentant des vecteurs normaux à la surface à raison de un par sommet.

8) Dessin d'une liste de facettes : Dfacet et Dmixfacet

Il y a deux méthodes possibles :

1. Pour un solide S sous forme d'une liste de facettes (avec points 3D), la méthode est :

   g:Dfacet(S, options)

   où $\langle S\rangle$ est la liste de facettes et $\langle options\rangle$ une table définissant les options. Celles-ci sont :

- `mode=ld.mShaded` : définit le mode de représentation. Les valeurs possibles sont :
  - `ld.mWireframe` : mode fil de fer, on dessine les arêtes seulement. Lorsque l'option `usepalette` est différente de `nil`, la couleur de chaque arête et calculée dans la palette (suivant le même mode que les facettes lorsque celles-ci sont peintes).
  - `ld.mFlat` ou `ld.mFlatHidden` : on dessine les faces de couleur unie, ainsi que les arêtes.
  - `ld.mShaded` ou `ld.mShadedHidden` : on dessine les faces de couleur nuancée en fonction de leur inclinaison, ainsi que les arêtes.

  - `ld.mShadedOnly` : on dessine les faces de couleur nuancée en fonction de leur inclinaison, mais pas les arêtes.
- `contrast=1` : ce nombre permet d'accentuer ou diminuer la nuance des couleurs des facettes dans les modes `ld.mShaded`, `ld.mShadedHidden`, `ld.mShadedOnly`.
- `edgestyle=<style courant>` : chaîne qui définit le style de ligne des arêtes.
- `edgecolor=<couleur courante>` : chaîne qui définit la couleur des arêtes.
- `edgewidth=<épaisseur courante>` : épaisseur de trait des arêtes en dixième de point.
- `opacity=1` : nombre entre 0 et 1 qui permet de mettre une transparence ou non sur les facettes.
- `backcull=false` : avec la valeur `true`, les facettes considérées comme non visibles (vecteur normal non dirigé vers l'observateur) ne sont pas affichées. Cette option est intéressante pour les polyèdres convexes car elle permet de diminuer le nombre de facettes à dessiner.
- `clip=false` : avec la valeur `true`, les facettes sont clippées par la fenêtre 3D.
- `twoside` : booléen qui vaut true par défaut, ce qui signifie qu'on distingue les deux côtés des facettes (intérieur et extérieur), les deux côtés n'auront pas exactement la même couleur.
- `reverse=false` : avec la valeur `true` l'orientation des facettes est inversée.
- `color="white"` : chaîne définissant la couleur de remplissage des facettes.
- `usepalette=nil` : cette option permet éventuellement de préciser une palette de couleurs pour peindre les facettes ainsi qu'un mode de calcul, la syntaxe est : `usepalette={palette,mode [, minmax]}`, où ⟨*palette*⟩ désigne une table de couleurs qui sont elles-mêmes des tables de la forme $\{r, g, b\}$ où r, g et b sont des nombres entre 0 et 1. L'argument ⟨*mode*⟩ peut être :
  - soit une des chaînes : `"x"`, `"y"`, `"z"`. Dans le premier cas par exemple, les facettes au centre de gravité d'abscisse minimale ont la première couleur de la palette, les facettes au centre de gravité d'abscisse maximale ont la dernière couleur de la palette, pour les autres, la couleur est calculée en fonction de l'abscisse du centre de gravité par interpolation linéaire.
  - soit une fonction : ⟨*mode*⟩ : $f \mapsto \text{mode}(f) \in \mathbb{R}$, où f désigne une facette (liste de points 3D). Les facettes ayant la valeur minimale ont la première couleur de la palette, celles ayant la valeur maximale ont la dernière couleur de la palette, pour les autres la couleur est calculée par interpolation linéaire.

  L'argument facultatif ⟨*minmax*⟩ est une table de deux nombres, ces deux nombres sont utilisés comme valeur minimale et valeur maximale de l'argument ⟨*mode*⟩. En l'absence de ⟨*minmax*⟩, ces deux valeurs sont calculées automatiquement.

2. Pour plusieurs listes de facettes dans un même dessin, la méthode est :

g:Dmixfacet(S1, options1, S2, options2, ...)

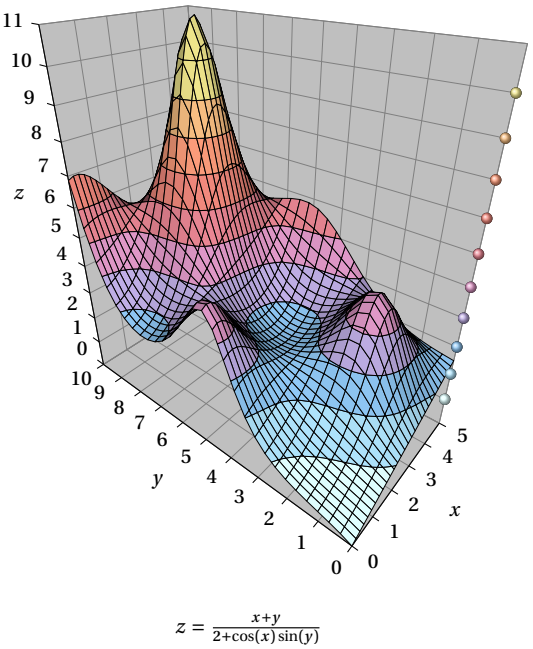
où ⟨*S1*⟩, ⟨*S2*⟩, ..., sont des listes de facettes, et ⟨*options1*⟩, ⟨*options2*⟩, ..., sont les options correspondantes. Les options d'une liste de facettes s'appliquent aussi aux suivantes si elles ne sont pas changées. Ces options sont identiques à la méthode précédente.

Cette méthode est utile pour dessiner plusieurs solides ensemble, à condition qu'il n'y ait pas d'intersections entre les objets, car celles-ci ne sont pas gérées ici.

```
\begin{luadraw}{name=courbes_niv}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M, Z = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M, ld.cpx.Z
local cos, sin = math.cos, math.sin
local g = ld.graph3d:new{window3d={0,5,0,10,0,11}, adjust2d=true, size={10,10},
    viewdir={"central",220,60,15,M(2.5,5,5.5)}}
g:Labelsize("footnotesize")
local S = ld.cartesian3d(function(u,v) return (u+v)/(2+cos(u)*sin(v)) end,0,5,0,10,{30,30})
local n = 10 -- nombre de niveaux
local Colors = ld.getpalette(ld.palGasFlame,n,true) -- liste de 10 couleurs au format table
local niv, S1 = {}
for k = 1, n do
    S1, S = ld.cutfacet(S,{M(0,0,k),-vecK}) -- section de S avec le plan z=k
    ld.insert(niv,{S1, {color=Colors[k],mode=ld.mShaded,edgewidth=0.5}}) -- S1 est la partie sous le plan et S au
     ↪ dessus
end
ld.insert(niv,{S, {color=Colors[n+1]}}) -- insertion du dernier niveau
```

```
-- niv est une liste du type {facettes1, options1, facettes2, options2, ...}
g:Dboxaxes3d({grid=true, gridcolor="gray",fillcolor="lightgray"})
g:Dmixfacet(table.unpack(niv))
for k = 1, n do
    g:Dballdots3d( M(5,0,k), ld.rgb(Colors[k]))
end
g:Dlabel("$z=\\frac{x+y}{2+\\cos(x)\\sin(y)}$", Z((g:Xinf()+g:Xsup())/2, g:Yinf()), {pos="N"})
g:Show()
\end{luadraw}
```

FIGURE 19 : Exemple de courbes de niveaux sur une surface



9) Fonctions de construction de listes de facettes

Les fonctions suivantes renvoient un solide sous forme d'une liste de facettes (avec points 3D).

surface()

Deux syntaxes possibles :

1. **ld.surface(f, u1, u2, v1, v2 [, grid])** renvoie la surface paramétrée par la fonction $\langle f\rangle : (u,v) \mapsto f(u,v) \in \mathbf{R}^3$. L'intervalle pour le paramètre u est donné par $\langle u1\rangle$ et $\langle u2\rangle$. L'intervalle pour le paramètre v est donné par $\langle v1\rangle$ et $\langle v2\rangle$. Le paramètre facultatif $\langle grid\rangle$ vaut $\{25,25\}$ par défaut, il définit le nombre de points à calculer pour le paramètre u suivi du nombre de points à calculer pour le paramètre v (les valeurs de u et v sont équiréparties).
2. **ld.surface(f, mesh)** avec $\langle mesh\rangle = \{\{u_1,\ldots,u_n\},\{v_1,\ldots,v_m\}\}$, renvoie la surface paramétrée par la fonction $\langle f\rangle : (u,v) \mapsto f(u,v) \in \mathbf{R}^3$. Les valeurs des paramètres u et v sont données par l'argument $\langle mesh\rangle$, elles doivent être dans l'ordre strictement croissant, mais elles ne sont pas forcément équiréparties.

Il y a deux variantes pour les surfaces :

cartesian3d()

La fonction **ld.cartesian3d(f, x1, x2, y1, y2 [, grid, addwall])** renvoie la surface cartésienne d'équation $z = f(x,y)$ où $\langle f\rangle : (x,y) \mapsto f(x,y) \in \mathbb{R}$. L'intervalle pour x est donné par $\langle x1\rangle$ et $\langle x2\rangle$. L'intervalle pour y est donné par $\langle y1\rangle$ et $\langle y2\rangle$. Le paramètre facultatif $\langle grid\rangle$ vaut $\{25,25\}$ par défaut, il définit le nombre de points à calculer pour x suivi du nombre de points à calculer pour y. Le paramètre $\langle addwall\rangle$ vaut `0` ou `"x"`, ou `"y"`, ou `"xy"` (`0` par défaut). Lorsque cette option vaut `"x"` (ou `"xy"`), la fonction renvoie, après la liste des facettes composant la surface, une liste de facettes séparatrices

(murs ou cloisons) entre chaque "couche" de facettes, une couche correspond à deux valeurs consécutives du paramètre x[2]. Avec la valeur `"y"` (ou `"xy"`) c'est une liste de facettes séparatrices (murs) entre chaque "couche" correspond à deux valeurs consécutives du paramètre "y"[3]. Cette option peut être utile avec la méthode **g:Dscene3d()** (uniquement), car les cloisons séparatrices forment une partition de l'espace isolant les facettes de la surface, ce qui permet d'éviter des calculs d'intersection inutiles entre elles. C'est notamment le cas avec des surfaces non convexes.

Par exemple, voici le code de la figure 1 :

```
\begin{luadraw}{name=point_col}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window3d={-2,2,-2,2,-4,4}, window={-3.5,3,-5,5}, size={8,9,0}, viewdir={120,60}}
local S = ld.cartesian3d(function(u,v) return u^2-v^2 end, -2,2,-2,2,{20,20}) -- surface d'équation z=x^2-y^2
local Tx = g:Intersection3d(S, {Origin,vecI}) --intersection entre S et le plan yOz
local Ty = g:Intersection3d(S, {Origin,vecJ}) --intersection entre S et le plan xOz
g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray",drawbox=true})
g:Dfacet(S,{mode=ld.mShadedOnly,color="ForestGreen"}) -- dessin de la surface
g:Dedges(Tx, {color="Crimson", hidden=true, width=8}) -- dessin de l'intersection avec yOz
g:Dedges(Ty, {color="Navy",hidden=true, width=8}) -- dessin de l'intersection avec xOz
g:Dpolyline3d( {M(2,0,4),M(-2,0,4),M(-2,0,-4)}, "Navy,line width=.8pt")
g:Dpolyline3d( {M(0,-2,4),M(0,2,4),M(0,2,-4)}, "Crimson,line width=.8pt")
g:Show()
\end{luadraw}
```

cylindrical_surface()

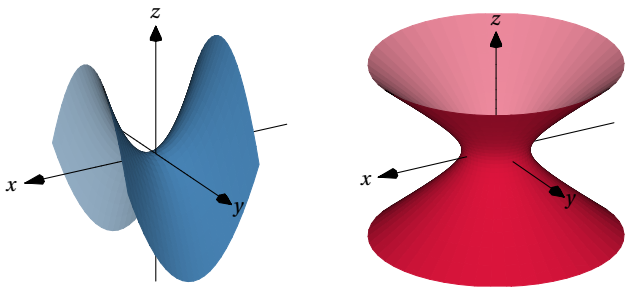
La fonction **ld.cylindrical_surface(r, z, u1, u2, theta1, theta2 [, grid, addwall])** renvoie la surface paramétrée en cylindrique par *r(u,theta), theta, z(u,theta)*. Les arguments ⟨*r*⟩ et ⟨*z*⟩ sont donc deux fonctions de u et θ, à valeurs réelles. L'intervalle pour u est donné par ⟨*u1*⟩ et ⟨*u2*⟩. L'intervalle pour θ est donné par ⟨*theta1*⟩ et ⟨*theta2*⟩ (en radians). Le paramètre facultatif ⟨*grid*⟩ vaut {25, 25} par défaut, il définit le nombre de points à calculer pour u suivi du nombre de points à calculer pour v. Le paramètre ⟨*addwall*⟩ vaut 0 ou `"v"` ou `"z"` ou `"vz"` (0 par défaut). Lorsque cette option vaut `"v"` ou `"vz"` , la fonction renvoie, après la liste des facettes composant la surface, une liste de facettes séparatrices (murs ou cloisons) entre chaque "couche" de facettes, une couche correspond à deux valeurs consécutives de l'angle θ[4]. Lorsque cette option vaut `"z"` ou `"vz"`, la fonction renvoie, après la liste des facettes composant la surface, une liste de facettes séparatrices (murs ou cloisons) entre chaque "couche" de facettes, une couche correspond à deux valeurs consécutives de la cote z[5], les valeurs de z sont calculées à partir des valeurs du paramètres u et avec la valeur ⟨*theta1*⟩, ceci est utile lorsque z ne dépend que u (et donc pas de θ). Cette option peut être utile avec la méthode **g:Dscene3d()** (uniquement), car les cloisons séparatrices forment une partition de l'espace isolant les facettes de la surface, ce qui permet d'éviter des calculs d'intersection inutiles entre elles. C'est notamment le cas avec des surfaces non convexes.

```
\begin{luadraw}{name=surface_with_addWall}
local ld = luadraw
local pi, ch, sh = math.pi, math.cosh, math.sinh
local O = ld.pt3d.Origin
local g = ld.graph3d:new{window3d={-4,4,-4,4,-5,5}, window={-10,10,-4,4}, size={10,10}, viewdir={60,60}}
g:Labelsize("footnotesize")
local S,wall = ld.cartesian3d(function(x,y) return x^2-y^2 end,-2,2,-2,2,nil,"xy")
g:Saveattr(); g:Viewport(-10,0,-4,4); g:Coordsystem(-4.5,4.5,-4.5,4.75)
g:Dscene3d(
    g:addWall(wall), -- 2 intersections de facettes avec cette instruction, et 529 sans
    g:addFacet(S,{color="SteelBlue"}),
    g:addAxes(O,{arrows=1}) )
g:Restoreattr()
g:Saveattr(); g:Viewport(0,10,-4,4); g:Coordsystem(-5,5,-5,5)
local r = function(u,v) return ch(u) end
local z = function(u,v) return sh(u) end
```

2. Ces cloisons sont en fait des plans d'équation x =constante
3. Ces cloisons sont en fait des plans d'équation y =constante
4. Ces cloisons sont en fait des plans d'équation θ =constante
5. Ces cloisons sont en fait des plans d'équation z =constante

```
S,wall = ld.cylindrical_surface(r,z,-2,2,-pi,pi,{25,51},"zv")
g:Dscene3d(
    g:addWall(wall), -- 13 intersections de facettes avec cette instruction, et plus de 17000 sans ...
    g:addFacet(S,{color="Crimson"}),
    g:addAxes(O,{arrows=1}) )
g:Restoreattr()
g:Show()
\end{luadraw}
```

FIGURE 20 : Surfaces utilisant l'option *addwall*



curve2cone()

La fonction **ld.curve2cone(f, t1, t2, S, options)** construit un cône de sommet $\langle S\rangle$ (point 3D) et de base la courbe paramétrée par $\langle f\rangle$: $t \mapsto f(t) \in \mathbf{R}^3$ sur l'intervalle défini par $\langle t1\rangle$ et $\langle t2\rangle$. L'argument $\langle options\rangle$ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

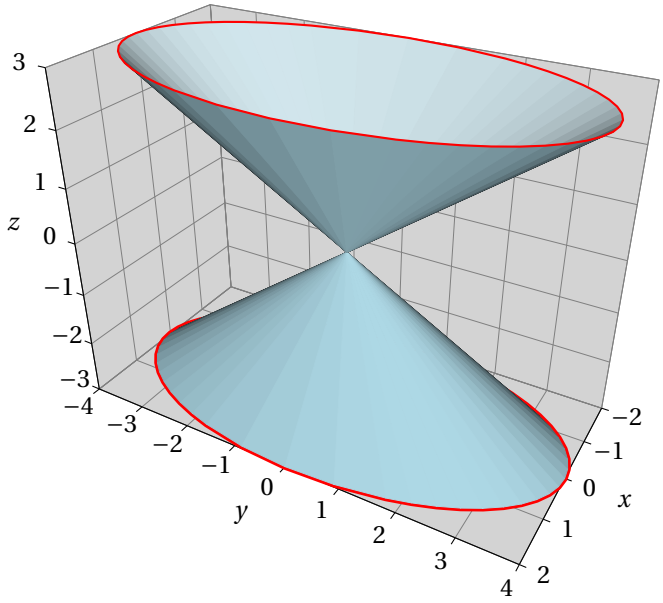
- `nbdots=15` : nombre minimal de points de la courbe à calculer.
- `ratio=0` : nombre représentant le rapport d'homothétie (de centre le sommet $\langle S\rangle$) pour construire l'autre partie du cône, avec la valeur 0 il n'y a pas de deuxième partie.
- `nbdiv=0` : entier positif indiquant le nombre de fois que l'intervalle entre deux valeurs consécutives du paramètre t peut être coupé en deux (dichotomie) lorsque les points correspondants sont trop éloignés.
- `obj=false` : avec la valeur `false` cette fonction construit une liste de facettes, avec les valeur `true` elle construit une table à trois champs : `{vertices={points 3D}, facets={{index1,…},…}, normals={vecteurs 3D}}`.

La fonction renvoie une séquence :

1. la liste de facettes ou bien la table au format *obj*, suivie
2. d'une ligne polygonale 3D qui représente les bords du cône.

```
\begin{luadraw}{name=curve2cone}
local ld = luadraw
local O, M = ld.pt3d.Origin, ld.pt3d.M
local cos, sin, pi = math.cos, math.sin, math.pi
local g = ld.graph3d:new{ window3d={-2,2,-4,4,-3,3}, window={-5.5,5.5,-5.5,5},
    size={10,10}, viewdir="central"}
local f = function(t) return M(2*cos(t),4*sin(t),-3) end -- ellipse dans le plan z=-3
local C, bord = ld.curve2cone(f,-pi,pi,O,{nbdiv=2, ratio=-1})
g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
g:Dpolyline3d(bord[1],"red,line width=2.4pt") -- bord inférieur
g:Dfacet(C, {mode=ld.mShadedOnly,color="LightBlue"}) -- cône
g:Dpolyline3d(bord[2],"red,line width=0.8pt") -- bord supérieur
g:Show()
\end{luadraw}
```

FIGURE 21 : Exemple de cône elliptique



curve2cylinder()

La fonction **ld.curve2cylinder(f, t1, t2, V [, options])** construit un cylindre d'axe dirigé par le vecteur ⟨*V*⟩ (point 3D) et de base la courbe paramétrée par ⟨*f*⟩ : $t \mapsto f(t) \in \mathbf{R}^3$ sur l'intervalle défini par ⟨*t1*⟩ et ⟨*t2*⟩. La seconde base est la translatée de la première avec le vecteur ⟨*V*⟩. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `nbdots=15` : nombre minimal de points de la courbe à calculer.
- `nbdiv=0` : entier positif indiquant le nombre de fois que l'intervalle entre deux valeurs consécutives du paramètre *t* peut être coupé en deux (dichotomie) lorsque les points correspondants sont trop éloignés
- `obj=false` : avec la valeur `false` cette fonction construit une liste de facettes, avec les valeur `true` elle construit une table à trois champs : `{vertices={points 3D}, facets={{index1,…},…}, normals={vecteurs 3D}}`.

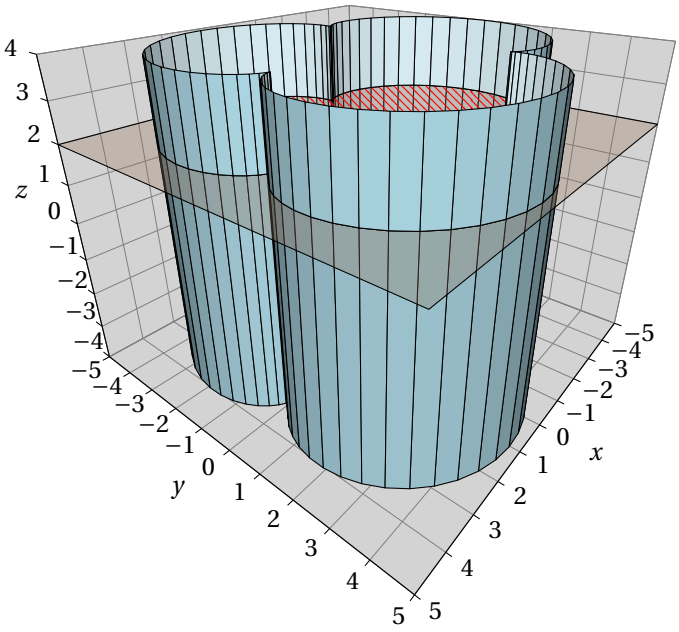
La fonction renvoie une séquence :

1. la liste de facettes ou bien la table au format *obj*, suivie
2. d'une ligne polygonale 3D qui représente les bords du cylindre.

```
\begin{luadraw}{name=curve2cylinder}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M
local cos, sin, pi = math.cos, math.sin, math.pi

local g = ld.graph3d:new{ window3d={-5,5,-5,5,-4,4},window={-9,8,-10.5,5.5},
    viewdir={"central",39,64}, size={10,10}}
local f = function(t) return M(4*cos(t)-cos(4*t),4*sin(t)-sin(4*t),-4) end -- courbe dans le plan z=-3
local V = 8*vecK
local C = ld.curve2cylinder(f,-pi,pi,V,{nbdots=25,nbdiv=2})
local plan = {M(0,0,2), -vecK} -- plan de coupe z=2
local C1, C2, section = ld.cutfacet(C,plan)
g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
g:Dfacet(C1, {mode=ld.mShaded,color="LightBlue"}) -- partie sous le plan
g:Dfacet(g:Plane2facet(plan), {opacity=0.3,color="Chocolate"}) -- dessin du plan sous forme d'une facette
g:Filloptions("fdiag","red"); g:Dpolyline3d(section) -- dessin de la section
g:Dfacet(C2, {mode=ld.mShaded,color="LightBlue"}) -- partie du cylindre au dessus du plan
g:Show()
\end{luadraw}
```

FIGURE 22 : Section d'un cylindre non circulaire



line2tube() ; section2tube()

La fonction **ld.line2tube(L, r [, options])** construit un tube centré sur ⟨*L*⟩ qui doit être une liste de points 3D, l'argument ⟨*r*⟩ représente le rayon de ce tube. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `nbfacet=3` : nombre de facettes latérales du tube.
- `close=false` : booléen indiquant si la ligne polygonale doit être refermée.
- `hollow=false` : booléen indiquant si les deux extrémités du tube doivent être ouvertes ou non. Lorsque l'option `close` vaut `true`, l'option `hollow` prend automatiquement la valeur `true`.
- `angle={0,360}` : par défaut le tube est circulaire, avec cette option on peut modifier l'angle de départ et d'arrivée (en degrés) pour que la section ne soit qu'une partie d'un cercle.
- `opensection=false` : avec la valeur `true` la section ne sera pas refermée, c'est à dire qu'il n'y aura pas de facettes entre le dernier et le premier point de la section, le tube aura une ouverture latérale.
- `addwall=0` : nombre qui vaut 0 ou 1. Lorsque cette option vaut 1, la fonction renvoie, après le tube, une liste de facettes séparatrices (murs) entre chaque "tronçon" du tube, ce qui peut être utile avec la méthode **g:Dscene3d()** (uniquement).
- `obj=false` : avec la valeur `false` cette fonction renvoie une liste de facettes, avec les valeur `true` elle renvoie une table à trois champs : `{vertices={points 3D}, facets={{index1,…},…}, normals={vecteurs 3D}}`.

La fonction **ld.section2tube(section, L [, options])** construit également un tube centré sur ⟨*L*⟩ qui doit être une liste de points 3D, l'argument ⟨*section*⟩ doit être une facette centrée sur le premier point de ⟨*L*⟩, elle représente une section du tube qui va être construit. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

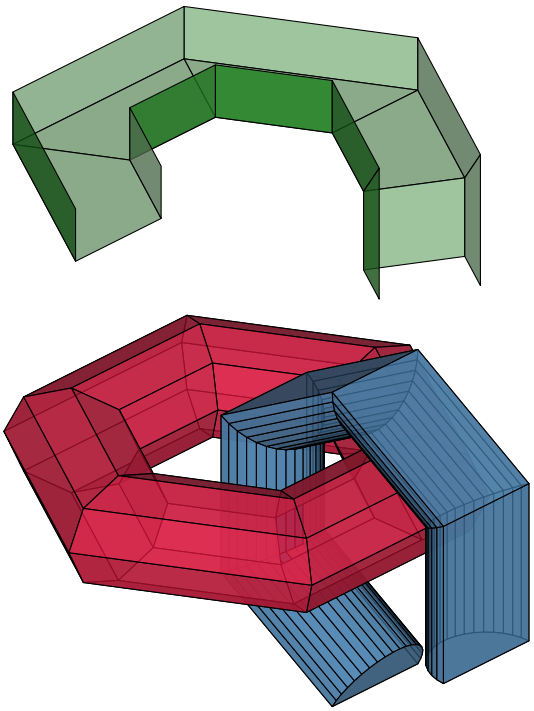
- `nbfacet=3` : nombre de facettes latérales du tube.
- `close=false` : booléen indiquant si la ligne polygonale doit être refermée.
- `hollow=false` : booléen indiquant si les deux extrémités du tube doivent être ouvertes ou non. Lorsque l'option `close` vaut `true`, l'option `hollow` prend automatiquement la valeur `true`.
- `opensection=false` : avec la valeur `true` la section ne sera pas refermée, c'est à dire qu'il n'y aura pas de facettes entre le dernier et le premier point de la section, le tube aura une ouverture latérale.
- `addwall=0` : nombre qui vaut 0 ou 1. Lorsque cette option vaut 1, la fonction renvoie, après le tube, une liste de facettes séparatrices (murs) entre chaque "tronçon" du tube, ce qui peut être utile avec la méthode **g:Dscene3d()** (uniquement).
- `obj=false` : avec la valeur `false` cette fonction renvoie une liste de facettes, avec les valeur `true` elle renvoie une

table à trois champs : `{vertices={points 3D}, facets={{index1,…},…}, normals={vecteurs 3D}}`.

```
\begin{luadraw}{name=line2tube_section2tube}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window={-5,6,-4.5,8}, viewdir={45,60}, margin={0,0,0,0}, size={10,10}}
local L1 = ld.map(pt3d.toPoint3d,ld.polyreg(0,3,6)) -- hexagone régulier dans le plan xOy, centre O de sommet M(3,0,0)
local L2 = ld.shift3d(ld.rotate3d(L1,90,{Origin,vecJ}),3*vecJ)
local L3 = ld.shift3d(ld.reverse(L1),6*vecK)
L3[6] = L3[5]-2*vecK -- modification du dernier point
local section = ld.shift3d({M(4,0,0.5),M(4,0,-0.5),M(2,0,-0.5),M(2,0,0.5)},6*vecK)
local T1 = ld.line2tube(L1,1,{nbfacet=8,close=true}) -- tube 1 refermé
local T2 = ld.line2tube(L2,1,{nbfacet=20, angle={-90,90}}) -- tube 2 semi-circulaire non refermé
local T3 = ld.section2tube(section, L3,{hollow=true, opensection=true}) -- ouverture latérale
g:Dmixfacet( T1, {color="Crimson",opacity=0.8}, T2, {color="SteelBlue"}, T3, {color="ForestGreen"} )
g:Show()
\end{luadraw}
```

FIGURE 23 : Exemple avec line2tube et section2tube



read_table3d()

La fonction **ld.read_table3d(table, options)** permet d'interpréter le contenu de l'argument ⟨*table*⟩. Cette fonction renvoie une table dont les champs contiennent, suivant les options, des facettes, des sommets, des arêtes, ou des fonctions.

- Par défaut, l'argument ⟨*table*⟩ représente les valeurs d'une (ou plusieurs) fonction(s) $f_j : (x,y) \mapsto f_j(x,y) \in \mathbb{R}$ $(1 \leqslant j \leqslant n)$ sur un pavé $[x_1;x_2] \times [y_1;y_2]$, avec un certain nombre de valeurs pour x dans $[x_1;x_2]$, et un certain nombre de valeurs pour y dans $[y_1;y_2]$. La ⟨*table*⟩ est une liste de listes du type :

$$\{x_i, y_i, f_1(x_i,y_i), \ldots, f_n(x_i,y_i)\}$$

  Ces listes doivent être rangées dans l'ordre lexicographique suivant x et y (ou suivant y et x). Par exemple, voici la table représentant la fonction $(x,y) \mapsto x^2+y^2$ sur $[-1;1] \times [-1;1]$ avec 5 valeurs par intervalle (donc 25 listes) :

```
{ { -1.0, -1.0, 2.0 },
{ -1.0, -0.5, 1.25 },
{ -1.0, 0.0, 1.0 },
```

```
{ -1.0, 0.5, 1.25 },
{ -1.0, 1.0, 2.0 },
{ -0.5, -1.0, 1.25 },
{ -0.5, -0.5, 0.5 },
{ -0.5, 0.0, 0.25 },
{ -0.5, 0.5, 0.5 },
{ -0.5, 1.0, 1.25 },
{ 0.0, -1.0, 1.0 },
{ 0.0, -0.5, 0.25 },
{ 0.0, 0.0, 0.0 },
{ 0.0, 0.5, 0.25 },
{ 0.0, 1.0, 1.0 },
{ 0.5, -1.0, 1.25 },
{ 0.5, -0.5, 0.5 },
{ 0.5, 0.0, 0.25 },
{ 0.5, 0.5, 0.5 },
{ 0.5, 1.0, 1.25 },
{ 1.0, -1.0, 2.0 },
{ 1.0, -0.5, 1.25 },
{ 1.0, 0.0, 1.0 },
{ 1.0, 0.5, 1.25 },
{ 1.0, 1.0, 2.0 } }
```

De telles tables peuvent aussi être lues dans un fichier avec la fonction **ld.read_csv_file()** à raison d'une liste par ligne.

Remarque : toutes les valeurs de la table doivent être numériques.

- L'argument ⟨*options*⟩ est une table dont les champs sont les options, celles-ci sont (avec leur valeur par défaut) :
  - `mode="xyz"` : chaîne de caractères définissant le type de table. Le mode `"xyz"` (par défaut) correspond au type décrit ci-dessus. Il y a deux autres modes possibles :
    * `mode="y/x"`, dans ce la cas la table est de la forme suivante :

```
    x1  x2  ... xn
y1  z11 z12 ... z1n
y2  z21 z22 ... z2n
...
yp  zp1 zp2 ... zpn
```

    * `mode="x/y"`, dans ce la cas la table est de la forme suivante :

```
    y1  y2  ... yp
x1  z11 z12 ... z1p
x2  z21 z22 ... z2p
...
xn  zn1 zn2 ... znp
```

  - `header=nil`, liste de chaînes de caractères représentant le nom des colonnes (aucune par défaut).
  - `x=1`, `y=2`, `z=3` : numéros des colonnes représentant respectivement les valeurs de x, de y et z. Lorsqu'une option `header` a été précisée, on peut utiliser le nom des colonnes à la place des numéros.
  - `func={}` : liste de numéros de colonnes (ou noms de colonnes si l'option `header` a été précisée). Chacune des colonnes mentionnées sera transformée en fonction de x et y (par interpolation linéaire), la table renvoyée aura un champ appelé *func* contenant la liste de ces fonctions dans l'ordre (s'il n'y a qu'une fonction, alors le champ *func* contiendra cette fonction).
  - `facets=true` : avec la valeur `true` (par défaut), la table renvoyée aura un champ appelé *facets* contenant la liste des facettes (une facette est une liste de points 3D).

- edges=false : avec la valeur true, la table renvoyée aura un champ appelé *edges* contenant les contours des facettes rectangulaires (ligne polygonale 3D).
- vertices=false : avec la valeur true, la table renvoyée aura un champ appelé *vertices* contenant les points 3D M(x,y,z) de la table.
- triangle=false : avec la valeur true, les facettes seront triangulaires et non pas rectangulaires.
- bbox=false : avec la valeur true, la table renvoyée aura un champ appelé *bbox* contenant une table de la forme $\{x_1,x_2,y_1,y_1,z_1,z_2\}$ représentant la boîte englobante.

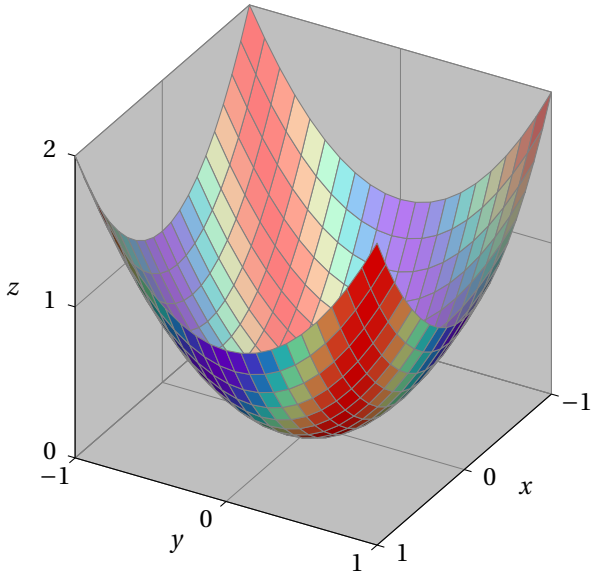
Pour faire un exemple, nous créons le fichier *data3d.csv* avec le code suivant :

```
local ld = luadraw
local f = io.open(ld.cachedir.."data3d.csv", "w")
local h = function(x,y) return x^2+y^2 end
local color = function(x,y) return math.cos(math.pi*(x-y)) end
local xvalues, yvalues = ld.linspace(-1,1,21), ld.linspace(-1,1,21)
for _,x in ipairs(xvalues) do
    for _,y in ipairs(yvalues) do
        f:write( x.." "..y.." "..h(x,y).." "..color(x,y).."\n")
    end
end
f:close()
```

Lecture et utilisation du fichier :

```
\begin{luadraw}{name=read_table3d}
local ld = luadraw
local pt3d  = ld.pt3d
local data3d = ld.read_csv_file( ld.cachedir.."data3d.csv", {header=false, sep=" "})
local S = ld.read_table3d(data3d, {bbox=true, func=4})
local color = function(f) -- f is a facet
    local G = pt3d.isobar3d(f)
    return S.func(G.x,G.y)
end
local g = ld.graph3d:new{ window3d=S.bbox, adjust2d=true, size={10,10} }
g:Dboxaxes3d({grid=true, gridcolor="gray", fillcolor="lightgray"})
g:Dfacet( S.facets, {usepalette={ld.palRainbow,color}, edgecolor=ld.Gray, reverse=true})
g:Show()
\end{luadraw}
```

FIGURE 24 : La fonction *ld.read_table3d()*



rotcurve()

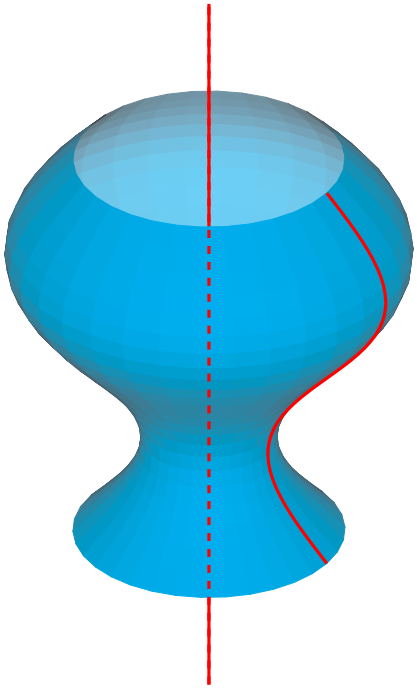
La fonction **ld.rotcurve(p, t1, t2, axe, angle1, angle2 [, options])** construit sous forme d'une liste de facettes, la surface balayée par la courbe paramétrée par ⟨*p*⟩ : $t \mapsto p(t) \in \mathbf{R}^3$ sur l'intervalle défini par ⟨*t1*⟩ et ⟨*t2*⟩, en la faisant tourner autour de ⟨*axe*⟩ (qui est une table de la forme {point 3D, vecteur 3D} représentant une droite orientée de l'espace), d'un angle allant de ⟨*angle1*⟩ (en degrés) à ⟨*angle2*⟩. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `grid={25,25}` : table constituée de deux nombres, le premier est le nombre de points calculés pour le paramètre t, et le second le nombre de points calculés pour le paramètre angulaire.
- `addwall=0` : nombre qui vaut 0 ou 1 ou 2. Lorsque cette option vaut 1, la fonction renvoie, après la surface, une liste de facettes séparatrices (murs) entre chaque "couche" de facettes (une couche correspond à deux valeurs consécutives du paramètre t), et avec la valeur 2 c'est une liste de facettes séparatrices (murs) entre chaque "tranche" de rotation (une couche correspond à deux valeurs consécutives du paramètre angulaire, ceci est intéressant lorsque la courbe est dans un même plan que l'axe de rotation). Cette option peut être utile avec la méthode **g:Dscene3d()** (uniquement).
- `obj=false` : avec la valeur `false` cette fonction renvoie une liste de facettes, avec les valeur `true` elle renvoie une table à trois champs : `{vertices={points 3D}, facets={{index1,…},…}, normals={vecteurs 3D}}`.

```
\begin{luadraw}{name=rotcurve}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M
local cos, sin, pi = math.cos, math.sin, math.pi
local g = ld.graph3d:new{viewdir={30,60},size={10,10}}
local p = function(t) return M(0,sin(t)+2,t) end -- courbe dans le plan yOz
local axe = {Origin,vecK}
local S = ld.rotcurve(p,pi,-pi,axe,0,360,{grid={25,35}})
local visible, hidden = g:Classifyfacet(S)
g:Dfacet(hidden, {mode=ld.mShadedOnly,color="cyan"})
g:Dline3d(axe,"red,line width=1.2pt")
g:Dfacet(visible, {mode=ld.mShadedOnly,color="cyan"})
g:Dline3d(axe,"red,line width=1.2pt,dashed")
g:Dparametric3d(p,{t={-pi,pi},draw_options="red,line width=1.2pt"})
g:Show()
```

```
\end{luadraw}
```

FIGURE 25 : Exemple avec rotcurve



Remarque : si l'orientation de la surface ne semble pas bonne, il suffit d'échanger les paramètres ⟨*t1*⟩ et ⟨*t2*⟩, ou bien ⟨*angle1*⟩ et ⟨*angle2*⟩.

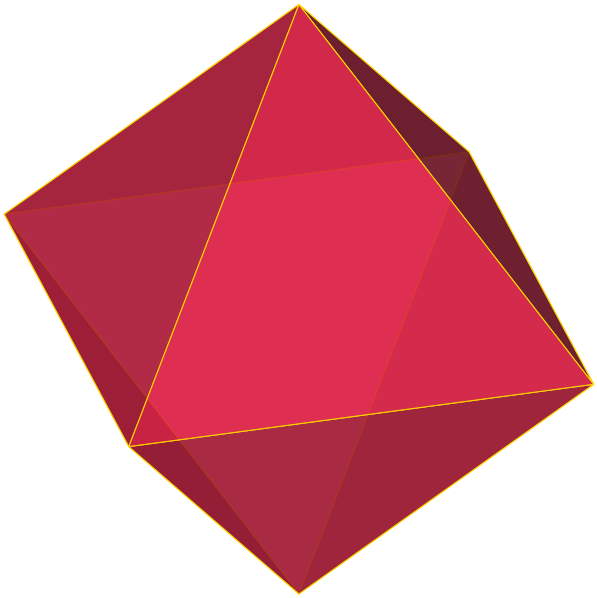
rotline()

La fonction **ld.rotline(L, axe, angle1, angle2 [, options])** construit sous forme d'une liste de facettes, la surface balayée par la liste de points 3D ⟨*L*⟩ en la faisant tourner autour de ⟨*axe*⟩ (qui est une table de la forme {point 3D, vecteur 3D} représentant une droite orientée de l'espace), d'un angle allant de ⟨*angle1*⟩ (en degrés) à ⟨*angle2*⟩. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `nbdots=25` : nombre de points calculés pour le paramètre angulaire.
- `close=false` : booléen qui indique si ⟨*L*⟩ doit être refermée.
- `addwall=0` : nombre qui vaut 0 ou 1 ou 2. Lorsque cette option vaut 1, la fonction renvoie, après la liste des facettes composant la surface, une liste de facettes séparatrices (murs) entre chaque "couche" de facettes (une couche correspond à deux valeurs consécutives du paramètre *t*), et avec la valeur 2 c'est une liste de facettes séparatrices (murs) entre chaque "tranche" de rotation (une couche correspond à deux valeurs consécutives du paramètre angulaire, ceci est intéressant lorsque la courbe est dans un même plan que l'axe de rotation). Cette option peut être utile avec la méthode **g:Dscene3d()** (uniquement).
- `obj=false` : avec la valeur `false` cette fonction renvoie une liste de facettes, avec les valeur `true` elle renvoie une table à trois champs : `{vertices={points 3D}, facets={{index1,…},…}, normals={vecteurs 3D}}`.

```
\begin{luadraw}{name=rotline}
local ld = luadraw
local pt3d = ld.pt3d
local M = pt3d.M
local g = ld.graph3d:new{window={-4,4,-4,4},size={10,10}}
local L = {M(0,0,4),M(0,4,0),M(0,0,-4)} -- liste de points dans le plan yOz
local axe = {pt3d.Origin, pt3d.vecK}
local S = ld.rotline(L,axe,0,360,{nbdots=5}) -- le point 1 et le point 5 sont confondus
g:Dfacet(S,{color="Crimson",edgecolor="Gold",opacity=0.8})
g:Show()
\end{luadraw}
```

FIGURE 26 : Exemple avec rotline



10) Arêtes d'un solide

Un objet de type "edge" est une table à deux champs, un champ nommé *visible* qui contient une ligne polygonale 3D correspondant aux arêtes visibles, et un autre champ nommé *hidden* qui contient une ligne polygonale 3D correspondant aux arêtes cachées.

- La méthode **g:Edges(P)** où ⟨*P*⟩ est un polyèdre, renvoie les arêtes de P sous forme d'un objet de type "edge". Une arête de argu est visible lorsqu'elle appartient à au moins une face visible.
- La méthode **g:Intersection3d(P, plane)** où argu est un polyèdre ou bien une liste de facettes, renvoie sous forme d'objet de type "edge" l'intersection entre argu et le plan représenté par ⟨*plane*⟩ (c'est une table de la forme {A,u} où A est un point du plan et *u* un vecteur normal, ce sont donc deux points 3D).
- La méthode **g:Dedges(edges, options)** permet de dessiner ⟨*edges*⟩ qui doit être un objet de type "edge". L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :
  - `hidden=false` : booléen qui indique si les arêtes cachées doivent être dessinées).
  - `visible=true` : booléen qui indique si les arêtes visibles doivent être dessinées.
  - `clip=false` : booléen qui indique si les arêtes doivent être clippées par la fenêtre 3D.
  - `hiddenstyle=ld.Hiddenlinestyle` : chaîne de caractères définissant le style de ligne des arêtes cachées, par défaut cette option contient la valeur de la variable globale `ld.Hiddenlinestyle` (qui vaut `"dotted"` par défaut).
  - `hiddencolor=color` : chaîne de caractères définissant la couleur des arêtes cachées, par défaut cette option contient la même couleur que l'option `color`.
  - `style=<style courant>` : chaîne de caractères définissant le style de ligne des arêtes visibles.
  - `color=<couleur courante>` : chaîne de caractères définissant la couleur des arêtes visibles.
  - `width=<épaisseur courante>` : nombre représentant l'épaisseur de trait des arêtes (en dixième de point).
- **Complément** :
  - La fonction **ld.facetedges(F)** où ⟨*F*⟩ est une liste de facettes ou bien un polyèdre, renvoie une liste de segments 3D représentant toutes les arêtes de ⟨*F*⟩. Le résultat n'est pas un objet de type "edge", et il se dessine avec la méthode **g:Dpolyline3d()**. Bien sûr, chaque arête n'apparaît qu'une seule fois dans la liste.
  - La fonction **ld.facetvertices(F)** où ⟨*F*⟩ est une liste de facettes ou bien un polyèdre, renvoie la liste de tous les sommets de ⟨*F*⟩ (points 3D).

11) Méthodes et fonctions s'appliquant à des facettes ou polyèdres

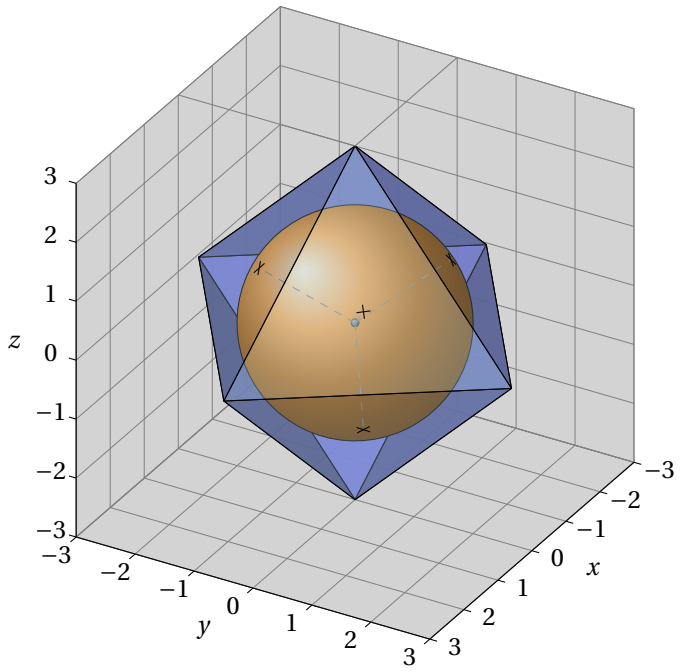
- La méthode **g:Isvisible(F)** où ⟨*F*⟩ désigne **une** facette (liste d'au moins 3 points 3D coplanaires et non alignés), renvoie `true` si la facette ⟨*F*⟩ est visible (vecteur normal dirigé vers l'observateur). Si A, B et C sont les trois premiers points de ⟨*F*⟩, le vecteur normal est calculé en faisant le produit vectoriel $\vec{AB} \wedge \vec{AC}$.
- La méthode **g:Classifyfacet(F)** où ⟨*F*⟩ est une liste de facettes ou bien un polyèdre, renvoie **deux** listes de facettes, la première est la liste des facettes visibles, et la suivante, la liste des facettes non visibles.
- La méthode **g:Sortfacet(F [, backcull])** où ⟨*F*⟩ est une liste de facettes, renvoie cette liste de facettes triées de la plus éloignée à la plus proche de l'observateur. L'argument facultatif ⟨*backcull*⟩ est un booléen qui vaut `false` par défaut, lorsqu'il a la valeur `true`, les facettes non visibles sont exclues du résultat (seules les facettes visibles sont alors renvoyées après avoir été triées). Le calcul de l'éloignement d'un facette se fait sur son centre de gravité. La technique dite du "peintre" consiste à afficher les facettes de la plus éloignée à la plus proche, donc dans l'ordre de la liste renvoyée par cette fonction (le résultat affiché n'est cependant pas toujours correct en fonction de la taille et de la forme des facettes).
- La méthode **g:Sortpolyfacet(P [, backcull])** où ⟨*P*⟩ est un polyèdre, renvoie la liste des facettes de ⟨*P*⟩ (facettes avec points 3D) triées de la plus éloignée à la plus proche de l'observateur. L'argument facultatif ⟨*backcull*⟩ est un booléen qui vaut `false` par défaut, lorsqu'il a la valeur `true`, les facettes non visibles sont exclues du résultat comme pour la méthode précédente. Ces deux méthodes de tris sont utilisées par les méthodes de dessin de polyèdres ou facettes (**g:Dpoly()**, **g:Dfacet()** et **g:Dmixfacet()**).
- La méthode **g:Outline(P)** où ⟨*P*⟩ est un polyèdre, renvoie le "contour" de ⟨*P*⟩ sous la forme d'une table à deux champs, un champ nommé *visible* qui contient une ligne polygonale 3D représentant les "arêtes" (segments) appartenant à une seule facette, celle-ci étant visible, ou bien à deux facettes, une visible et une cachée ; l'autre champ est nommé *hidden* et contient une ligne polygonale 3D représentant les "arêtes" appartenant à une seule facette, celle-ci étant cachée.
- La fonction **ld.border(P)** où ⟨*P*⟩ est un polyèdre ou une liste de facette, renvoie une ligne polygonale 3D qui correspond aux arêtes appartenant à une seule facette de ⟨*P*⟩ (ces arêtes sont mises "bout à bout" pour former une ligne polygonale).
- La fonction **ld.getfacet(P [, list])** où ⟨*P*⟩ est un polyèdre, renvoie la liste des facettes de ⟨*P*⟩ (avec points 3D) dont le numéro figure dans la table ⟨*list*⟩. Si l'argument ⟨*list*⟩ n'est pas précisé, c'est la liste de toutes les facettes de ⟨*P*⟩ qui est renvoyée (dans ce cas c'est la même chose que **ld.poly2facet(P)**).
- La fonction **ld.facet2plane(L)** où ⟨*L*⟩ est soit une facette, soit une liste de facettes, renvoie soit le plan contenant la facette, soit la liste des plans contenant chacune des facettes de ⟨*L*⟩. Un plan est une table du type {A,u} où A est un point du plan et *u* un vecteur normal au plan (donc deux points 3D).
- La fonction **ld.reverse_face_orientation(F)** où ⟨*F*⟩ et soit une facette, soit une liste de facette, soit un polyèdre, renvoie un résultat de même nature que ⟨*F*⟩ mais dans lequel l'ordre sur les sommets de chaque facette a été inversé. Cela peut être utile lorsque l'orientation de l'espace à été modifiée.

```
\begin{luadraw}{name=sphere_octaedre}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local poly = require "luadraw_polyhedrons"
local g = ld.graph3d:new{ window3d={-3,3,-3,3,-3,3}, size={10,10}}
local P = poly.octahedron(Origin,M(0,0,3)) -- polyèdre défini dans le module luadraw_polyhedrons
P = ld.rotate3d(P,-10,{Origin,vecK}) -- rotate3d sur un polyèdre renvoie un polyèdre
local V, H = g:Classifyfacet(P) -- V pour facettes visibles, H pour hidden
local S = ld.map(function(p) return {ld.proj3d(Origin,p),p[2]} end, ld.facet2plane(V) )
-- S contient la liste de : {projeté, vecteur normal} (projetés de Origin sur les faces visibles)
local R = pt3d.abs(S[1][1]) -- rayon de la sphère
g:Dboxaxes3d({grid=true, gridcolor="gray", fillcolor="LightGray"})
g:Dfacet(H, {color="blue",opacity=0.9}) -- dessin des facettes non visibles
g:Dsphere(Origin,R,{mode=ld.mBorder,color="orange"}) -- dessin de la sphère
g:Dballdots3d(Origin,"gray",0.75) -- centre de la sphère
for _,D in ipairs(S) do -- segments reliant l'origine aux projetés
    g:Dpolyline3d( {Origin,D[1]},"dashed,gray")
end
```

```
g:Dfacet(V,{opacity=0.4, color="LightBlue"}) -- facettes visibles de l'octaèdre
g:Dcrossdots3d(S,nil,0.75) -- dessin des projetés sur les faces
g:Dpolyline3d( {M(0,-3,3), M(0,0,3), M(-3,0,3)},"gray")
g:Show()
\end{luadraw}
```

FIGURE 27 : Sphère inscrite dans un octaèdre avec projection du centre sur les faces



12) Découper un solide : cutpoly, cutfacet et cutfacet2

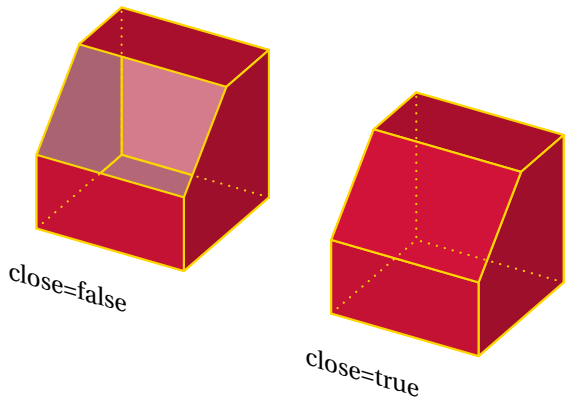
- La fonction **ld.cutpoly(P, plane [, close])** permet de couper le polyèdre ⟨*P*⟩ avec le plan ⟨*plane*⟩ (table du type {A,n} où A est un point du plan et *n* un vecteur normal au plan). La fonction renvoie 3 choses : la partie située dans le demi-espace contenant le vecteur *n* (sous forme d'un polyèdre), suivie de la partie située dans l'autre demi-espace (toujours sous forme d'un polyèdre), suivie de la section sous forme d'une facette orientée par $-n$. Lorsque l'argument facultatif ⟨*close*⟩ vaut `true`, la section est ajoutée aux deux polyèdres résultants, ce qui a pour effet de les refermer (`false` par défaut).

  Remarque : lorsque le polyèdre ⟨*P*⟩ n'est pas convexe, le résultat de la section n'est pas toujours correct.

```
\begin{luadraw}{name=cutpoly}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window3d={-3,3,-3,3,-3,3}, window={-4,4,-3,3},size={10,10}}
local P = ld.parallelep(M(-1,-1,-1),2*vecI,2*vecJ,2*vecK)
local A, B, C = M(0,-1,1), M(0,1,1), M(1,-1,0)
local plane = {A, pt3d.prod(B-A,C-A)}
local P1 = ld.cutpoly(P,plane)
local P2 = ld.cutpoly(P,plane,true)
g:Lineoptions(nil,"Gold",8)
g:Dpoly( ld.shift3d(P1,-2*vecJ), {color="Crimson",mode=ld.mShadedHidden} )
g:Dpoly( ld.shift3d(P2,2*vecJ), {color="Crimson",mode=ld.mShadedHidden} )
g:Dlabel3d(
    "close=false", M(2,-2,-1), {dir={vecJ,vecK}},
    "close=true", M(2,2,-1), {} )
g:Show()
\end{luadraw}
```

FIGURE 28 : Cube coupé par un plan (cutpoly), avec *close*=false et avec *close*=true

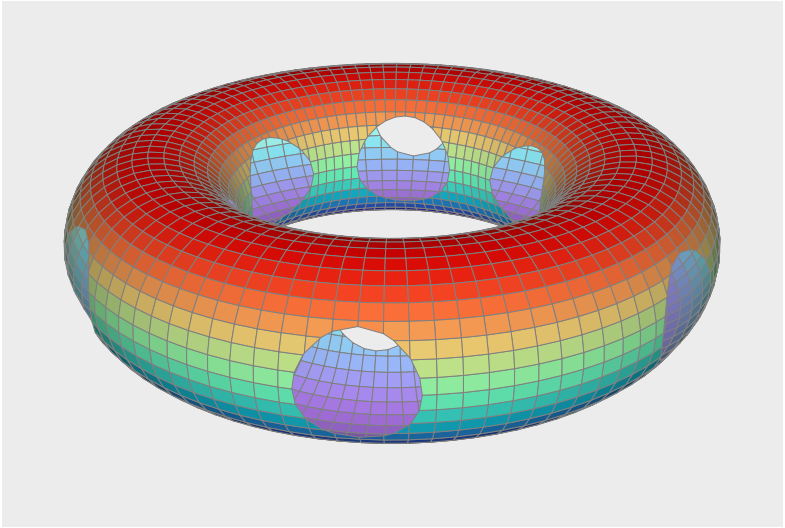


- La fonction **ld.cutfacet(F, plane [, close])**, où ⟨F⟩ est une facette, une liste de facettes, ou un polyèdre, fait la même chose que la fonction précédente sauf que cette fonction renvoie des listes de facettes et non pas des polyèdres. Cette fonction a été utilisée dans l'exemple des courbes de niveau à la figure 19.
- La fonction **ld.cutfacet2(F, f, signe)**, permet également un découpage. L'argument ⟨F⟩ est une facette, une liste de facettes, ou un polyèdre. L'argument ⟨f⟩ est une fonction ⟨f⟩ : $\mathbf{R}^3 \to \mathbf{R}$, et l'argument ⟨*signe*⟩ est soit la chaîne `">"`, soit la chaîne `"<"`. Les facettes sont découpées avec la surface d'équation $f(x,y,z)=0$, la fonction renvoie deux listes de facettes, la première liste contient les facettes qui vérifient $f(x,y,z)>0$ lorsque ⟨*signe*⟩=`">"` (ou celles qui vérifient $f(x,y,z)<0$ lorsque ⟨*signe*⟩=`"<"`).

  Attention : la fonction f doit être continue, et doit changer **au plus une fois** de signe le long de chaque arête.

```
\begin{luadraw}{name=torus_with_holes}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M, pt3d.Mc
local g = ld.graph3d:new{ window={-4,4,-3,2}, bg="lightgray!30", viewdir={"central",5,65}, size={10,10} }
local cos, sin, pi = math.cos, math.sin, math.pi
local R, r, rc = 2.75, 1, 0.6
local f = function(t) return M(R+r*cos(t),0,r*sin(t)) end
local C  = ld.rotcurve(f,-pi,pi,{Origin,vecK}, 30,-30, {grid={37,17}}) -- un sixième du tore
local f = function(x,y,z) return y^2+z^2-rc^2 end -- f(x,y,z)=0 est l'équation du cylindre
C = ld.cutfacet2(C,f,">") -- C est découpé avec le cylindre
local torus = table.copy(C)
for k = 1, 5 do -- reconstitution du tore en entier
    torus = ld.concat(torus, ld.rotate3d(C,k*60,{Origin,vecK}))
end
g:Dfacet( torus, {usepalette={ld.palRainbow,"z"}, edgecolor="gray", edgewidth=1})
g:Show()
\end{luadraw}
```

FIGURE 29 : La fonction *ld.cutfacet2*



13) Clipper des facettes avec un polyèdre convexe : clip3d

La fonction **ld.clip3d(S, P [, exterior])** clippe le solide ⟨*S*⟩ (liste de facettes ou bien polyèdre) avec le solide convexe ⟨*P*⟩ (liste de facettes ou bien polyèdre) et renvoie la liste de facettes qui en résulte. L'argument facultatif ⟨*exterior*⟩ est un booléen qui vaut `false` par défaut, dans ce cas c'est la partie de ⟨*S*⟩ qui est intérieure à ⟨*P*⟩ qui est renvoyée, sinon c'est la partie de ⟨*S*⟩ extérieure à ⟨*P*⟩ qui est renvoyée.

Remarque : le résultat n'est pas toujours satisfaisant pour la partie extérieure.

Cas particulier : clipper une liste de facettes S (ou bien un polyèdre) avec la fenêtre 3D courante peut se faire avec cette fonction de la manière suivante :

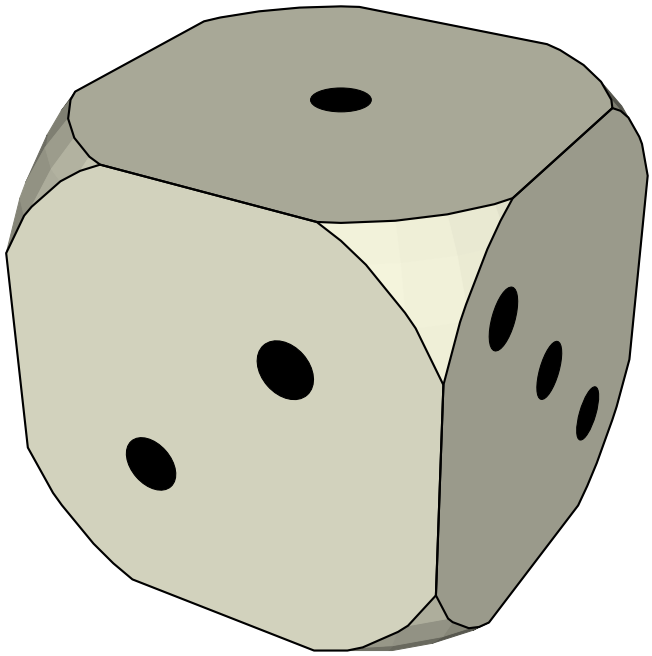
S = ld.clip3d(S, g:Box3d())

En effet, la méthode **g:Box3d()** renvoie la fenêtre 3D courante sous forme d'un parallélépipède.

```
\begin{luadraw}{name=clip3d}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window={-3,3,-3,3},size={10,10}, viewdir="central"}
local S = ld.sphere(Origin,3)
local C = ld.parallelep(M(-2,-2,-2),4*vecI,4*vecJ,4*vecK)
local C1 = ld.clip3d(S,C) -- sphère clippée par le cube
local C2 = ld.clip3d(C,S) -- cube clippé par la sphère
local V = g:Classifyfacet(C2) -- facettes visibles de C2
g:Dfacet( ld.concat(C1,C2), {color="Beige",mode=ld.mShadedOnly,backcull=true} ) -- que les faces visibles
g:Dpolyline3d(V,true,"line width=0.8pt") -- contour des faces visibles de C2
local A, B, C, D = M(2,-2,-2), M(2,2,2), M(-2,2,-2), M(0,0,2) -- dessin des points noirs
g:Filloptions("full","black")
g:Dcircle3d( D,0.25,vecK); g:Dcircle3d( (2*A+B)/3,0.25,vecI)
g:Dcircle3d( (A+2*B)/3,0.25,vecI); g:Dcircle3d( (3*B+C)/4,0.25,vecJ)
g:Dcircle3d( (B+C)/2,0.25,vecJ); g:Dcircle3d( (B+3*C)/4,0.25,vecJ)
g:Show()
\end{luadraw}
```

FIGURE 30 : Exemple avec clip3d : construction d'un dé à partir d'un cube et d'une sphère



14) Clipper un plan avec un polyèdre convexe : clipplane

La fonction **ld.clipplane(plane, P)**, où l'argument ⟨*plane*⟩ est une table de la forme *{A,n}* représentant le plan passant par A (point 3D) et de vecteur normal n (point 3D non nul), et ⟨P⟩ est un polyèdre convexe, renvoie la section du polyèdre par le plan, si elle existe, sous forme d'une facette (liste de points 3D) orientée par n.

V La méthode Dscene3d

1) Le principe, les limites

Le défaut majeur des méthodes **g:Dpoly()**, **g:Dfacet()** et **g:Dmixfacet()** est de ne pas gérer les intersections éventuelles entre facettes de différents solides, sans compter que parfois, même pour un polyèdre convexe simple, l'algorithme du peintre ne donne pas toujours le bon résultat (car le tri de facettes se fait uniquement sur leur centre de gravité). D'autre part, ces méthodes permettent de dessiner uniquement des facettes.

Le principe de la méthode **g:Dscene3d()** est de classer les objets 3D à dessiner (facettes, lignes polygonales, points, labels,...) dans un arbre (qui représente la scène). À chaque nœud de l'arbre il y a un objet 3D, appelons-le A, et deux descendants, l'un des descendants va contenir les objets 3D qui sont devant l'objet A (c'est à dire plus près de l'observateur que A), et l'autre descendant va contenir les objets 3D qui sont derrière l'objet A (c'est à dire plus loin de l'observateur que A).

En particulier, pour classer une facette B par rapport à une facette A qui est déjà dans l'arbre, on procède ainsi : on découpe la facette B avec le plan contenant la facette A, ce qui donne en général deux "demi" facettes, une qui sera devant A (celle dans le demi-espace "contenant" l'observateur) , et l'autre qui sera donc derrière A.

Cette méthode est efficace mais comporte des limites car elle peut entraîner une explosion du nombre de facettes dans l'arbre augmentant ainsi sa taille de manière exponentielle, ce qui peut rendre rédhibitoire l'utilisation de cette méthode lorsqu'il y a beaucoup de facettes (temps de calcul long[6], taille trop importante du fichier **.tkz*, temps de dessin par TikZ trop long). Par contre, elle est très efficace lorsqu'il y a peu de facettes, et donc peu d'intersections de facettes (objets convexes avec peu de facettes). De plus, il est possible de dessiner sous la scène 3D et au-dessus, c'est à dire avant l'utilisation de la méthode **g:Dscene3d()**, et après son utilisation.

Cette méthode doit donc être réservée à des scènes très simples. Pour des scènes 3D complexes le format vectoriel n'est pas adapté, mieux vaux se tourner alors vers des d'autres outils comme POV-Ray ou Blender ou WebGL ...

6. Lua est un langage interprété donc l'exécution est en général plus longue qu'avec un langage compilé.

2) Construction d'une scène 3D

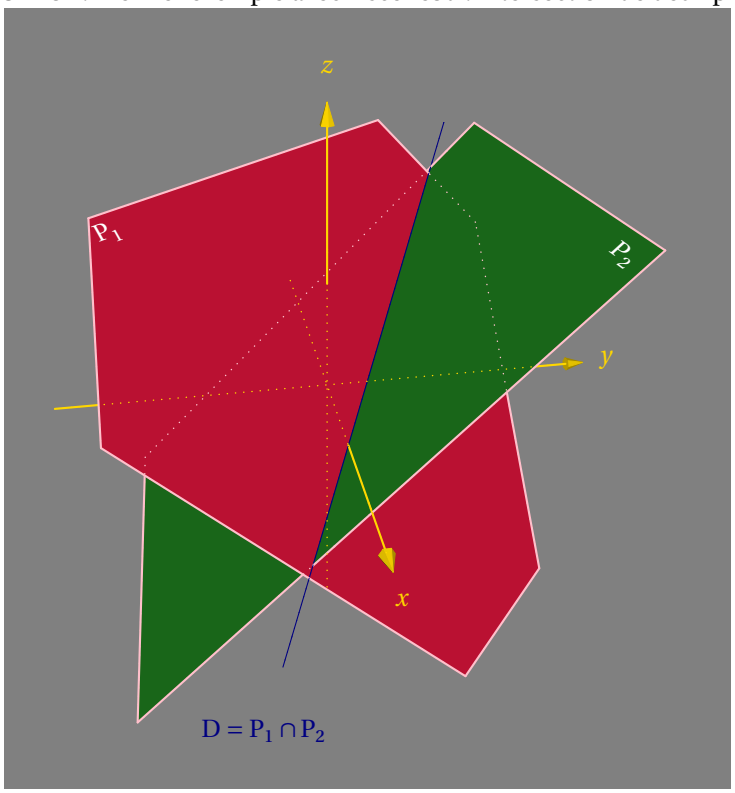
La méthode **g:Dscene3d(...)** permet cette construction. Elle prend en argument les objets 3D qui vont constituer cette scène les uns après les autres. Ces objets 3D sont eux-mêmes fabriqués à partir de méthodes dédiées qui vont être détaillées plus loin. Dans la version actuelle, ces objets 3D peuvent être :

- des polyèdres,
- des listes de facettes (avec points 3D),
- des lignes polygonales 3D,
- des points 3D,
- des labels,
- des axes,
- des plans, des droites,
- des angles,
- des cercles, des arcs de cercle.

```
\begin{luadraw}{name=intersection_plans}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{viewdir={"central",-10,60}, window={-5,6.5,-6.5,6},bg="gray", size={10,10}}
local P1 = ld.planeEq(1,1,1,-2) -- plan d'équation x+y+z-2=0
local P2 = {Origin, vecK-vecJ} -- plan passant par O et normal à (0,-1,1)
local D = ld.interPP(P1,P2) -- droite d'intersection entre P1 et P2 (D = {A,u})
local posD = D[1]+1.85*D[2] -- pour placer le label
ld.Hiddenlines = true; ld.Hiddenlinestyle = "dotted" -- affichage des lignes cachées en pointillées
g:Dscene3d(
    g:addPlane(P1, {color="Crimson",edge=true,edgecolor="Pink",edgewidth=8}), -- ajout du plan P1
    g:addPlane(P2, {color="ForestGreen",edge=true,edgecolor="Pink",edgewidth=8}), -- ajout du plan P2
    g:addLine(D, {color="Navy",edgewidth=12}), -- ajout de la droite D
    g:addAxes(Origin, {arrows=1, color="Gold",width=8}), -- ajout des axes fléchés
    g:addLabel( -- ajout de labels, ceux-ci auraient pu être ajoutés par dessus la scène
        "$D=P_1\\cap P_2$",posD,{color="Navy"},
        "$P_2$", M(3,0,0)+3.5*M(0,1,1)+0.2*vecI,{color="white",dir={vecI,vecJ+vecK}},
        "$P_1$",M(2,0,0)+1.85*M(-1,-1,2)-1.5*M(-1,1,0), {dir={M(-1,1,0),M(-1,-1,2)}} )
)
g:Show()
\end{luadraw}
```

FIGURE 31 : Premier exemple avec Dscene3d : intersection de deux plans



3) Méthodes pour ajouter un objet dans la scène 3D

Ces méthodes sont à utiliser comme argument de la méthode **g:Dscene3d(...)** comme dans l'exemple ci-dessus.

Ajouter des facettes : g:addFacet et g:addPoly

La méthode **g:addFacet(list, options)** où ⟨*list*⟩ est une facette ou bien une liste de facettes (avec points 3D), permet d'ajouter ces facettes à la scène.

La méthode **g:addPoly(P, options)** permet d'ajouter le polyèdre ⟨*P*⟩ à la scène.

Dans les deux cas, l'argument ⟨*options*⟩ est une table dont les champs définissent ces options (avec leur valeur par défaut) sont :

- `color="white"` : définit la couleur de remplissage des facettes, cette couleur sera nuancée en fonction de l'inclinaison de celles-ci. Par défaut, le bord des facettes n'est pas dessiné (seulement le remplissage).
- `usepalette=nil`, cette option permet de préciser une palette de couleurs pour peindre les facettes ainsi qu'un mode de calcul, la syntaxe est : `usepalette={palette,mode[,minmax]}`, où ⟨*palette*⟩ désigne une table de couleurs qui sont elles-mêmes des tables de la forme $\{r, g, b\}$ où r, g et b sont des nombres entre 0 et 1. L'argument ⟨*mode*⟩ peut être :
  - soit une des chaînes : `"x"`, `"y"`, `"z"`. Dans le premier cas par exemple, les facettes au centre de gravité d'abscisse minimale ont la première couleur de la palette, les facettes au centre de gravité d'abscisse maximale ont la dernière couleur de la palette, pour les autres, la couleur est calculée en fonction de l'abscisse du centre de gravité par interpolation linéaire.
  - soit une fonction : ⟨*mode*⟩ : $f \mapsto \text{mode}(f) \in \mathbb{R}$, où f désigne une facette (liste de points 3D). Les facettes ayant la valeur minimale ont la première couleur de la palette, celles ayant la valeur maximale ont la dernière couleur de la palette, pour les autres la couleur est calculée par interpolation linéaire.

  L'argument facultatif ⟨*minmax*⟩ est une table de deux nombres, ces deux nombres sont utilisés comme valeur minimale et valeur maximale de l'argument ⟨*mode*⟩. En l'absence de ⟨*minmax*⟩, ces deux valeurs sont calculées automatiquement.
- `opacity=1` : nombre entre 0 et 1 pour définir l'opacité des facettes (1 signifie pas de transparence).
- `backcull=false` : booléen qui indique si les facettes non visibles doivent être exclues de la scène. Par défaut elles sont présentes.
- `clip=false` : booléen qui indique si les facettes doivent être clippées par la fenêtre 3D.

- `contrast=1` : valeur numérique permettant d'accentuer ou diminuer de contraste de couleur entre les facettes. Avec la valeur 0 toutes les facettes ont la même couleur.
- `twoside=true` : booléen qui indique si on distingue la face interne de la face externe des facettes. La couleur de la face interne est un peu plus claire que celle de l'externe.
- `reverse=false` : avec la valeur `true` l'orientation des facettes est inversée.
- `edge=false` : booléen qui indique si les arêtes doivent être ajoutées à la scène.
- `edgecolor=<couleur courante>` : indique la couleur des arêtes lorsqu'elles sont dessinées, c'est la couleur courante par défaut.
- `edgewidth=<épaisseur courante>` : indique l'épaisseur de trait (en dixième de point) des arêtes.
- `hidden=ld.Hiddenlines` : booléen qui indique si les arêtes cachées doivent être représentées. `ld.Hiddenlines` est une variable globale qui vaut `false` par défaut.
- `hiddenstyle=ld.Hiddenlinestyle` : chaîne définissant le style de ligne des arêtes cachées. `ld.Hiddenlinestyle` est une variable globale qui vaut `"dotted"` par défaut.
- `hiddenscale=ld.Hiddenlinescale` : nombre représentant un pourcentage, l'épaisseur des lignes cachées est égale à celle des lignes visibles multipliée par ce nombre. `ld.Hiddenlinescale` est une variable globale qui vaut `2/3` par défaut.
- `matrix=ld.ID3d` : matrice 3D de transformation des facettes, par défaut celle-ci est la matrice 3D de l'identité, c'est à dire la table `{M(0,0,0),vecI,vecJ,vecK}`.

Ajouter un plan : g:addPlane et g:addPlaneEq

La méthode **g:addPlane(P, options)** permet d'ajouter le plan $\langle P \rangle$ à la scène 3D, ce plan est défini sous la forme d'une table {A, u} où A est un point du plan (point 3D) et u un vecteur normal au plan (point 3D non nul). Cette fonction détermine l'intersection entre ce plan et le parallélépipède donné par l'option `window3d` (définie à la création du graphe), ce qui donne une facette, c'est celle-ci qui est ajoutée à la scène. Cette méthode utilise **g:addFacet()**.

La méthode **g:addPlaneEq(coef, options)** où $\langle coef \rangle$ est une table constituée de quatre réels $\{a, b, c, d\}$, permet d'ajouter à la scène le plan d'équation $ax + by + cz + d = 0$ (cette méthode utilise la précédente).

Dans les deux cas, l'argument facultatif $\langle options \rangle$ est une table dont les champs définissent les options, celles-ci sont celles de la méthode **g:addFacet()**, plus les options :

- `rectangle=nil` : avec la valeur `nil` le plan est intersecté avec la fenêtre 3D et c'est la facette qui en résulte qui sera dessinée, elle n'est pas forcément rectangulaire, avec `rectangle={V,L1,L2}`, la facette dessinée sera rectangulaire avec des côtés de longueurs $\langle L1 \rangle$ et $\langle L2 \rangle$, et le vecteur $\langle V \rangle$ (qui doit appartenir au plan) imposera la direction d'un côté du rectangle. Ce vecteur $\langle V \rangle$ est facultatif, s'il est omis la fonction le choisira elle-même. L'argument $\langle L2 \rangle$ est facultatif, s'il est omis, il a implicitement la même valeur que $\langle L1 \rangle$.
- `scale=1` : ce nombre est un rapport d'homothétie, il est pris en compte seulement lorsque `rectangle=nil`, dans ce cas, on applique à la facette l'homothétie de centre le centre de gravité de la facette et de rapport `scale`. Cela permet de jouer sur la taille du plan dans sa représentation.

Ajouter une ligne polygonale : g:addPolyline

La méthode **g:addPolyline(L, options)** où $\langle L \rangle$ est une liste de points 3D, ou une liste de listes de points 3D, permet d'ajouter $\langle L \rangle$ à la scène. L'argument $\langle options \rangle$ est une table dont les champs définissent les options, celles-ci (avec leur valeur par défaut) sont :

- `style=<style courant>` : pour définir le style de ligne, c'est le style courant par défaut.
- `color=<couleur courante>` : couleur de la ligne, c'est la couleur courante par défaut.
- `close=false` : indique si la ligne $\langle L \rangle$ (ou chaque composante de $\langle L \rangle$) doit être refermée.
- `clip=false` : indique si la ligne $\langle L \rangle$ (ou chaque composante de $\langle L \rangle$) doit être clippée par la fenêtre 3D.
- `width=<épaisseur courante>` : épaisseur de la ligne en dixième de point, c'est l'épaisseur courante par défaut.
- `opacity=1` : opacité du tracé de ligne (1 signifie pas de transparence).
- `hidden=ld.Hiddenlines` : booléen qui indique si les parties cachées de la ligne doivent être représentées. `ld.Hiddenlines` est une variable globale qui vaut `false` par défaut.
- `hiddenstyle=ld.Hiddenlinestyle` : chaîne définissant le style de ligne des parties cachées. `ld.Hiddenlinestyle` est une variable globale qui vaut `"dotted"` par défaut.

- `hiddenscale=ld.Hiddenlinescale` : nombre représentant un pourcentage, l'épaisseur des lignes cachées est égale à celle des lignes visibles multipliée par ce nombre. `ld.Hiddenlinescale` est une variable globale qui vaut `2/3` par défaut.
- `arrows=0` : cette option peut valoir 0 (aucune flèche ajoutée à la ligne), 1 (une flèche ajoutée en fin de ligne), ou 2 (une flèche en début et en fin de ligne). Les flèches sont des petits cônes.
- `arrowscale={1,1}`, permet de jouer sur la taille des flèches (largeur pour le premier nombre, hauteur pour le second). Lorsque `arrowscale` est un nombre, la deuxième valeur est considérée égale à la première.
- `matrix=ld.ID3d` : matrice 3D de transformation (de la ligne), par défaut celle-ci est la matrice 3D de l'identité, c'est à dire la table `{M(0,0,0),vecI,vecJ,vecK}`.
- `double=nil` : cette option expérimentale permet d'ajouter une bordure de chaque côté de la ligne, la syntaxe est `double={border color, border width}`. Avec la valeur par défaut (`nil`) il n'y a pas de bordure.

Ajouter des axes : g:addAxes

La méthode **g:addAxes(O, options)** permet d'ajouter les axes : ($\langle O\rangle$,*vecI*), ($\langle O\rangle$,*vecJ*) et ($\langle O\rangle$,*vecK*) à la scène 3D, où l'argument $\langle O\rangle$ est un point 3D. Les options sont celles de la méthode **g:addPolyline()**, plus l'option `legend=true` qui permet d'ajouter automatiquement une légende à l'extrémité de chaque axe, ces légendes sont gérées par l'option `labels={"$x$", "$y$","$z$"}`. Les axes ne sont pas gradués.

Ajouter une droite : g:addLine

La méthode **g:addLine(d, options)** permet d'ajouter la droite $\langle d\rangle$ à la scène, cette droite est une table de la forme {A, u} où A est un point de la droite (point 3D) et u un vecteur directeur (point 3D non nul). L'argument facultatif $\langle options\rangle$ est une table dont les champs définissent les options, celles-ci sont celles de la méthode **g:addPolyline()**, plus l'option `scale=1` : ce nombre est un rapport d'homothétie, on applique l'homothétie de centre le milieu du segment représentant la droite, et de rapport `scale`. Cela permet de jouer sur la taille du segment dans sa représentation, ce segment est la droite clippée par le polyèdre donné par l'option `window3d` (définie à la création du graphe), ce qui donne une segment (éventuellement vide).

Ajouter un angle "droit" : g:addAngle

La méthode **g:addAngle(B, A, C [, r, options])** permet d'ajouter l'angle $\widehat{\mathrm{BAC}}$ sous forme d'un parallélogramme de côté $\langle r\rangle$ (0.25 par défaut), seuls deux côtés sont représentés. les arguments $\langle B\rangle$, $\langle A\rangle$ et $\langle C\rangle$ sont des points 3D. Les options sont celles de la méthode **g:addPolyline()**.

Ajouter un arc de cercle : g:addArc

La méthode **g:addArc(B, A, C, r, sens [, normal, options])** permet d'ajouter l'arc de cercle de centre $\langle A\rangle$ (point 3D), de rayon $\langle r\rangle$, allant de $\langle B\rangle$ vers $\langle C\rangle$ (points 3D) dans le sens direct si $\langle sens\rangle$ vaut 1 (indirect sinon). L'arc est tracé dans le plan passant par $\langle A\rangle$ et orthogonal au vecteur$\langle normal\rangle$ (point 3D non nul), c'est ce même vecteur qui oriente le plan. Si le vecteur $\langle normal\rangle$ n'est pas précisé, alors par défaut celui-ci sera le produit vectoriel $\vec{\mathrm{AB}} \wedge \vec{\mathrm{AC}}$. Les options sont celles de la méthode **g:addPolyline()**.

Ajouter un cercle : g:addCircle

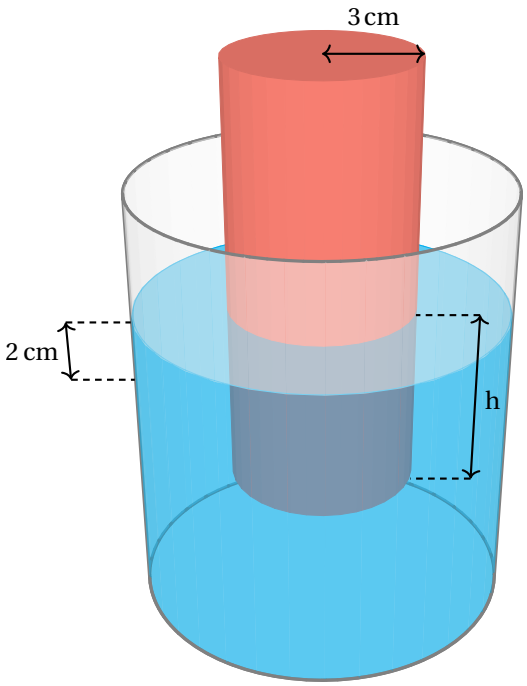
La méthode **g:addCircle(A, r, normal, options)** permet d'ajouter le cercle de centre $\langle A\rangle$ (point 3D) et de rayon $\langle A\rangle$ dans le plan passant par $\langle A\rangle$ et orthogonal au vecteur$\langle normal\rangle$ (point 3D non nul). Les options sont celles de la méthode **g:addPolyline()**.

```
\begin{luadraw}{name=cylindres_imbriques}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window={-5,5,-7,5}, viewdir={"central",30,65,20},
    size={10,10},margin={0,0,0,0}}
ld.Hiddenlines = false
```

```
local R, r, A, B = 3, 1.5
local C1 = ld.cylinder(M(0,0,-5),5*vecK,R) -- pour modéliser l'eau
local C2 = ld.cylinder(Origin,2*vecK,R,35,true) -- partie du contenant au dessus de l'eau (cylindre ouvert)
local C3 = ld.cylinder(M(0,0,-3),7*vecK,r) -- petit cylindre plongé dans l'eau
-- sous la scène 3D
g:Lineoptions(nil,"gray",12)
g:Dcylinder(M(0,0,-5),7*vecK,R,{hiddenstyle="noline"}) -- contour du contenant (grand cylindre)
-- scène 3D
g:Dscene3d(
    g:addPoly(C1,{contrast=0.125,color="cyan",opacity=0.5}), -- eau
    g:addPoly(C2,{contrast=0.125,color="WhiteSmoke", opacity=0.5}), -- partie du contenant au-dessus de l'eau
    g:addPoly(C3,{contrast=0.25,color="Salmon",backcull=true}), -- petit cylindre dans l'eau
    g:addCircle(M(0,0,2),R,vecK,{color="gray"}), -- bord supérieur du contenant
    g:addCircle(M(0,0,-5),R,vecK,{color="gray"}), -- bord inférieur du contenant
    g:addCircle(Origin,R-0.025,vecK, {width=2,color="cyan"}) -- bord supérieur eau
)
-- par dessus la scène 3D
g:Lineoptions(nil,"black",8); A = 4*vecK; B = A+r*g:ScreenX()
g:Dpolyline3d( {A,B}, "<->"); g:Dlabel3d("$3\\,$cm",(A+B)/2,{pos="N",dist=0.25})
A = Origin+(r+1)*g:ScreenX(); A = ld.rotate3d(A,-10,{Origin,vecK})
B = A-3*vecK
g:Dpolyline3d( {A,B}, "<->"); g:Dlabel3d("h",(A+B)/2,{pos="E"})
g:Lineoptions("dashed")
g:Dpolyline3d({{A,A-g:ScreenX()},{B,B-g:ScreenX()}})
A = Origin-(R+1)*g:ScreenX(); A = ld.rotate3d(A,10,{Origin,vecK})
B = A-vecK
g:Dpolyline3d({{A,A+g:ScreenX()},{B,B+g:ScreenX()}})
g:Linestyle("solid")
g:Dpolyline3d( {A,B}, "<->"); g:Dlabel3d("$2$\\,cm",(A+B)/2,{pos="W"})
g:Show()
\end{luadraw}
```

FIGURE 32 : Cylindre plein plongé dans de l'eau



Remarques :

- La méthode **g:ScreenX()** renvoie le vecteur de l'espace (point 3D) correspondant au vecteur d'affixe 1 dans le plan de l'écran, et la méthode **g:ScreenY()** renvoie le vecteur de l'espace (point 3D) correspondant au vecteur d'affixe $\imath$ dans le plan de l'écran.
- Pour le petit cylindre (C3) on utilise l'option `backcull=true` pour diminuer le nombre de facettes, par contre, on ne

le fait pas pour les deux autres cylindres (C1 et C2) car ils sont transparents.

Ajouter des points : g:addDots

La méthode **g:addDots(dots, options)** permet d'ajouter des points 3D à la scène. L'argument ⟨*dots*⟩ est soit un point 3D, soit une liste de points 3D. L'argument facultatif ⟨*options*⟩ est une table dont les champs définissent les options, celles-ci sont (avec la valeur par défaut) :

- `style="ball"` : chaîne définissant le style de points, ce sont tous les styles de points 2D, plus le style `"ball"` (sphère) qui est le style par défaut.
- `color="black"` : chaîne définissant la couleur des points.
- `scale=1` : nombre permettant de jouer sur la taille des points.
- `matrix=ld.ID3d` : matrice 3D de transformation, par défaut celle-ci est la matrice 3D de l'identité, c'est à dire la table `{M(0,0,0),vecI,vecJ,vecK}`.

Ajouter des labels : g:addLabel

La méthode **g:addLabel(text1, anchor1, options1, text2, anchor2, options2, ...)** permet d'ajouter les labels ⟨*text1*⟩, ⟨*text2*⟩, etc. Les arguments (obligatoires) ⟨*anchor1*⟩, ⟨*anchor2*⟩, etc, sont des points 3D représentant les points d'ancrage des labels. Les arguments (obligatoires) ⟨*options1*⟩, ⟨*options2*⟩, etc, sont des tables dont les champs définissent les options, celles-ci sont (avec la valeur par défaut) :

- `color=<couleur courante des labels>` : chaîne définissant la couleur du label.
- `pos=<style courant des labels>` : chaîne définissant la position du label par rapport au point d'ancrage (comme en 2D : `"N"`, `"NW"`, `"W"`, ...), initialisée au style en cours des labels.
- `dist=0` : exprime la distance entre le label et son point d'ancrage (dans le plan de l'écran).
- `size=<taille courante>` : chaîne définissant la taille du label.
- `dir={}` : table définissant le sens de l'écriture dans l'espace (sens usuel par défaut). En général, `dir={dirX,dirY,dep}`, et les 3 valeurs ⟨*dirX*⟩, ⟨*dirY*⟩ et ⟨*dep*⟩ sont trois points 3D représentant 3 vecteurs, les deux premiers (obligatoires) indiquent le sens de l'écriture, le troisième (facultatif) indique un déplacement (translation) du label par rapport au point d'ancrage.
- `showdot=false` : booléen qui indique si un point (2D) doit être dessiné au point d'ancrage.
- `matrix=ld.ID3d` : matrice 3D de transformation, par défaut celle-ci est la matrice 3D de l'identité, c'est à dire la table `{M(0,0,0),vecI,vecJ,vecK}`.

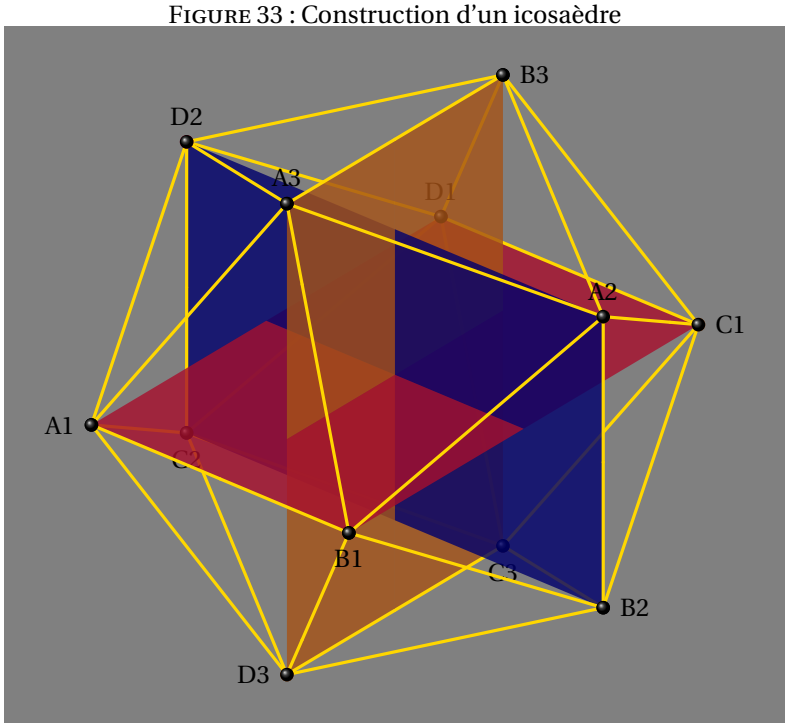
Remarque : comme en 2D, les options d'un label s'appliquent aux labels suivants tant qu'elles ne sont pas modifiées.

```
\begin{luadraw}{name=icosaedre}
local ld = luadraw
local M = ld.pt3d.M

local g = ld.graph3d:new{window={-2.25,2.25,-2,2}, viewdir={40,60},bg="gray",size={10,10},margin={0,0,0,0}}
ld.Hiddenlines = false
local phi = (1+math.sqrt(5))/2 -- nombre d'or
local A1, B1, C1, D1 = M(phi,-1,0), M(phi,1,0), M(-phi,1,0), M(-phi,-1,0) -- dans le plan z=0
local A2, B2, C2, D2 = M(0,phi,1), M(0,phi,-1), M(0,-phi,-1), M(0,-phi,1) -- dans le plan x=0
local A3, B3, C3, D3 = M(1,0,phi), M(-1,0,phi), M(-1,0,-phi), M(1,0,-phi) -- dans le plan y=0
local ico = {
{A1,B1,A3}, {B1,A1,D3}, {D1,C1,C3}, {C1,D1,B3},
{B2,A2,B1}, {A2,B2,C1}, {D2,C2,A1}, {C2,D2,D1},
{B3,A3,A2}, {A3,B3,D2}, {D3,C3,B2}, {C3,D3,C2},
{A1,A3,D2}, {B1,A2,A3}, {A2,C1,B3}, {D1,D2,B3},
{B2,B1,D3}, {A1,C2,D3}, {B2,C3,C1}, {C2,D1,C3} }
g:Dscene3d(
    g:addFacet({A2,B2,C2,D2},{color="Navy",twoside=false,opacity=0.8}),
    g:addFacet({A1,B1,C1,D1},{color="Crimson",twoside=false,opacity=0.8}),
    g:addFacet({A3,B3,C3,D3},{color="Chocolate",twoside=false,opacity=0.8}),
    g:addPolyline(ld.facetedges(ico), {color="Gold",width=12}), -- dessin des arêtes uniquement
    g:addDots({A1,B1,C1,D1,A2,B2,C2,D2,A3,B3,C3,D3}, {color="black",scale=1.2}),
    g:addLabel("A1",A1,{style="W",dist=0.1}, "B1",B1,{style="S"}, "C2",C2,{}, "C3",C3,{},
        "A3",A3,{style="N"}, "D1",D1,{}, "A2",A2,{}, "D2",D2,{}, "B3",B3,{style="E"},
```

```
        "C1",C1,{}, "B2",B2,{}, "D3",D3,{style="W"} )
)
g:Show()
\end{luadraw}
```

FIGURE 33 : Construction d'un icosaèdre



Ajouter des cloisons séparatrices : g:addWall

Les cloisons séparatrices sont des objets 3D qui sont insérés en tout premier dans l'arbre représentant la scène. Ces objets ne sont pas dessinés (donc invisibles), leur rôle est de partitionner l'espace car une facette qui est d'un côté d'une cloison séparatrice ne peut pas être découpée par le plan d'une facette qui est de l'autre côté de la cloison. Cela permet dans certains cas de diminuer significativement le nombre de découpage de facettes (ou lignes polygonales) lors de la construction de la scène. Une cloison séparatrice peut être un plan entier (donc une table de deux points 3D la forme $\{A, n\}$, c'est à dire un point et un vecteur normal), ou bien seulement une facette.

La syntaxe est : **g:addWall(C, options)** où $\langle C \rangle$ est soit un plan, soit une liste de plans, soit une facette, soit une liste de facettes. L'argument $\langle options \rangle$ est une table définissant une seule option qui est :

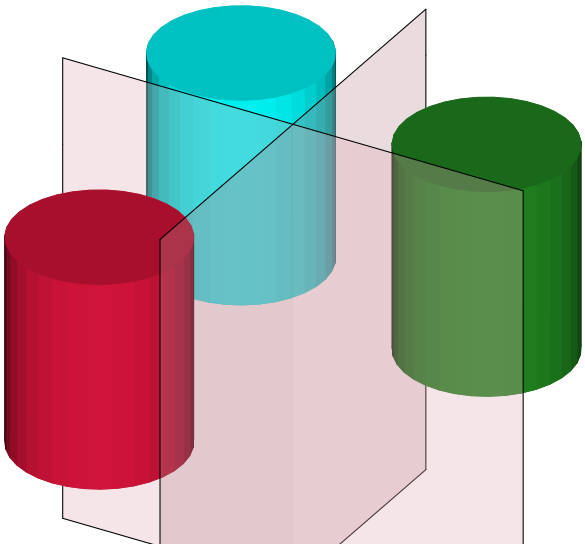
- `matrix=ld.ID3d` : matrice 3D de transformation, par défaut celle-ci est la matrice 3D de l'identité, c'est à dire la table code{M(0,0,0),vecI,vecJ,vecK}.

Dans l'exemple suivant les deux cloisons séparatrices ont été dessinées afin de les visualiser, mais normalement elles sont invisibles :

```
\begin{luadraw}{name=addWall}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{size={10,10},window={-8,8,-4,8}, margin={0,0,0,0}}
local C = ld.cylinder(M(0,0,-1),5*vecK,2)
g:Dscene3d(
    g:addWall( {{Origin,vecI}, {Origin,vecJ}}),
    g:addPlane({Origin,vecI}, {color="Pink",opacity=0.3,scale=1.125,edge=true}), -- premier mur
    g:addPlane({Origin,vecJ}, {color="Pink",opacity=0.3,scale=1.125,edge=true}), -- second mur
    g:addPoly( ld.shift3d(C,M(-3,-3,1)), {color="Cyan"} ),
    g:addPoly( ld.shift3d(C,M(-3,3,0.5)), {color="ForestGreen"} ),
    g:addPoly( ld.shift3d(C,M(3,-3,-0.5)), {color="Crimson"} )
)
g:Show()
\end{luadraw}
```

FIGURE 34 : Exemple avec addWall (les deux facettes transparentes roses sont normalement invisibles)



Remarques sur cet exemple :

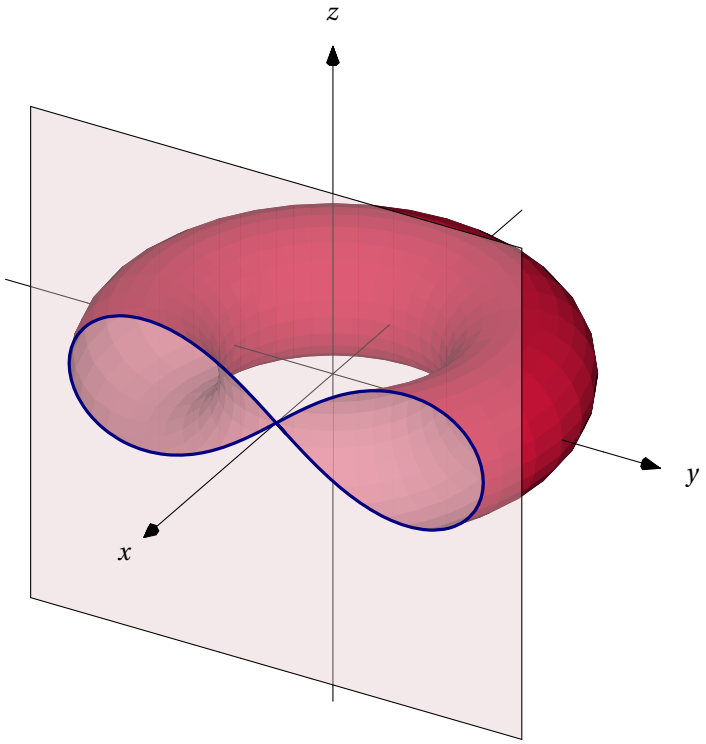
- avec les deux cloisons séparatrices, il n'y a aucune facette découpée, et la scène en contient exactement 111 (37 par cylindre).
- sans les cloisons séparatrices, il y a 117 découpages (inutiles) de facettes, ce qui porte leur nombre à 228 dans la scène.
- avec les deux cloisons séparatrices, et l'option `backcull=true` pour chaque cylindre, il n'y a aucune facette découpée, et la scène en contient 57 seulement.

Voici un autre exemple bien plus probant où l'utilisation de cloisons séparatrices est indispensable pour avoir un dessin de taille raisonnable. Il s'agit de l'obtention d'une lemniscate comme intersection d'un tore avec un certain plan. Le tore étant non convexe le nombre de découpage inutile de facettes peut être très important.

```
\begin{luadraw}{name=torus}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{size={10,10}, margin={0,0,0,0}}
local cos, sin, pi = math.cos, math.sin, math.pi
local R, r = 2.5, 1
local x0 = R-r
local f = function(t) return M(0,R+r*cos(t),r*sin(t)) end
local plan = {M(x0,0,0),-vecI} -- plan dont la section avec le tore donne la lemniscate
local C, wall = ld.rotcurve(f,-pi,pi,{Origin,vecK},360,0,{grid={25,37},addwall=2})
local C1 = ld.cutfacet(C,plan) -- partie du tore dans le demi espace contenant -vecI
g:Dscene3d(
    g:addWall(plan), g:addWall(wall), -- ajout de cloisons séparatrices
    g:addFacet( C1, {color="Crimson", backcull=false}),
    g:addPlane(plan, {color="Pink",opacity=0.4,edge=true}), -- plan de coupe
    g:addAxes( Origin, {arrows=1})
)
-- équation cartésienne du tore : (x^2+y^2+z^2+R^2-r^2)^2-4*R^2*(x^2+y^2) = 0
-- la lemniscate a donc pour équation (x0^2+y^2+z^2+R^2-r^2)^2-4*R^2*(x0^2+y^2)=0 (courbe implicite)
local h = function(y,z) return (x0^2+y^2+z^2+R^2-r^2)^2-4*R^2*(x0^2+y^2) end
local I = ld.implicit(h,-4,4,-3,3,{50,50}) -- ligne polygonale 2D (liste de listes de complexes)
local lemniscate = ld.map(function(z) return M(x0,z.re,z.im) end, I[1]) -- conversion en coordonnées spatiales
g:Dpolyline3d(lemniscate,"Navy,line width=1.2pt")
g:Show()
\end{luadraw}
```

FIGURE 35 : Tore et lemniscate



Remarques sur cet exemple :

- Avec les cloisons séparatrices on a 30 facettes qui sont coupées et un fichier **.tkz* de 140 Ko environ.
- Sans les cloisons séparatrices on a 2068 découpages de facettes (!) et un fichier **.tkz* de 550 Ko environ.
- On aurait pu utiliser la section de coupe qui est renvoyée par la fonction **ld.cutfacet()**, mais le résultat n'est pas très satisfaisant (cela vient du fait que le tore est non convexe).
- Si on n'avait pas voulu les axes traversant le tore et le plan de coupe, on aurait pu faire le dessin avec la méthode **g:Dfacet()**, en remplaçant l'instruction **g:Dscene3d()** par :

```
g:Dfacet(C1, {mode=ld.mShadedOnly,color="Crimson"} )
g:Dfacet( g:Plane2facet(plan,0.75), {color="Pink",opacity=0.4})
```

  On obtient exactement la même chose mais sans les axes (et sans découpage de facettes bien sûr).

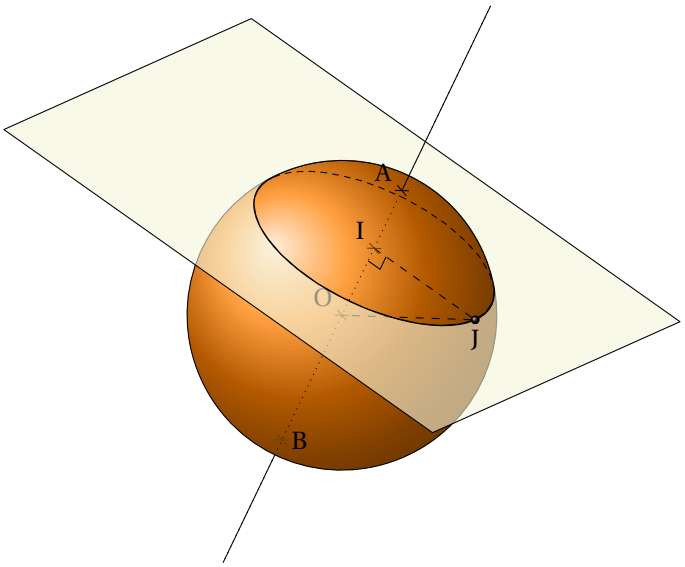
Pour conclure cette partie : on utilise la méthode **g:Dscene3d()** lorsqu'il n'est pas possible de faire autrement, par exemple lorsqu'il y a des intersections (peu nombreuses) qui ne peuvent pas être traitées "à la main". Mais ce n'est pas le cas de toutes les intersections ! Dans l'exemple suivant, on représente une section de sphère par un plan mais sans passer par la méthode **g:Dscene3d()** car celle-ci obligerait à dessiner une sphère à facettes ce qui n'est pas très joli. L'astuce ici, consiste à dessiner la sphère avec la méthode **g:Dsphere()**, puis dessiner par dessus le plan sous forme d'une facette préalablement trouée, le trou correspondant au contour (chemin 3D) de la partie de la sphère située au-dessus du plan :

```
\begin{luadraw}{name=section_sphere}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{ window3d={-4,4,-4,4,-4,4}, window={-5.5,5.5,-4,5}, viewdir={30,75}, size={10,10}}
local O, R = Origin, 2.5 -- center et rayon
local S, P = ld.sphere(O,R), {M(0,0,1.5),vecK+vecJ/2} -- la sphère et le plan de coupe
local w, n = pt3d.normalize(P[2]), g.Normal -- vecteurs unitaires normaux à P pour w et à l'écran pour n
local I, r = ld.interPS(P,{O,R}) -- centre et rayon du petit cercle (intersection entre le plan et la sphère)
local C = g:Intersection3d(S,P) -- C est une liste d'arêtes
local N = I-O
local J = I+r*pt3d.normalize(vecJ-vecK/2) -- un point sur le petit cercle
local a = R/pt3d.abs(N)
local A, B = O+a*N, O-a*N -- points d'intersection de l'axe (O,I) avec la sphère
local c1, alpha = ld.Orange, 0.4
```

```
local coul = {c1[1]*alpha, c1[2]*alpha,c1[3]*alpha} -- pour simuler la transparence
g:Dhline( g:Proj3d({B,-N})) -- demi-droite (le point B est non visible)
g:Dsphere(O,R,{mode=ld.mBorder,color="orange"})
g:Dline3d(A,B,"dotted") -- droite (A,B) en pointillés
g:Dedges(C, {hidden=true,hiddenstyle="dashed"}) -- dessin de l'intersection
g:Dpolyline3d({I,J,O},"dashed")
g:Dangle3d(O,I,J) -- angle droit
g:Dcrossdots3d({{B,N},{I,N},{O,N}},ld.rgb(coul),0.75) -- points dans la sphère
g:Dlabel3d("$O$", O, {pos="NW"})
local L = C.visible[1] -- partie visible de l'intersection (arc de cercle)
A1 = L[1]; A2 = L[#L] -- extrémités de L
local F = g:Plane2facet(P) -- plan converti en facette
-- plan troué sous forme de chemin 3D, le trou est le contour de la partie de la sphère au-dessus du plan
ld.insert(F,{"l","cl",A1,"m",I,A2,r,-1,w,"ca",Origin,A1,R,-1,n,"ca"})
g:Dpath3d( F,"fill=Beige,fill opacity=0.6") -- dessin du plan troué
g:Dhline( g:Proj3d({A,N})) -- demi-droite, partie supérieure de l'axe (AB)
g:Dcrossdots3d({A,N},"black",0.75); g:Dballdots3d(J,"black",0.75)
g:Dlabel3d("$A$", A, {pos="NW"}, "$I$", I, {}, "$B$", B, {pos="E"}, "$J$", J, {pos="S"})
g:Show()
\end{luadraw}
```

FIGURE 36 : Section de sphère sans Dscene3d()



VI Constructions géométriques

Dans cette section sont regroupées les fonctions construisant des figures géométriques sans méthode graphique dédiée.

1) Cercle circonscrit, cercle inscrit : circumcircle3d(), incircle3d()

- La fonction **ld.circumcircle3d(A, B, C)**, où $\langle A\rangle$, $\langle B\rangle$ et $\langle C\rangle$ sont trois points 3D non alignés, renvoie le cercle circonscrit au triangle formé par ces trois points, sous la forme d'une séquence : O, R, n, où O est le centre du cercle, R son rayon, et n un vecteur normal au plan du cercle.
- La fonction **ld.incircle3d(A, B, C)**, où $\langle A\rangle$, $\langle B\rangle$ et $\langle C\rangle$ sont trois points 3D non alignés, renvoie le cercle inscrit dans le triangle formé par ces trois points, sous la forme d'une séquence : I, R, n, où I est le centre du cercle, R son rayon, et n un vecteur normal au plan du cercle.

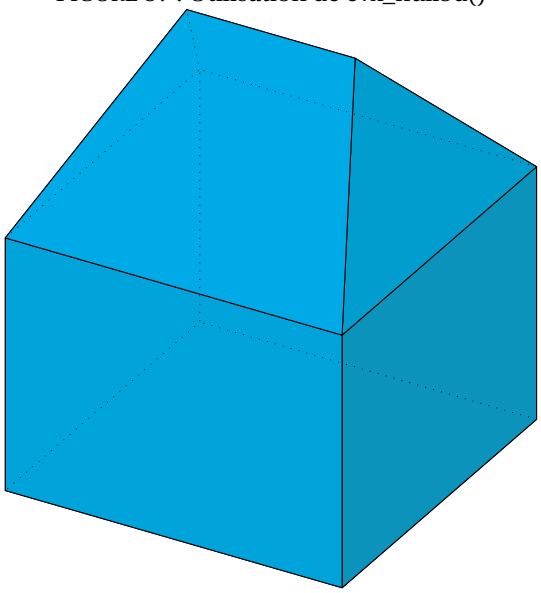
2) Enveloppe convexe : cvx_hull3d()

La fonction **ld.cvx_hull3d(L)** où $\langle L\rangle$ est une liste de points 3D **distincts**, calcule et renvoie l'enveloppe convexe de $\langle L\rangle$ sous la forme d'une liste de facettes.

```
\begin{luadraw}{name=cvx_hull3d}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window={-2,4,-6,1},bbox=false,size={10,10}}
local L = {Origin, 4*vecI, M(4,4,0), 4*vecJ}
ld.insert(L, ld.shift3d(L,-3*vecK))
ld.insert(L, {M(2,1,2), M(2,3,2)})
local V = ld.cvx_hull3d(L)
local P = ld.facet2poly(V)
g:Dpoly(P , {color="cyan",mode=ld.mShadedHidden})
g:Show()
\end{luadraw}
```

FIGURE 37 : Utilisation de cvx_hull3d()



Cas particulier : lorsque tous les points de $\langle L\rangle$ sont dans un même plan, on peut utiliser la fonction **ld.cvx_hull3dcoplanar(L, n)** où $\langle n\rangle$ est un vecteur orthogonal au plan. Cette fonction renvoie une facette (liste de points 3D).

3) Plans : plane(), planeEq(), orthoframe(), plane2ABC()

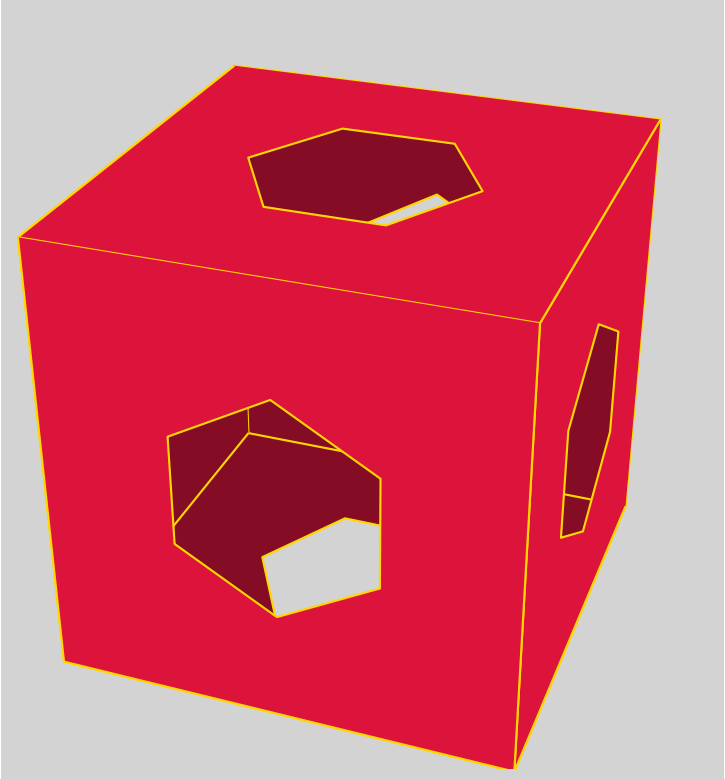
Un plan de l'espace est une table de la forme $\{A, n\}$ où A est un point du plan (point 3D) et n un vecteur normal au plan (point 3D non nul).

- La fonction **ld.plane(A, B, C)** envoie le plan passant par les trois points 3D $\langle A\rangle$, $\langle B\rangle$ et $\langle C\rangle$ (s'ils sont non alignés, sinon le résultat est `nil`).
- La fonction **ld.planeEq(a, b, c, d [, inequality])** renvoie le plan dont une équation cartésienne est $ax+by+cz+d=0$ (si les coefficients $\langle a\rangle$, $\langle a\rangle$ et $\langle a\rangle$ ne sont pas tous nuls, sinon le résultat est `nil`). L'argument optionnel $\langle inequality\rangle$ (`nil` par défaut) peut aussi prendre comme valeur `'<'` ou `'>'`, le vecteur normal au plan est alors choisi de telle sorte qu'il soit dans le demi-espace vérifiant l'inégalité $ax+by+c<0$ (ou $ax+by+c>0$ dans l'autre cas).
- La fonction **ld.plane2ABC(P)** où $\langle P\rangle = \{A, n\}$ désigne un plan, renvoie une séquence de trois points 3D A, B, C, appartenant au plan, et tels que $(A, \vec{AB}, \vec{AC})$ soit un repère orthonormal direct de ce plan.
- La fonction **ld.orthoframe(P [, u])** où $\langle P\rangle = \{A, n\}$ désigne un plan, renvoie une séquence de trois points 3D A, u, v, tels que (A, u, v) soit un repère orthonormal direct de ce plan. L'argument optionnel $\langle u\rangle$, s'il est présent, doit être un vecteur non nul du plan, ce sera le premier vecteur de base une fois normalisé.

```
\begin{luadraw}{name=plans}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M
```

```
local g = ld.graph3d:new{window={-3,3,-3.25,3.25}, margin={0,0,0,0},
    viewdir={"central",20,60}, bg="LightGray", size={10,10}}
ld.Hiddenlines = true; ld.Hiddenlinestyle = "dashed"
local p = ld.polyreg(0,1,6)
local P = ld.parallelep(M(-2,-2,-2),4*vecI,4*vecJ,4*vecK)
local V = g:Sortpolyfacet(P)
local list = {}
g:Filloptions("full","Crimson",1,true); -- true pour le mode evenodd
g:Lineoptions("solid","Gold",8)
for _, F in ipairs(V) do
    local P1 = ld.plane(pt3d.isobar3d(F),F[1],F[2]) -- plan de la facette F
    local A, u, v = ld.orthoframe(P1) -- repère orthonormé sur la facette avec centre de gravité comme origine
    local p1 = ld.map(function(z) return A+z.re*u+z.im*v end,p) -- hexagone reproduit sur la facette
    table.insert(p1,2,"m")
    local color = "Crimson"
    if not g:Isvisible(F) then color = "Crimson!60!black" end
    g:Dpath3d( ld.concat(F,{"l"},p1,{"l","cl"}),"fill="..color ) -- dessin de la facette "trouée" avec l'hexagone
end
g:Show()
\end{luadraw}
```

FIGURE 38 : Faces d'un cube trouées avec un hexagone régulier



4) Sphère circonscrite, Sphère inscrite : circumsphere(), insphere()

- La fonction **ld.circumsphere(A, B, C, D)**, où $\langle A\rangle$, $\langle B\rangle$, $\langle C\rangle$ et $\langle D\rangle$ sont quatre points 3D non coplanaires, renvoie la sphère circonscrite au tétraèdre formé par ces quatre points, sous la forme d'une séquence : I, R, où I est le centre de la sphère, et R son rayon.
- La fonction **ld.insphere(A, B, C, D)**, où $\langle A\rangle$, $\langle B\rangle$, $\langle C\rangle$ et $\langle D\rangle$ sont quatre points 3D non coplanaires, renvoie la sphère inscrite dans le tétraèdre formé par ces quatre points, sous la forme d'une séquence : I, R, où I est le centre de la sphère, et R son rayon.

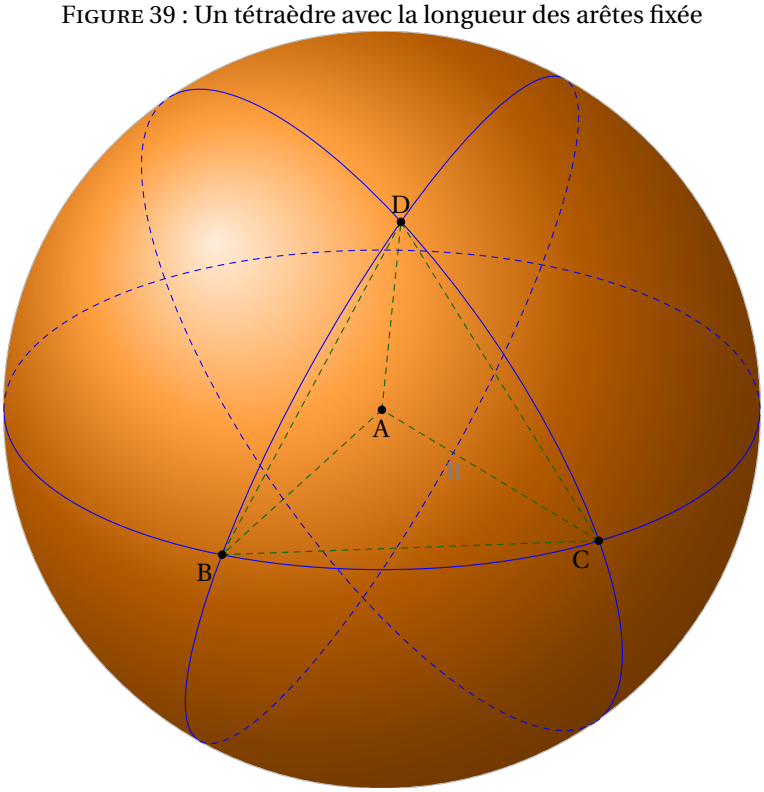
5) Tétraèdre à longueurs fixées : tetra_len()

La fonction **ld.tetra_len(ab, ac, ad, bc, bd, cd)** calcule les sommets A, B, C, D d'un tétraèdre dont les longueurs des arêtes sont données, c'est à dire tels que AB =$\langle ab\rangle$, AC =$\langle ac\rangle$, AD =$\langle ad\rangle$, BC =$\langle bc\rangle$, BD =$\langle bd\rangle$ et CD =$\langle cd\rangle$. La fonction renvoie la séquence de quatre points A, B, C, D. Le sommet A est toujours le point M(0, 0, 0) (`pt3d.Origin`) et le sommet B est toujours

le point `ab*pt3d.vecI` et le sommet C dans le plan xy. Le tétraèdre en tant que polyèdre peut ensuite être construit avec la fonction **ld.tetra(A, B-A, C-A, D-A)**.

```
\begin{luadraw}{name=tetra_len}
local ld = luadraw
local g = ld.graph3d:new{window={-4,4,-4,4},margin={0,0,0,0},viewdir={25,65},size={10,10}}
ld.Hiddenlines = true; ld.Hiddenlinestyle = "dashed"
require 'luadraw_spherical'
local R = 4
local A,B,C,D = ld.tetra_len(R,R,R,R,R,R)
local T = ld.tetra(A,B-A,C-A,D-A)
g:Define_sphere({radius=R})
g:DSpolyline( ld.facetedges(T), {color="DarkGreen"})
g:DSgreatcircle( {B,C},{color="Blue"} )
g:DSgreatcircle( {B,D},{color="Blue"} )
g:DSgreatcircle( {C,D},{color="Blue"}  )
g:DSlabel("$R$",(2*A+C)/3,{pos="S"})
g:Dspherical()
g:Ddots3d({A,B,C,D})
g:Dlabel3d("$A$",A,{pos="S"},"$B$",B,{pos="SW"},"$C$",C,{},"$D$",D,{pos="N"} )
g:Show()
\end{luadraw}
```

FIGURE 39 : Un tétraèdre avec la longueur des arêtes fixée



6) Triangles : sss_triangle3d(), sas_triangle3d(), asa_triangle3d()

Ces fonctions sont la version 3D des fonctions sss_triangle(), sas_triangle(), asa_triangle() déjà décrites.

- La fonction **ld.sss_triangle3d(ab, bc, ca)** où $\langle ab\rangle$, $\langle bc\rangle$ et $\langle ca\rangle$ sont trois longueurs, calcule et renvoie une liste de trois points 3D {A, B, C} formant les sommets d'un triangle direct dans le plan xOy dont les longueurs des côtés sont les arguments, c'est à dire AB $=\langle ab\rangle$, BC $=\langle bc\rangle$ et CA $=\langle ca\rangle$, lorsque cela est possible. Le sommet A est toujours le point M(0, 0, 0) (`pt3d.Origin`) et le sommet B est toujours le point `ab*pt3d.vecI`. Ce triangle peut être dessiné avec la méthode **g:Dpolyline3d()**.
- La fonction **ld.sas_triangle3d(ab, alpha, ca)** où $\langle ab\rangle$ et $\langle ca\rangle$ sont deux longueurs, $\langle alpha\rangle$ un angle en degrés, calcule et renvoie une liste de trois points 3D {A, B, C} formant les sommets d'un triangle dans le plan xOy tel que AB $=\langle ab\rangle$, CA $=\langle ca\rangle$, et tel que l'angle $(\vec{AB}, \vec{AC})$ a pour mesure $\langle alpha\rangle$, lorsque cela est possible. Le sommet A est toujours le point M(0, 0, 0) (`pt3d.Origin`) et le sommet B est toujours le point `ab*pt3d.vecI`. Ce triangle peut être dessiné avec la méthode **g:Dpolyline3d()**.

- La fonction **ld.asa_triangle3d(alpha, ab, beta)** où ⟨*ab*⟩ est une longueur, ⟨*alpha*⟩ et ⟨*beta*⟩ deux angles en degrés, calcule et renvoie une liste de trois points 3D {A, B, C} formant les sommets d'un triangle dans le plan xOy tel que AB =⟨*ab*⟩, tel que l'angle $(\vec{AB}, \vec{AC})$ a pour mesure ⟨*alpha*⟩, et tel que l'angle $(\vec{BA}, \vec{BC})$ a pour mesure ⟨*beta*⟩, lorsque cela est possible. Le sommet A est toujours le point M(0, 0, 0) (`pt3d.Origin`) et le sommet B est toujours le point `ab*pt3d.vecI`. Ce triangle peut être dessiné avec la méthode **g:Dpolyline3d()**.

VII Transformations calcul matriciel et quelques fonctions mathématiques

1) Transformations 3D

Dans les fonctions qui suivent :

- l'argument ⟨*L*⟩ est soit un point 3D, soit un polyèdre, soit une liste de points 3D (facette) soit une liste de listes de points 3D (liste de facettes),
- une droite ⟨*d*⟩ est une liste de deux points 3D {A, *u*} : un point de la droite (A) et un vecteur directeur (*u*),
- un plan ⟨*P*⟩ est une liste de deux points 3D {A, *n*} : un point du plan (A) et un vecteur normal au plan (*n*).

Le résultat renvoyé est de même type que ⟨*L*⟩.

Appliquer une fonction de transformation : ftransform3d

La fonction **ld.ftransform3d(L, f)** renvoie l'image de ⟨*L*⟩ par la fonction ⟨*f*⟩, celle-ci doit être une fonction de $\mathbf{R}^3$ vers $\mathbf{R}^3$ (à un point 3D elle associe un point 3D).

Projections : proj3d, proj3dO, dproj3d

- La fonction **ld.proj3d(L, P)** renvoie l'image de ⟨*L*⟩ par la projection orthogonale sur le plan ⟨*P*⟩.
- La fonction **ld.proj3dO(L, P, v)** renvoie l'image de ⟨*L*⟩ par la projection sur le plan ⟨*P*⟩ parallèlement à la direction du vecteur ⟨*v*⟩ (point 3D non nul).
- La fonction **ld.dproj3d(L, d)** renvoie l'image de ⟨*L*⟩ par la projection sur la droite ⟨*d*⟩.

Projections sur les axes ou les plans liés aux axes

- La fonction **ld.pxy(L [, z0])** renvoie l'mage de ⟨*L*⟩ par la projection orthogonale sur le plan $z =$⟨*z0*⟩ (par défaut $z0 = 0$).
- La fonction **ld.pyz(L [, x0])** renvoie l'mage de ⟨*L*⟩ par la projection orthogonale sur le plan $x =$⟨*x0*⟩ (par défaut $x0 = 0$).
- La fonction **ld.pxz(L [, y0])** renvoie l'mage de ⟨*L*⟩ par la projection orthogonale sur le plan $y =$⟨*y0*⟩ (par défaut $y0 = 0$).
- La fonction **ld.px(L)** renvoie l'mage de ⟨*L*⟩ par la projection orthogonale sur l'axe Ox.
- La fonction **ld.py(L)** renvoie l'mage de ⟨*L*⟩ par la projection orthogonale sur l'axe Oy.
- La fonction **ld.pz(L)** renvoie l'mage de ⟨*L*⟩ par la projection orthogonale sur l'axe Oz.

Symétries : sym3d, sym3dO, dsym3d, psym3d

- La fonction **ld.sym3d(L, P)** renvoie l'image de ⟨*L*⟩ par la symétrie orthogonale par rapport au plan ⟨*P*⟩.
- La fonction **ld.sym3dO(L, P, v)** renvoie l'image de ⟨*L*⟩ par la symétrie par rapport au plan ⟨*P*⟩ et parallèlement à la direction du vecteur ⟨*v*⟩ (point 3D non nul).
- La fonction **ld.dsym3d(L, d)** renvoie l'image de ⟨*L*⟩ par la symétrie orthogonale par rapport la droite ⟨*d*⟩.
- La fonction **ld.psym3d(L, point)** renvoie l'image de ⟨*L*⟩ par la symétrie par rapport à ⟨*point*⟩ (point 3D).

Rotation : rotate3d, rotateaxe3d

- La fonction **ld.rotate3d(L, angle, d)** renvoie l'image de ⟨*L*⟩ par la rotation d'axe ⟨*d*⟩ (orientée par le vecteur directeur qui est *d*[2]), et de ⟨*angle*⟩ degrés.
- La fonction **ld.rotateaxe3d(L, v1, v2 [, center])** renvoie l'image de ⟨*L*⟩ par une rotation d'axe passant par le point 3D ⟨*center*⟩ et qui transforme le vecteur ⟨*v1*⟩ en le vecteur ⟨*v2*⟩, ces vecteurs sont normalisés par la fonction. L'argument ⟨*center*⟩ est facultatif et par défaut c'est le point `pt3d.Origin`.

Homothétie : scale3d

La fonction **ld.scale3d(L, k [, center])** renvoie l'image de ⟨*L*⟩ par l'homothétie de centre le point 3D ⟨*center*⟩, et de rapport ⟨*k*⟩. L'argument ⟨*center*⟩ est facultatif et par défaut c'est le point `pt3d.Origin`.

Inversion : inv3d

La fonction **ld.inv3d(L, radius [, center])** renvoie l'image de ⟨*L*⟩ par l'inversion par rapport à la sphère de centre ⟨*center*⟩, et de rayon ⟨*radius*⟩. L'argument ⟨*center*⟩ est facultatif et par défaut c'est le point `Origin`.

Stéréographie : projstereo et inv_projstereo

Fonction **ld.projstereo(L, S, N, h)** : l'argument ⟨*L*⟩ désigne un point 3D ou une liste de points 3D ou une liste de listes de points 3D, appartenant tous à la sphère ⟨*S*⟩, où ⟨*S*⟩={C, *r*} (C est le centre de la sphère, et *r* le rayon). L'argument ⟨*N*⟩ désigne un point de la sphère qui sera le pôle de la projection. L'argument ⟨*h*⟩ est un réel qui définit le plan de la projection, ce plan est perpendiculaire à l'axe (CN), et passe par le point $\mathrm{I} = \mathrm{C} + h\frac{\vec{\mathrm{CN}}}{\mathrm{CN}}$ (avec $h = 0$ c'est le plan équatorial, avec $h = -r$ c'est le plan tangent à la sphère au pôle opposé). La fonction renvoie l'image de ⟨*L*⟩ par la projection stéréographique par rapport à la sphère ⟨*S*⟩ avec ⟨*N*⟩ comme pôle, et sur le plan {I, N − C}.

Fonction inverse **ld.inv_projstereo(L, S, N)** : ⟨*S*⟩= {C, *r*} est la sphère de centre C et de rayon *r*, ⟨*N*⟩ est un point de la sphère ⟨*S*⟩ (pôle), et ⟨*L*⟩ est un point 3D ou une liste de points 3D ou une liste de listes de points 3D appartenant tous à un même plan orthogonal à l'axe (CN). La fonction renvoie l'image de ⟨*L*⟩ par l'inverse de la projection stéréographique par rapport à ⟨*S*⟩ et de pôle ⟨*N*⟩.

Translation : shift3d

La fonction **ld.shift3d(L, v)** renvoie l'image de ⟨*L*⟩ par la translation de vecteur ⟨*v*⟩ (point 3D).

2) Calcul matriciel

Si f est une application affine de l'espace $\mathbf{R}^3$, on appellera matrice de f la liste (table) :

`{f(pt3d.Origin),Lf(pt3d.vecI),Lf(pt3d.vecJ),Lf(pt3d.vecK)}`

où Lf désigne la partie linéaire de f (on a `Lf(pt3d.vecI)=f(pt3d.vecI)-f(pt3d.Origin)`, etc). La matrice identité est notée `ld.ID3d` dans *luadraw*, elle correspond simplement à la liste `{pt3d.Origin,pt3d.vecI,pt3d.vecJ,pt3d.vecK}`.

applymatrix3d et applyLmatrix3d

- La fonction **ld.applymatrix3d(A, M)** applique la matrice ⟨*M*⟩ au point 3D ⟨*A*⟩ et renvoie le résultat (ce qui revient à calculer f(A) si M est la matrice de f). Si ⟨*A*⟩ n'est pas un point 3D, la fonction renvoie ⟨*A*⟩.
- La fonction **ld.applyLmatrix3d(A, M)** applique la partie linéaire la matrice ⟨*M*⟩ au point 3D ⟨*A*⟩ et renvoie le résultat (ce qui revient à calculer Lf(A) si M est la matrice de f). Si ⟨*A*⟩ n'est pas un point 3D, la fonction renvoie ⟨*A*⟩.

composematrix3d

La fonction **ld.composematrix3d(M1, M2)** effectue le produit matriciel ⟨*M1*⟩×⟨*M2*⟩ et renvoie le résultat.

getmatrix3d

La fonction **ld.getmatrix3d(T1, T2)** calcule et renvoie la matrice qui transforme le tétraèdre ⟨*T1*⟩ en le tétraèdre ⟨*T2*⟩, avec ⟨*T1*⟩= $\{a_1, b_1, c_1, d_1\}$ (liste de quatre points 3D non coplanaires), et ⟨*T2*⟩= $\{a_2, b_2, c_2, d_2\}$ (liste de quatre points 3D).

invmatrix3d

La fonction **ld.invmatrix3d(M)** calcule et renvoie l'inverse de la matrice ⟨*M*⟩ lorsque cela est possible.

matrix3dof

La fonction **ld.matrix3dof(f)** calcule et renvoie la matrice de $\langle f \rangle$, qui doit être une application affine de l'espace $\mathbf{R}^3$ (à un point 3D elle associe un point 3D).

mtransform3d et mLtransform3d

- La fonction **ld.mtransform3d(L, M)** applique la matrice $\langle M \rangle$ à la liste $\langle L \rangle$ et renvoie le résultat. $\langle L \rangle$ doit être une liste de points 3D (une facette) ou une liste de listes de points 3D (liste de facettes).
- La fonction **ld.mLtransform3d(L, M)** applique la partie linéaire la matrice $\langle M \rangle$ à la liste $\langle L \rangle$ et renvoie le résultat. $\langle L \rangle$ doit être une liste de points 3D (une facette) ou une liste de listes de points 3D (liste de facettes).

planetransform

La fonction **ld.planetransform(P, M)** calcule et renvoie l'image du plan $\langle P \rangle$ par la matrice 3D $\langle M \rangle$. Le plan $\langle P \rangle$ est une liste de la forme {A, n} où A est un point du plan et n un vecteur normal au plan, la fonction renvoie une liste du même type.

3) Matrice associée au graphe 3D

Lorsque l'on crée un graphe dans l'environnement *luadraw*, par exemple :

```
local ld = luadraw
local g = ld.graph3d:new{size={10,10}}
```

l'objet g créé possède une matrice 3D de transformation qui est initialement l'identité. Toutes les méthodes graphiques appliquent automatiquement la matrice 3D de transformation du graphe. Pour manipuler cette matrice, on dispose des méthodes qui suivent.

g:Composematrix3d()

La méthode **g:Composematrix3d(M)** multiplie la matrice 3D du graphe g par la matrice $\langle M \rangle$ (avec $\langle M \rangle$ à droite) et le résultat est affecté à la matrice 3D du graphe. L'argument $\langle M \rangle$ doit donc être une matrice 3D.

g:Det3d()

La méthode **g:Det3d()** envoie 1 lorsque la matrice 3D de transformation a un déterminant positif, et -1 dans le cas contraire. Cette information est utile lorsqu'on a besoin de savoir si l'orientation de l'espace a été changée ou non.

g:IDmatrix3d()

La méthode **g:IDmatrix3d()** réaffecte l'identité à la matrice 3D du graphe g.

g:Mtransform3d()

La méthode **g:Mtransform3d(L)** applique la matrice du graphe 3D de g à $\langle L \rangle$ et renvoie le résultat, l'argument $\langle L \rangle$ doit être une liste de points 3D (une facette) ou une liste de listes de points 3D (liste de facettes).

g:MLtransform3d()

La méthode **g:MLtransform3d(L)** applique la partie linéaire de la matrice 3D du graphe g à $\langle L \rangle$ et renvoie le résultat. L'argument $\langle L \rangle$ doit être une liste de points 3D (une facette) ou une liste de listes de points 3D (liste de facettes).

g:Rotate3d()

La méthode **g:Rotate3d(angle, axe)** modifie la matrice 3D de transformation du graphe g en la composant avec la matrice de la rotation d'angle $\langle angle \rangle$ (en degrés) et d'axe $\langle axe \rangle$.

g:Scale3d()

La méthode **g:Scale3d(factor [, center])** modifie la matrice 3D de transformation du graphe *g* en la composant avec la matrice de l'homothétie de rapport ⟨*factor*⟩ et de centre ⟨*center*⟩. L'argument ⟨*center*⟩ est un point 3D qui vaut `pt3d.Origin` par défaut.

g:Setmatrix3d()

La méthode **g:Setmatrix3d(M)** permet d'affecter la matrice ⟨*M*⟩ à la matrice 3D de transformation du graphe *g*.

g:Shift3d()

La méthode **g:Shift3d(v)** modifie la matrice 3D de transformation du graphe *g* en la composant avec la matrice de la translation de vecteur ⟨*v*⟩ qui doit être un point 3D.

4) Fonctions mathématiques supplémentaires

clippolyline3d()

La fonction **ld.clippolyline3d(L, poly [, exterior, close])** clippe la ligne polygonale 3D ⟨*L*⟩ avec le polyèdre **convexe** ⟨*poly*⟩, si l'argument facultatif ⟨*exterior*⟩ vaut `true`, alors c'est la partie extérieure au polyèdre qui est renvoyée (`false` par défaut), si l'argument facultatif ⟨*close*⟩ vaut `true`, alors la ligne polygonale est refermée (`false` par défaut). ⟨*L*⟩ est une liste de points 3D ou une liste de listes de points 3D.

Remarque : le résultat n'est pas toujours satisfaisant pour la partie extérieure.

Cas particulier : clipper une ligne polygonale 3D ⟨*L*⟩ avec la fenêtre 3D courante peut se faire avec cette fonction de la manière suivante :

```
L=ld.clippolyline3d(L,g:Box3d())
```

En effet, la méthode **g:Box3d()** renvoie la fenêtre 3D courante sous forme d'un parallélépipède.

clipline3d()

La fonction **ld.clipline3d(line, poly)** clippe la droite ⟨*line*⟩ avec le polyèdre **convexe** ⟨*poly*⟩, la fonction renvoie la partie de la droite intérieure au polyèdre. L'argument ⟨*line*⟩ est une table de la forme {A, *u*} où A est un point de la droite et *u* un vecteur directeur (deux points 3D).

Cas particulier : clipper une droite ⟨*d*⟩ avec la fenêtre 3D courante peut se faire avec cette fonction de la manière suivante :

```
d=ld.clipline3d(d,g:Box3d())
```

En effet, la méthode **g:Box3d()** renvoie la fenêtre 3D courante sous forme d'un parallélépipède (⟨*d*⟩ devient alors un segment).

cutpolyline3d()

La fonction **ld.cutpolyline3d(L, plane [, close])** coupe la ligne polygonale 3D ⟨*L*⟩ avec le plan ⟨*plane*⟩, si l'argument facultatif *argu* vaut `true`, alors la ligne est refermée (`false` par défaut). ⟨*L*⟩ est une liste de points 3D ou une liste de listes de points 3D, ⟨*plane*⟩ est une table de la forme {A, *n*} où A est un point du plan et *n* un vecteur normal (deux points 3D).

Le fonction renvoie trois choses :

- la partie de ⟨*L*⟩ qui est dans le demi-espace contenant le vecteur *n*,
- suivie de la partie de ⟨*L*⟩ qui est dans l'autre demi-espace,
- suivie de la liste des points d'intersection.

getbounds3d()

La fonction **ld.getbounds3d(L)** renvoie sous forme d'une séquence les limites *xmin, xmax, ymin, ymax, zmin, zmax* de la ligne polygonale 3D ⟨*L*⟩ (liste de points 3D ou une liste de listes de points 3D).

interDP()

La fonction **ld.interDP(d, P)** calcule et renvoie (si elle existe) l'intersection entre la droite ⟨*d*⟩ et le plan ⟨*P*⟩.

interPP()

La fonction **ld.interPP(P1, P2)** calcule et renvoie (si elle existe) l'intersection entre les plans ⟨*P1*⟩ et ⟨*P2*⟩.

interDD()

La fonction **ld.interDD(D1, D2 [, epsilon])** calcule et renvoie (si elle existe) l'intersection entre les droites ⟨*D1*⟩ et ⟨*D1*⟩. L'argument ⟨*epsilon*⟩ vaut 10^{-10} par défaut (sert à tester si un certain flottant est nul).

interCS()

La fonction **ld.interCS(C, S)** calcule et renvoie (si elle existe) l'intersection entre le cercle ⟨*C*⟩= {A, *r*, *n*} (A est le centre du cercle, *r* le rayon et *n* un vecteur normal au plan du cercle), et la sphère ⟨*S*⟩= {B, R} (B est le centre de la sphère et R le rayon). La fonction renvoie soit `nil` (intersection vide), soit un seul point, soit deux points (séquence).

interDS()

La fonction **ld.interDS(d, S)** calcule et renvoie (si elle existe) l'intersection entre la droite ⟨*d*⟩ et la sphère ⟨*S*⟩ qui est une table de la forme {C, *r*} avec C le centre (point 3D) et *r* le rayon de la sphère. La fonction renvoie soit `nil` (intersection vide), soit un seul point, soit deux points.

interPS()

La fonction **ld.interPS(P, S)** calcule et renvoie (si elle existe) l'intersection entre le plan ⟨*P*⟩ et la sphère ⟨*S*⟩= {C, *r*} avec C le centre (point 3D) et *r* le rayon de la sphère. La fonction renvoie soit `nil` (intersection vide), soit une séquence de la forme I, *r*, *n*, où I est un point 3D représentant le centre d'un cercle, *r* son rayon et *n* un vecteur normal au plan du cercle, ce cercle est l'intersection cherchée.

interSS()

La fonction **ld.interSS(S1, S2)** calcule et renvoie (si elle existe) l'intersection entre la sphère ⟨*S1*⟩= {C1, *r*1} et ⟨*S2*⟩= {C2, *r*2}. La fonction renvoie soit `nil` (intersection vide), ou bien une séquence de la forme I, *r*, *n*, où I est un point 3D représentant le centre d'un cercle, *r* son rayon et *n* un vecteur normal au plan du cercle, ce cercle est l'intersection cherchée.

interSSS()

La fonction **ld.interSSS(S1, S2, S3)** calcule et renvoie (si elle existe) l'intersection entre les sphères ⟨*S1*⟩= {C1, *r*1}, ⟨*S2*⟩= {C2, *r*2} et ⟨*S3*⟩= {C3, *r*3}. La fonction renvoie soit `nil` (intersection vide), soit un seul point, soit deux points (séquence).

merge3d()

La fonction **ld.merge3d(L [, epsilon])** recolle si c'est possible, les composantes connexes de ⟨*L*⟩ qui doit être une liste de listes de points 3D, la fonction renvoie le résultat. L'argument ⟨*epsilon*⟩ vaut par défaut 10^{-10}, il est utilisé lors des comparaisons.

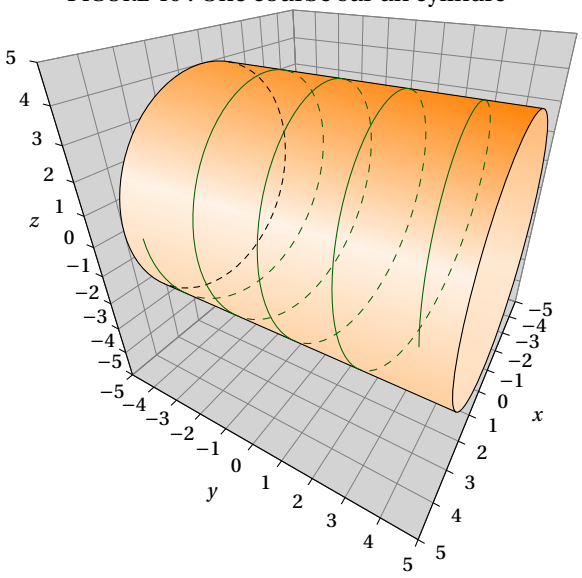
split_points_by_visibility()

La fonction **ld.split_points_by_visibility(L, visible_function)** où ⟨*L*⟩ est une liste de points 3D, ou une liste de listes de points 3D, et où ⟨*visible_function*⟩ est une fonction telle que `visible_function(A)` retourne `true` si le point 3D A est visible, `false` sinon, permet de trier les points de ⟨*L*⟩ suivant qu'ils sont visibles ou non. La fonction renvoie une séquence de deux tables : *visible_points*, *hidden_points*.

```
\begin{luadraw}{name=curve_on_cylinder}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M, Mc = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M, pt3d.Mc

local g = ld.graph3d:new{adjust2d=true,bbox=false,size={10,10}, viewdir="central"}
g:Labelsize("footnotesize")
ld.Hiddenlines = true; ld.Hiddenlinestyle = "dashed"
local curve_on_cylinder = function(curve,cylinder)
-- curve is a 3D polyline on a cylinder,
-- cylinder = {A,r,V,B}
    local A,r,V,B = table.unpack(cylinder)
    if B == nil then B = V; V = B-A end
    local U = B-A
    local visible_function
    if ld.projection_mode == "central" then
        visible_function = function(N)
            local I = ld.dproj3d(N,{A,U})
            local M1, M2 = ld.interCS({I,r,U},{ (I+ld.camera)/2, pt3d.abs(I-ld.camera)/2})
            local alpha = pt3d.angle3d(M1-I,ld.camera-I)
            return pt3d.angle3d(N-I,ld.camera-I) <= alpha
        end
    else
        visible_function = function(N)
            local I = ld.dproj3d(N,{A,U})
            return (pt3d.dot(N-I,g.Normal) >= 0)
        end
    end
    return ld.split_points_by_visibility(curve,visible_function)
end
-- test
local A, r, B = -5*vecJ, 4, 5*vecJ -- cylinder
local p = function(t) return Mc(r,t,t/3) end
local Curve = ld.rotate3d( ld.parametric3d(p,-4*math.pi,4*math.pi),90,{Origin,vecI})
local Vi, Hi = curve_on_cylinder(Curve,{A,r,B})
local curve_color = "DarkGreen"
g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
g:Dcylinder(A,r,B,{color="orange"})
g:Dpolyline3d(Vi,curve_color)
g:Dpolyline3d(Hi,curve_color..","..ld.Hiddenlinestyle)
g:Show()
\end{luadraw}
```

FIGURE 40 : Une courbe sur un cylindre



VIII Exemples plus poussés

1) La boîte de sucre

Le problème[7] est de dessiner des sucres dans une boîte. Il faut pouvoir positionner le nombre que l'on veut de morceaux, et où on veut dans la boite[8] sans avoir à réécrire tout le code. Autre contrainte : pour alléger au maximum la figure, seules les facettes réellement vues doivent être affichées. Dans le code proposé ci-dessous on garde les angles de vues par défaut, et :

- les sucres sont des cubes de côté 1 (on modifie ensuite la matrice 3D du graphe pour les "allonger"),
- chaque morceau est repéré par les coordonnées (x, y, z) du coin supérieur droit de la face avant, avec x entier entre 1 et Lg, y entier entre 1 et lg et z entier entre 1 et ht.
- pour mémoriser les positions des morceaux on utilise une matrice *positions* à trois dimensions, une pour x, une pour y et une pour z, avec la convention que *positions[x][y][z]* vaut 1 s'il y a un sucre à la position (x, y, z), et 0 sinon.
- pour chaque morceau il y a au plus trois faces visibles : celles du dessus, celle de droite et celle de devant[9], mais on ne dessine la face du dessus que s'il n'y a pas un autre morceau de sucre au-dessus, on ne dessine la face du droite que s'il n'y a pas un autre morceau à droite, et on ne dessine la face de devant que s'il n'y a pas un autre morceau devant. On construit ainsi la liste des facettes réellement vues.
- Dans l'affichage de la scène, il faut **mettre la boîte en premier**, sinon les facettes de celle-ci vont être découpées par les plans des facettes des morceaux de sucre. Les facettes des morceaux de sucre ne peuvent pas être découpées par la boîte car ils sont tous dedans.

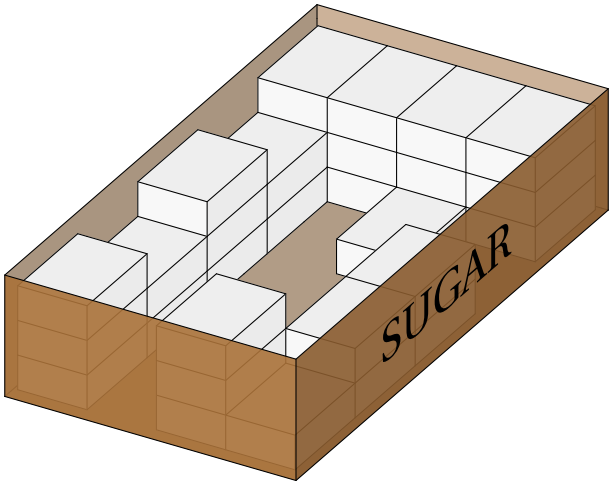
```
\begin{luadraw}{name=boite_sucres}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M, Mc = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M, pt3d.Mc

local g = ld.graph3d:new{window={-9,8,-10,4},size={10,10}}
ld.Hiddenlines = false
local Lg, lg, ht = 5, 4, 3 -- longueur, largeur, hauteur (taille de la boîte)
local positions = {} -- matrice de dimension 3 initialisée avec des 0
for L = 1, Lg do
    local X = {}
    for l = 1, lg do
        local Y = {}
        for h = 1, ht do table.insert(Y,0) end
        table.insert(X,Y)
    end
    table.insert(positions,X)
end
local facetList = function() -- renvoie la liste des facettes à dessiner (attention à l'orientation)
    local facet = {}
    for x = 1, Lg do -- parcours de la matrice positions
        for y = 1, lg do
            for z = 1, ht do
                if positions[x][y][z] == 1 then -- il y a un sucre en (x,y,z)
                    if (z == ht) or (positions[x][y][z+1] == 0) then -- pas de sucre au-dessus, face du dessus visible
                        table.insert(facet, {M(x,y,z),M(x-1,y,z),M(x-1,y-1,z),M(x,y-1,z)}) -- insertion face du dessus
                    end
                    if (y == lg) or (positions[x][y+1][z] == 0) then -- pas de sucre à droite, face de droite visible
                        table.insert(facet, {M(x,y,z),M(x,y,z-1),M(x-1,y,z-1),M(x-1,y,z)}) -- insertion face de droite
                    end
                    if (x == Lg) or (positions[x+1][y][z] == 0) then -- pas de sucre devant donc face de devant visible
                        table.insert(facet, {M(x,y,z),M(x,y-1,z),M(x,y-1,z-1),M(x,y,z-1)}) -- insertion face de devant
                    end
                end
            end
        end
    end
    return facet
```

7. Problème posé dans un forum, l'objectif étant d'en faire des exercices de comptage pour des élèves.
8. Un morceau doit reposer soit sur le fond de la boîte, soit sur un autre morceau
9. À condition de ne pas changer les angles de vue!

```
end
-- création de la boîte (parallélépipède)
local O = Origin -0.1*M(1,1,1) -- pour ne pas que la boîte soit collée aux sucres
local boite = ld.parallelep(O, (Lg+0.2)*vecI, (lg+0.2)*vecJ, (ht+0.5)*vecK)
table.remove(boite.facets,2) -- on retire le dessus de la boîte, c'est la facette numéro 2
-- on positionne des sucres
for y = 1, 4 do for z = 1, 3 do  positions[1][y][z] = 1 end end
for x = 2, 5 do for z = 1, 2 do positions[x][1][z] = 1 end end
for z = 1, 3 do positions[5][3][z] = 1 end
for z = 1, 2 do positions[4][4][z] = 1 end
for z = 1, 2 do positions[3][4][z] = 1 end
positions[5][1][3] = 1; positions[3][1][3] = 1; positions[5][4][1] = 1; positions[2][3][1] = 1
g:Setmatrix3d({Origin,3*vecI,2*vecJ,vecK}) -- dilatation sur Ox et Oy pour "allonger" les cubes ...
g:Dscene3d( -- dessin
    g:addPoly(boite,{color="brown",edge=true,opacity=0.9}),
    g:addFacet(facetList(), {backcull=true,contrast=0.25,edge=true})    )
g:Labelsize("huge"); g:Dlabel3d( "SUGAR", M(Lg/2+0.1,lg+0.1,ht/2+0.1), {dir={-vecI,vecK}})
g:Show()
\end{luadraw}
```

FIGURE 41 : Boite de morceaux de sucre



2) Empilement de cubes

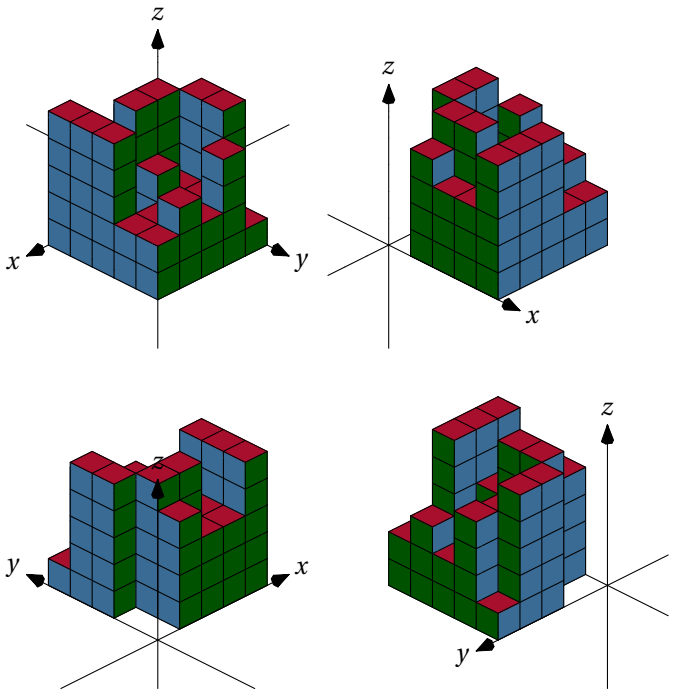
On peut modifier l'exemple précédent pour dessiner un empilement de cubes positionnés au hasard, avec 4 vues. On va positionner les cubes en en mettant un nombre aléatoire par colonne en commençant par le bas. On va faire 4 vues de l'empilement en ajoutant les axes pour se repérer entre ces différentes vues. Cela change un peu la recherche des facettes potentiellement visibles, il y a 5 cas par cube et non plus seulement 3 (devant, derrière, gauche, droite et dessus, on ne fait pas de vues de dessous). Pour plus de lisibilité de l'empilement, on utilise trois couleurs pour peindre les faces des cubes (deux faces opposées ont la même couleur).

```
\begin{luadraw}{name=cubes_empiles}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window3d={-6,6,-6,6,-6,6},size={10,10}}
ld.Hiddenlines = false
local Lg, lg, ht, a = 5, 5, 5, 2 -- longueur, largeur, hauteur de l'espace à remplir, taille d'un cube
local positions = {} -- matrice de dimension 3 initialisée avec des 0
for L = 1, Lg do
    local X = {}
    for l = 1, lg do
```

```
        local Y = {}
        for h = 1, ht do table.insert(Y,0) end
        table.insert(X,Y)
    end
    table.insert(positions,X)
end
for x = 1, Lg do  -- positionnement aléatoire de cubes
    for y = 1, lg do
        local nb = math.random(0,ht) -- on met nb cubes dans la colonne (x,y,*) en partant du bas
        for z = 1, nb do positions[x][y][z] = 1 end
    end
end
local dessus,gauche,devant = {},{},{} -- pour mémoriser les facettes
for x = 1, Lg do -- parcours de la matrice positions pour déterminer les facettes à dessiner
    for y = 1, lg do
        for z = 1, ht do
            if positions[x][y][z] == 1 then -- il y a un cube en (x,y,z)
                if (z == ht) or (positions[x][y][z+1] == 0) then -- pas de cube au-dessus donc face visible
                    table.insert(dessus,{M(x,y,z),M(x-1,y,z),M(x-1,y-1,z),M(x,y-1,z)}) -- insertion face du dessus
                end
                if (y == lg) or (positions[x][y+1][z] == 0) then -- pas de cube à droite donc face  visible
                    table.insert(gauche,{M(x,y,z),M(x,y,z-1),M(x-1,y,z-1),M(x-1,y,z)}) -- insertion face droite
                end
                if (y == 1) or (positions[x][y-1][z] == 0) then -- pas de cube à gauche donc face visible
                    table.insert(gauche,{M(x,y-1,z),M(x-1,y-1,z),M(x-1,y-1,z-1),M(x,y-1,z-1)}) -- insertion face gauche
                end
                if (x == Lg) or (positions[x+1][y][z] == 0) then -- pas de cube devant donc face visible
                    table.insert(devant,{M(x,y,z),M(x,y-1,z),M(x,y-1,z-1),M(x,y,z-1)}) -- insertion face avant
                end
                if (x == 1) or (positions[x-1][y][z] == 0) then -- pas de cube derrière donc face de derrière visible
                    table.insert(devant,{M(x-1,y,z),M(x-1,y,z-1),M(x-1,y-1,z-1),M(x-1,y-1,z)}) -- face arrière
                end
            end
        end
    end
end
g:Setmatrix3d({M(-a*Lg/2,-a*lg/2,-a*ht/2),a*vecI,a*vecJ,a*vecK}) -- pour centrer la figure et avoir des cubes de côté a
local dessin = function()
    g:Dscene3d(
        g:addFacet(dessus, {backcull=true,color="Crimson"}), g:addFacet(gauche, {backcull=true,color="DarkGreen"}),
        g:addFacet(devant, {backcull=true,color="SteelBlue"}),
        g:addPolyline(ld.facetedges(ld.concat(dessus,gauche,devant))), -- dessin des arêtes
        g:addAxes(Origin,{arrows=1}))
end
g:Saveattr(); g:Viewport(-5,0,0,5); g:Coordsystem(-11,11,-11,11); g:Setviewdir(45,60) -- en haut à gauche
 dessin(); g:Restoreattr()
g:Saveattr(); g:Viewport(0,5,0,5);g:Coordsystem(-11,11,-11,11); g:Setviewdir(-45,60) -- en haut à droite
dessin(); g:Restoreattr()
g:Saveattr(); g:Viewport(-5,0,-5,0);g:Coordsystem(-11,11,-11,11); g:Setviewdir(-135,60) -- en bas à gauche
dessin(); g:Restoreattr()
g:Saveattr(); g:Viewport(0,5,-5,0);g:Coordsystem(-11,11,-11,11); g:Setviewdir(135,60) -- en bas à droite
dessin(); g:Restoreattr()
g:Show()
\end{luadraw}
```

FIGURE 42 : Empilement de cubes



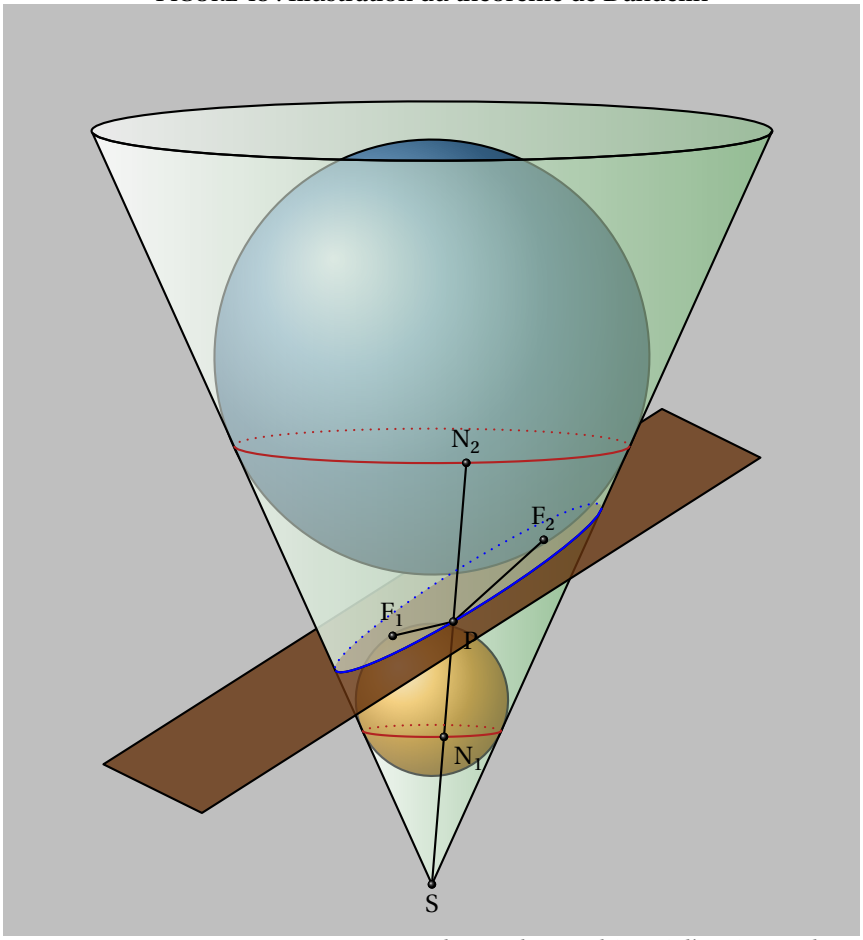
3) Illustration du théorème de Dandelin

```
\begin{luadraw}{name=Dandelin}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window3d={-5,5,-5,5,-5,5}, window={-5,5,-5,6}, bg="lightgray",viewdir={-10,85}}
g:Linewidth(8)
local sqrt = math.sqrt
local sqr = function(x) return x*x end
local L, a = 4.5, 2
local R = (a+5)*L/sqrt(100+L^2) --grosse sphère centre=M(0,0,a) rayon=R
local S2 = ld.sphere(M(0,0,a),R,45,45)
local k = 0.35 --rapport d'homothetie
local b, r = (a+5)*k-5, k*R -- petite sphère centre=M(0,0,b) rayon=r
local S1 = ld.sphere(M(0,0,b),r,45,45)
local c = (b+k*a)/(1+k) --deuxieme centre d'homothetie
local z = a+sqr(R)/(c-a) --image de c par l'inversion par rapport à la grosse sphère
local M1 = M(0,sqrt(sqr(R)-sqr(z-a)),z)--point de la grosse sphère et du plan tangent
local N = M1-M(0,0,a) -- vecteur normal au plan tangent
local plan = {M(0,0,c),-N} -- plan tangent
local z2 = a+sqr(R)/(-5-a) --image du sommet par l'inversion par rapport à la grosse sphère
local z1 = b+sqr(r)/(-5-b) -- image du sommet par l'inversion par rapport à la petite sphère
local P2 = M(sqrt(R^2-(z2-a)^2),0,z2)
local P1= M(sqrt(r^2-(z1-b)^2),0,z1)
local S = M(0,0,-5)
local P = ld.interDP({P1,P2-P1},plan)
local C = ld.cone(M(0,0,-5),10*vecK,L,45,true)
local ellips = g:Intersection3d(C,plan)
local plan1 = {M(0,0,z1),vecK}
local plan2 = {M(0,0,z2),vecK}
local L1, L2 = g:Intersection3d(S1,plan1), g:Intersection3d(S2,plan2)
local F1, F2 = ld.proj3d(M(0,0,b), plan), ld.proj3d(M(0,0,a), plan) --foyers
local s1, s2 = g:Proj3d(M(0,0,a)), g:Proj3d(M(0,0,b))
local V, H = g:Classifyfacet(C) -- on sépare facettes visibles et les autres
local V1, V2 = ld.cutfacet(V,plan)
local H1, H2 = ld.cutfacet(H,plan)
```

```
-- Dessin
-- faces non visibles sous le plan, remplissage seulement
g:Dpolyline3d( ld.border(H2),"left color=white, right color=DarkSeaGreen, draw=none" )
g:Dsphere( M(0,0,b), r, {mode=ld.mBorder,color="Orange"}) -- petite sphère
-- faces visibles sous le plan
g:Dpolyline3d( ld.border(V2),"left color=white, right color=DarkSeaGreen, fill opacity=0.4" )
g:Dpolyline3d({S,P}) -- segment [S,P] qui est sous le plan en partie
g:Dfacet( g:Plane2facet(plan,0.75), {color="Chocolate", opacity=0.8}) -- le plan
-- contour faces non visibles au dessus du plan, remplissage seulement
g:Dpolyline3d( ld.border(H1),"left color=white, right color=DarkSeaGreen,draw=none,fill opacity=0.7" )
g:Dsphere( M(0,0,a),R, {mode=ld.mBorder,color="SteelBlue"}) -- grosse sphère
-- contour faces visibles au dessus du plan
g:Dpolyline3d( ld.border(V1),"left color=white, right color=DarkSeaGreen, fill opacity=0.6" )
g:Dcircle3d(M(0,0,5),L,vecK) -- ouverture du cône
g:Dpolyline3d({{P,F1},{F2,P,P2}})
g:Dedges(L1,{hidden=true,color="FireBrick"})
g:Dedges(L2,{hidden=true,color="FireBrick"})
g:Dedges(ellips,{hidden=true, color="blue"})
g:Dballdots3d({F1,F2,S,P1,P,P2},nil,0.75)
g:Dlabel3d("$F_1$",F1,{pos="N"}, "$F_2$",F2,{}, "$N_2$",P2,{},"$S$",S,{pos="S"},
"$N_1$",P1,{pos="SE"}, "$P$",P,{pos="SE"} )
g:Show()
\end{luadraw}
```

FIGURE 43 : Illustration du théorème de Dandelin



On veut dessiner un cône avec une section par un plan et deux sphères à l'intérieur de ce cône (et tangentes au plan), mais sans dessiner de sphères ni de cônes à facettes. Le point de départ est néanmoins la création de ces solides à facettes, les sphères *S1* et *S2* (lignes 11 et 8 du listing) ainsi que le cône *C* en ligne 23. Le principe du dessin est le suivant :

1. On sépare les facettes du cône en deux catégories : les facettes visibles (tournées vers l'observateur) et les autres (variables *V* et *H* ligne 30), ce qui correspond en fait à l'avant du cône et l'arrière du cône.
2. On découpe les deux listes de facettes avec le plan (lignes 31 et 32). Ainsi, *V1* correspond aux facettes avant situées au-dessus du plan et *V2* correspond aux facettes avant situées sous le plan (même chose avec *H1* et *H2* pour l'arrière).
3. On dessine alors le contour de *H2* avec un remplissage (seulement) en gradient (ligne 34).

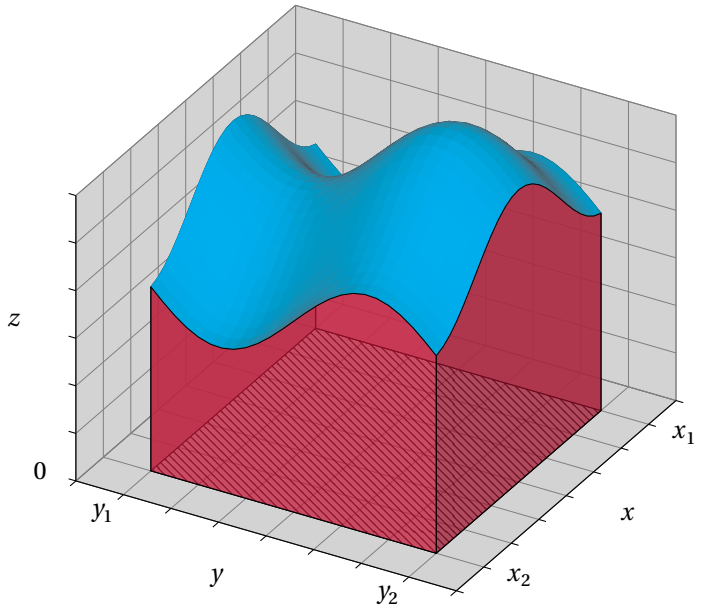
4. On dessine la petite sphère (en orange, ligne 35).
5. On dessine le contour de *V2* avec un remplissage en gradient et transparence pour voir la petite sphère (ligne 36).
6. On dessine le segment [S, P] (ligne 37) puis le plan sous forme de facette transparente (ligne 38).
7. On dessine le contour de *H1* avec un remplissage en gradient (ligne 39). C'est la partie arrière au dessus du plan.
8. On dessine la grande sphère (ligne 40).
9. On dessine enfin le contour de *V1* avec un remplissage en gradient (ligne 41) et transparence pour voir la sphère (c'est la partie avant du cône au dessus du plan), puis l'ouverture du cône (ligne 42).
10. On dessine les intersections entre le cône et les sphères (lignes 44 et 45) ainsi qu'entre le cône et le plan (ligne 46).

4) Volume défini par une intégrale double

```
\begin{luadraw}{name=volume_integrale}
local ld = luadraw
local M = ld.pt3d.M
local pi, sin, cos = math.pi, math.sin, math.cos

local g = ld.graph3d:new{window3d={-4,4,-4,4,0,6},adjust2d=true,margin={0,0,0,0},size={10,10}}
local x1, x2, y1, y2 = -3,3,-3,3 -- bornes
local f = function(x,y) return cos(x)+sin(y)+5 end -- fonction à intégrer
local p = function(u,v) return M(u,v,f(u,v)) end -- paramétrage surface z=f(x,y)
local Fx1 = ld.concat({ld.pxy(p(x1,y2)), ld.pxy(p(x1,y1))}, ld.parametric3d(function(t) return p(x1,t)
 ↪  end,y1,y2,25,false,0)[1])
local Fx2 = ld.concat({ld.pxy(p(x2,y1)), ld.pxy(p(x2,y2))}, ld.parametric3d(function(t) return p(x2,t)
 ↪  end,y2,y1,25,false,0)[1])
local Fy1 = ld.concat({ld.pxy(p(x1,y1)), ld.pxy(p(x2,y1))}, ld.parametric3d(function(t) return p(t,y1)
 ↪  end,x2,x1,25,false,0)[1])
local Fy2 = ld.concat({ld.pxy(p(x2,y2)), ld.pxy(p(x1,y2))}, ld.parametric3d(function(t) return p(t,y2)
 ↪  end,x1,x2,25,false,0)[1])
g:Dboxaxes3d({grid=true, gridcolor="gray",fillcolor="LightGray",labels=false})
g:Filloptions("fdiag","black"); g:Dpolyline3d( {M(x1,y1,0),M(x1,y2,0),M(x2,y2,0),M(x2,y1,0)}) -- dessous
g:Dfacet( {Fx1,Fy1},{mode=ld.mShaded,opacity=0.7,color="Crimson"} )
g:Dfacet(ld.surface(p,x1,x2,y1,y2), {mode=ld.mShadedOnly,color="cyan"})
g:Dfacet( {Fx2,Fy2},{mode=ld.mShaded,opacity=0.7,color="Crimson"} )
g:Dlabel3d("$x_1$", M(x1,4.75,0),{}, "$x_2$", M(x2,4.75,0),{},
"$y_1$", M(4.75,y1,0),{}, "$y_2$", M(4.75,y2,0),{}, "$0$",M(4,-4.75,0),{})
g:Show()
\end{luadraw}
```

FIGURE 44 : Volume correspondant à $\int_{x_1}^{x_2}\int_{y_1}^{y_2} f(x,y)dxdy$



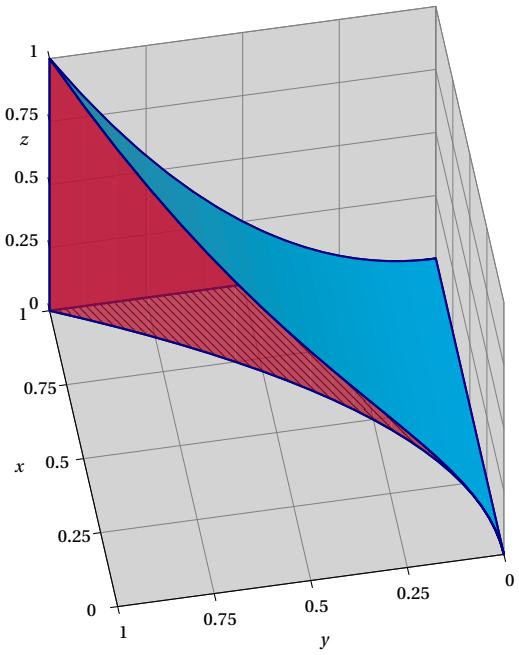
Ici le solide représenté a des faces latérales (*Fx1*, *Fx2*, *Fy1* et *Fy2*) présentant un côté qui est une courbe paramétrée. On prend donc les points de cette courbe paramétrée (sa première composante connexe) et on lui ajoute les projetés des deux extrémités sur le plan xOy. Il faut faire attention au sens de parcours pour que les faces soient bien orientées (normale vers l'extérieur), cette normale étant calculée à partir des trois premiers points de la face, il vaut mieux commencer la face par les deux projetés sur le plan pour être sur de l'orientation. On dessine en premier le dessous, puis les faces latérales, et on termine par la surface.

5) Volume défini sur autre chose qu'un pavé

```
\begin{luadraw}{name=volume2}
local ld = luadraw
local M = ld.pt3d.M
local pi, sin, cos = math.pi, math.sin, math.cos

local g = ld.graph3d:new{window3d={0,1,0,1,0,1}, margin={0,0,0,0},adjust2d=true,viewdir={170,40}, size={10,10}}
g:Labelsize("scriptsize")
local f = function(t) return M(t,t^2,0) end
local h = function(t) return M(1,t,t^2) end
local C = ld.parametric3d(f,0,1,25,false,0)[1] -- courbe y=x^2 dans le plan z=0 (première composante connexe)
local D = ld.parametric3d(h,1,0,25,false,0)[1] -- courbe z=y^2 dans le plan x=1, en sens inverse
local dessous = ld.concat({M(1,0,0)},C) -- forme la face du dessous
local arriere = ld.concat({M(1,1,0)},D) -- forme la face arrière
local  avant, dessus, A, B = {}, {}, nil, C[1]
for k = 2, #C do --on construit les faces avant et de dessus facette par facette, en partant des points de C
    A = B; B = C[k]
    table.insert(avant, {B,A,M(A.x,A.y,A.y^2),M(B.x,B.y,B.y^2)})
    table.insert(dessus, {M(B.x,B.y,B.y^2),M(A.x,A.y,A.y^2),M(1,A.y,A.y^2),M(1,B.y,B.y^2)})
end
g:Dboxaxes3d({grid=true, gridcolor="gray",fillcolor="LightGray", drawbox=false,
    xyzstep=0.25, zlabelstyle="W",zlabelsep=0})
g:Lineoptions(nil,"Navy",8)
g:Dpolyline3d(arriere,true,"fill=Crimson, fill opacity=0.6") -- face arrière (plane)
g:Filloptions("fdiag","black"); g:Dpolyline3d(dessous,true) -- dessous
g:Dmixfacet(avant,{color="Crimson",opacity=0.7,mode=ld.mShadedOnly}, dessus,{color="cyan",opacity=1})
g:Filloptions("none"); g:Dpolyline3d(ld.concat(ld.border(avant),ld.border(dessus)))
g:Show()
\end{luadraw}
```

FIGURE 45 : Volume : $0 \leqslant x \leqslant 1$; $0 \leqslant y \leqslant x^2$; $0 \leqslant z \leqslant y^2$



Dans cet exemple, la surface a pour équation $z = y^2$ (cylindre parabolique), mais nous ne sommes plus sur un pavé. La face avant n'est pas plane, on construit celle-ci à la manière d'un cylindre (ligne 14) avec des facettes verticales qui s'appuient sur la courbe C en bas, et sur la courbe $t \mapsto \mathrm{M}(t, t^2, t^4)$ en haut.

De même, la face du dessus (la surface) est construite à la manière d'un cylindre horizontal qui s'appuie sur les courbes D et $t \mapsto \mathrm{M}(t, t^2, t^4)$.

On pourrait ne pas construire à la main la surface (appelée *dessus* dans le code), et dessiner à la place la surface suivante (après la face avant) :

```
g:Dfacet( ld.surface(function(u,v) return M(u,v*u^2,v^2*u^4) end, 0,1,0,1), {mode=ld.mShadedOnly, color="cyan"})
```

mais elle comporte bien plus de facettes (25×25) que la construction sous forme de cylindre (21 facettes), ce qui est moins intéressant.

Chapitre 3

Annexes

I Extensions

1) Le module *luadraw_polyhedrons*

Ce module est encore à l'état d'ébauche et est appelé à s'étoffer par la suite. Comme son nom l'indique, il contient la définition de polyèdres. Toutes les données numériques sont issues du site Visual Polyhedra.

Utilisation : ce module n'ajoute pas de nouvelles méthodes graphiques, mais il renvoie une table de fonctions construisant des polyèdres. Par conséquent, il s'utilise ainsi (exemple) :

```
local poly = require 'luadraw_polyhedrons'
local M = pt3d.M
local T = poly.tetrahedron( M(0,0,0), M(1,1,1) )
```

Toutes les fonctions sont sur le même modèle : **<nom>(C, S [, all])** où ⟨*C*⟩ est le centre du polyèdre (point 3D) et ⟨*S*⟩ un sommet du polyèdre (point 3D), lorsque ⟨*C*⟩ ou ⟨*S*⟩ ont la valeur `nil`, c'est le polyèdre non transformé (de centre l'origine) qui est renvoyé. L'argument facultatif ⟨*all*⟩ est un booléen, lorsqu'il a la valeur `true` la fonction renvoie une séquence de quatre choses : *P, V, E, F* où :

- P est le solide en tant que polyèdre,
- V la liste (table) des sommets,
- E la liste (table) des arêtes (avec points 3D),
- F la liste des facettes (avec points 3D). Certains polyèdres ont plusieurs types de facettes, dans ce cas la résultat renvoyé est de la forme : *P, V, E, F1, F2, ...*, où F1, F2, ..., sont des listes de facettes. Cela peut permettre de les dessiner avec des couleurs différentes par exemple.

Lorsque l'argument ⟨*all*⟩ a la valeur `false`, qui est la valeur par défaut, la fonction ne renvoie que le polyèdre.

Voici les solides actuellement contenus dans ce module :

- Les solides de Platon, ces solides n'ont qu'un type des faces :
  - la fonction **tetrahedron(C, S [, all])** permet la construction d'un tétraèdre régulier de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D).
  - La fonction **octahedron(C, S [, all])** permet la construction d'un octaèdre de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D).
  - La fonction **cube(C, S [, all])** permet la construction d'un cube de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D).
  - La fonction **icosahedron(C, S [, all])** permet la construction d'un icosaèdre de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D).
  - La fonction **dodecahedron(C, S [, all])** permet la construction d'un dodécaèdre de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D).
- Les solides d'Archimède :
  - La fonction **cuboctahedron(C, S [, all])** permet la construction d'un cuboctaèdre de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.
  - La fonction **icosidodecahedron(C, S [, all])** permet la construction d'un icosidodécaèdre de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.

- La fonction **lsnubcube(C, S [, all])** permet la construction d'un cube adouci (forme 1) de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.
- La fonction **lsnubdodecahedron(C, S [, all])** permet la construction d'un dodécaèdre adouci (forme 1) de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.
- La fonction **rhombicosidodecahedron(C, S [, all])** permet la construction d'un rhombicosidodécaèdre de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a trois types de faces.
- La fonction **rhombicuboctahedron(C, S [, all])** permet la construction d'un rhombicuboctaèdre de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.
- La fonction **rsnubcube(C, S [, all])** permet la construction d'un cube adouci (forme 2) de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.
- La fonction **rsnubdodecahedron(C, S [, all])** permet la construction d'un dodécaèdre adouci (forme 2) de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.
- La fonction **truncatedcube(C, S [, all])** permet la construction d'un cube tronqué de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.
- La fonction **truncatedcuboctahedron(C, S [, all])** permet la construction d'un cuboctaèdre tronqué de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a trois types de faces.
- La fonction **truncateddodecahedron(C, S [, all])** permet la construction d'un dodécaèdre tronqué de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.
- La fonction **truncatedicosahedron(C, S [, all])** permet la construction d'un icosaèdre tronqué de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.
- La fonction **truncatedicosidodecahedron(C, S [, all])** permet la construction d'un icosidodécaèdre tronqué de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a trois types de faces.
- La fonction **truncatedoctahedron(C, S [, all])** permet la construction d'un octaèdre tronqué de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.
- La fonction **truncatedtetrahedron(C, S [, all])** permet la construction d'un tétraèdre tronqué de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.

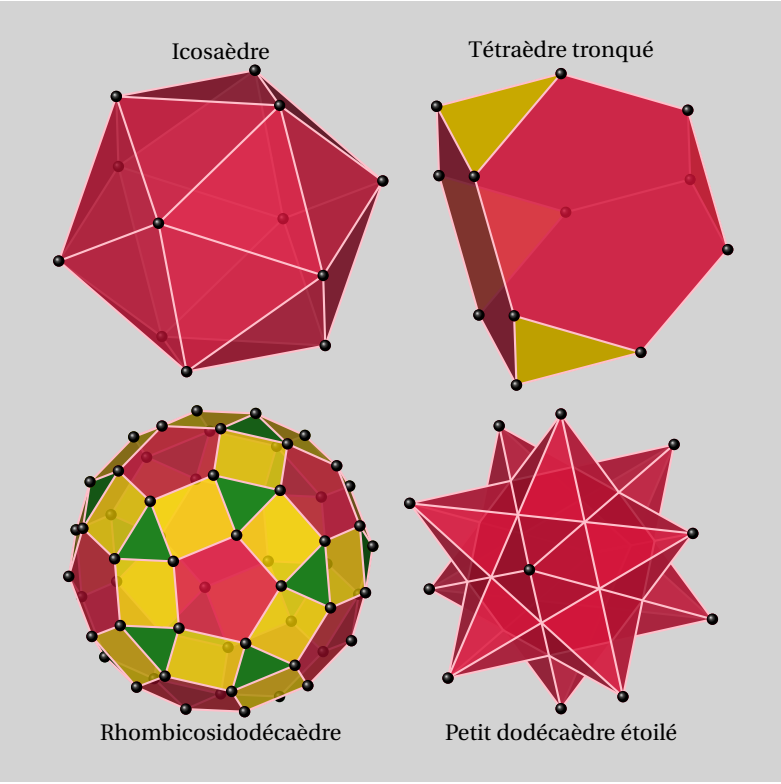
- Autres solides :
  - La fonction **octahemioctahedron(C, S [, all])** permet la construction d'un octahémioctaèdre de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a deux types de faces.
  - La fonction **small_stellated_dodecahedron(C, S [, all])** permet la construction d'un petit dodécaèdre étoilé de centre ⟨*C*⟩ (point 3D) et dont un sommet est ⟨*S*⟩ (point 3D). Ce solide a un seul type de faces.

```
\begin{luadraw}{name=polyhedrons}
local ld = luadraw
local M, Origin = ld.pt3d.M, ld.pt3d.Origin
local i = ld.cpx.I
local poly = require 'luadraw_polyhedrons' -- chargement du module

local g = ld.graph3d:new{bg="LightGray", size={10,10}}
g:Labelsize("small"); ld.Hiddenlines = false
-- en haut à gauche
g:Saveattr(); g:Viewport(-5,0,0,5); g:Coordsystem(-5,5,-5,5,true)
local T,S,A,F = poly.icosahedron(Origin,M(0,2,4.5),true)
g:Dscene3d(
    g:addFacet(F, {color="Crimson",opacity=0.8}),
    g:addPolyline(A, {color="Pink", width=8}),
    g:addDots(S) )
g:Dlabel("Icosaèdre",5*i,{})
g:Restoreattr()
-- en haut à droite
g:Saveattr()
g:Viewport(0,5,0,5); g:Coordsystem(-5,5,-5,5,true)
local T,S,A,F1,F2 = poly.truncatedtetrahedron(Origin,M(0,0,5),true) -- sortie complète, affichage dans une scène 3D
g:Dscene3d(
    g:addFacet(F1, {color="Crimson",opacity=0.8}),
    g:addFacet(F2, {color="Gold"}),
    g:addPolyline(A, {color="Pink", width=8}),
```

```
    g:addDots(S) )
g:Dlabel("Tétraèdre tronqué",5*i,{})
g:Restoreattr()
-- en bas à gauche
g:Saveattr(); g:Viewport(-5,0,-5,0); g:Coordsystem(-5,5,-5,5,true)
local T,S,A,F1,F2,F3 = poly.rhombicosidodecahedron(Origin,M(0,0,4.5),true)
g:Dscene3d(
    g:addFacet(F1, {color="Crimson",opacity=0.8}),
    g:addFacet(F2, {color="Gold",opacity=0.8}), g:addFacet(F3, {color="ForestGreen"}),
    g:addPolyline(A, {color="Pink", width=8}), g:addDots(S) )
g:Dlabel("Rhombicosidodécaèdre",-5*i,{})
g:Restoreattr()
-- en bas à droite
g:Saveattr(); g:Viewport(0,5,-5,0); g:Coordsystem(-5,5,-5,5,true)
local T,S,A,F1 = poly.small_stellated_dodecahedron(Origin,M(0,0,5),true)
g:Dscene3d(
    g:addFacet(F1, {color="Crimson",opacity=0.8}),
    g:addPolyline(A, {color="Pink", width=8}),
    g:addDots(S) )
g:Dlabel("Petit dodécaèdre étoilé",-5*i,{})
g:Restoreattr()
g:Show()
\end{luadraw}
```

FIGURE 1 : Polyèdres du module *luadraw_polyhedrons*



2) Le module *luadraw_spherical*

Ce module permet de dessiner un certain nombre d'objets sur une sphère (comme par exemple des cercles, des triangles sphériques, ...) sans avoir à gérer à la main les parties visibles ou non visibles. Le dessin se fait en trois temps :

1. On définit les caractéristiques de la sphère (centre, rayon, couleur,...)
2. On définit les objets à ajouter dans la scène, grâce à des méthodes dédiées.
3. On affiche le tout avec la méthode **g:Dspherical()**.

Bien sûr, toutes les méthodes de dessin 2D et 3D restent utilisables.

Utilisation : ce module ajoute de nouvelles méthodes graphiques à la classe *ld.graph3d*, il ne renvoie rien, les fonctions introduites par ce module vont dans l'espace de noms *luadraw*.

Fonctions globales du module

Par défaut ces fonctions s'appliquent à la sphère courante, mais certaines d'entre elles peuvent être appliquées à une autre sphère que la sphère courante, celles-ci ont leur dernier argument appelé ⟨*sph*⟩, celui-ci est optionnel, il vaut `nil` par défaut, mais lorsqu'il est précisé ce doit être une table de la forme {centre, rayon} (représentant une sphère).

- **ld.sM(x, y, z [, sph])** : renvoie un point de la sphère, c'est le point I de la sphère tel que la demi-droite [O, I) (O étant le centre de la sphère) passe par le point A de coordonnées cartésiennes (x, y, z). C'est le projeté du point M(x, y, z) sur la sphère partant du centre.
- **ld.sM(theta, phi [, sph])** : où ⟨*theta*⟩ et ⟨*phi*⟩ sont des angles en degrés, renvoie un point de la sphère donc les coordonnées sphériques sont *(R,theta,phi)* où R est le rayon de la sphère.
- **ld.smidpoint(A, B, x [, sph])** : ⟨*A*⟩ et ⟨*B*⟩ sont deux points de la sphère et ⟨*x*⟩ un nombre entre 0 et 1, la fonction renvoie le point d'abscisse curviligne ⟨*x*⟩ sur l'arc de grand cercle $\overset{\frown}{\text{AB}}$, c'est donc un point de la sphère. Par exemple avec $x = 0.5$ le point renvoyé est le milieu de l'arc $\overset{\frown}{\text{AB}}$.
- **ld.sbarycenter(list [, sph])** : où ⟨*list*⟩ = {A1,c1,A2,c2,…,An,cn}, les arguments ⟨*A1*⟩, …,⟨*An*⟩ sont des points de la sphère et les arguments ⟨*c1*⟩, …,⟨*cn*⟩ sont des nombres réels, la fonction calcule le barycentre de ces points avec les coefficients donnés et renvoie son projeté sur la sphère partant du centre.
- **ld.toSphere(A [, sph])** : renvoie le projeté du point ⟨*A*⟩ sur la sphère partant du centre.
- **ld.interSphericalC(P1, P2 [, sph])** : renvoie sous forme d'une séquence, les points d'intersection (s'ils existent) entre deux cercles appartenant à la sphère (pas nécessairement des grands cercles). Les deux arguments ⟨*P1*⟩ et ⟨*P2*⟩ sont deux plans, et c'est leur intersection avec la sphère qui forme les deux cercles dont on cherche l'intersection.
- **ld.interGreatC(C1,C2 [, sph])** : renvoie sous forme d'une séquence, les deux points d'intersection des deux grands cercles ⟨*C1*⟩ et ⟨*C2*⟩ appartenant à la sphère. Un grand cercle de la sphère est une liste de deux points de la sphère **non alignés avec le centre**.
- **ld.projstereo_Scircle(P, N, h)** : renvoie sous forme de chemin la projection stéréographique d'un cercle tracé sur la sphère. L'argument ⟨*P*⟩ est un plan et c'est son intersection avec la sphère qui forme le cercle qui va être projeté. L'argument ⟨*N*⟩ désigne un point de la sphère qui sera le pôle de la projection. L'argument ⟨*h*⟩ est un réel qui définit le plan de la projection, ce plan est perpendiculaire à l'axe (CN), où C est le centre de la sphère, et passe par le point $\text{I} = \text{C} + h\frac{\vec{\text{CN}}}{\text{CN}}$, avec $h = 0$ c'est le plan équatorial, avec $h = -\text{R}$, où R est le rayon de la sphère, c'est le plan tangent à la sphère au pôle opposé).
- **ld.projstereo_Sfacet(L, N, h [, close])** : renvoie sous forme de chemin la projection stéréographique d'une facette sphérique (tracée sur la sphère). L'argument ⟨*L*⟩ est une liste de points de la sphère qui forme les sommets de la facette, deux sommets consécutifs sont reliés par un arc de grand cercle (l'écart angulaire entre deux sommets consécutifs ne doit pas dépasser 180 degrés). L'argument ⟨*N*⟩ désigne un point de la sphère qui sera le pôle de la projection. L'argument ⟨*h*⟩ est un nombre réel qui définit le plan de projection. Ce plan est perpendiculaire à l'axe (CN), où C est le centre de la sphère, et passe par le point $\text{I} = \text{C} + h\frac{\vec{\text{CN}}}{\text{CN}}$. Pour $h = 0$, il s'agit du plan équatorial ; pour $h = -\text{R}$ (où R est le rayon de la sphère), il s'agit du plan tangent à la sphère au pôle opposé. L'argument optionnel ⟨*close*⟩ indique si la liste ⟨*L*⟩ doit être fermée (`true` par défaut).

Le module propose également les deux méthodes suivantes :

- La méthode **g:Split_facet(facet [, sph])**, où ⟨*facet*⟩ est une liste de points **qui doivent être sur la sphère**. La méthode renvoie une séquence de la forme :`front, back`, ce sont deux chemins 3D, le premier représente le contour de la partie visible de la facette sphérique (les arêtes sont des arcs de grands cercle), le second chemin représente le contour de la partie de la facette qui est sur la face cachée de la sphère.
- La méthode **g:Split_region(L [, sph])**, où ⟨*L*⟩ est une liste de points 3D représentant une **courbe fermée tracée sur la sphère**. La méthode renvoie une séquence de la forme :`front, back, out`, le premier élément est un chemin 3D représentant le contour de la partie visible de la région sphérique, le second élément est un chemin 3D représente le contour de la partie de la région qui est sur la face cachée de la sphère, le dernière élément est une table contenant les points correspondant aux extrémités des parties cachées.

Définition de la sphère

La sphère est définie avec la méthode **g:Define_sphere(options)**, où ⟨*options*⟩ est une table permettant d'ajuster chaque paramètre. Ceux-ci sont les suivants (avec leur valeur par défaut) :

- `center=pt3d.Origin`,

- `radius=3`,
- `color="orange"`,
- `opacity=1`,
- `mode=ld.mBorder`, mode d'affichage de la sphère (valeurs possibles ; `ld.mWireframe` ou `ld.mGrid` ou `ld.mBorder`),
- `edgecolor="lightgray"`,
- `edgestyle="solid"`,
- `hiddenstyle=ld.Hiddenlinestyle`,
- `hiddencolor="gray"`,
- `edgewidth=4`,
- `show=true`, pour montrer ou non la sphère,
- `back=true`, pour montrer ou non ce qui est derrière la sphère ou sur la face cachée,
- `inside=true`, pour montrer ou non ce qui est à l'intérieur de la sphère,
- `front=true`, pour montrer ou non ce qui est devant la sphère ou sur la face visible,
- `insidelabelcolor="darkgray"` : définit la couleur des labels dont le point d'ancrage est intérieur à la sphère.
- `arrowBstyle="->"` : type de flèche en fin de ligne
- `arrowAstyle="<-"` : type de flèche en début de ligne
- `arrowABstyle="<->"` : très peu utilisée car la plupart du temps les lignes tracées sur la sphère doivent être découpées.
- `hiddendelayed=false` : avec la valeur `false` les lignes cachées sont dessinées à la fin de l'instruction **g:Dspherical()**, avec la valeur `true` elles sont dessinées à la toute fin du graphique en cours ce qui peut être utile si vous avez ajouté après la sphère des éléments qui cachent une partie de celle-ci (cependant on peut modifier ce comportement localement avec l'option `hidden=true/false`).

La méthode **g:Clear_spherical()** permet de supprimer les objets qui ont été ajoutés à la scène, et remet les valeurs par défaut.

Ajout d'un label aux éléments graphiques

La plupart des éléments graphiques qui suivent acceptent dans leurs options, l'ajout d'un label. Les options concernant cet éventuel label sont :

- `label=""` : label à ajouter.
- `anchor1d=nil` : nombre entre 0 et 1 indiquant la position du label le long de la projection 2D de l'objet (0 pour la début de ligne, 1 pour la fin de ligne).
- `anchor=nil` : point 3D représentant le point d'ancrage du label dans l'espace.
- `anchor2d=cpx.Z(0.5,0.5)` : nombre complexe représentant la position du label dans le pavé $[0;1]\times[0;1]$ qui représente la boîte englobante 2D de la projection 2D de l'objet, donc par défaut le label est au centre de cette boîte. L'ordre de priorité est : `anchor`, `anchor1d`, `anchor2d` (si la première option vaut `nil`, la seconde est choisie, si elle vaut `nil` également, alors la troisième est choisie).
- `pos="center"`, indique la position du label par rapport au point d'ancrage, il peut valoir `"center"` (centré), `"N"` (nord), `"NE"` (nord est), `"E"` (est), `"SE"` (sud est), `"S"` (sud), `"SW"` (sud ouest), `"W"` (ouest), `"NW"` (nord ouest). Par défaut, il vaut `center`, et dans ce cas le label est centré sur le point d'ancrage.
- `dist=0`, distance en cm entre le label et son point d'ancrage lorsque `pos` n'est pas égal à `"center"`.
- `dir={}`, cette option est une table de la forme `{dirX, dirY [,dep]}` qui indique la direction de l'écriture. Les 3 valeurs `dirX`, `dirY` et `dep` sont trois vecteurs 3D, les deux premiers indiquent le sens de l'écriture, le troisième un déplacement (translation) du label par rapport au point d'ancrage. Le vecteur `dep` est nul par défaut.
- `node_options=""` est une chaîne (vide par défaut) destinée à recevoir des options qui seront directement passées à TikZ dans l'instruction *node[]*.

Ajouter un cercle : g:DScircle

La méthode **g:DScircle(P, options)** permet d'ajouter un cercle sur la sphère, l'argument $\langle P\rangle$ est une table de la forme $\{A, n\}$ qui représente un plan (passant par A et normal à n, deux points 3D). Le cercle est alors défini comme l'intersection de ce plan avec la sphère. L'argument $\langle options\rangle$ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `style=<style courant de ligne>`,

- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `out=nil`, si on affecte une variable de type liste à ce paramètre ⟨*out*⟩, alors la fonction ajoute à cette liste les deux points correspondant aux extrémités de l'arc caché, s'il y en a un, ce qui permet de les récupérer sans avoir à les calculer.
- options pour ajouter un label : voir page 159.

Ajouter un grand cercle : g:DSgreatcircle

La méthode **g:DSgreatcircle(AB, options)** permet d'ajouter un grand cercle sur la sphère, l'argument ⟨*AB*⟩ est une table de la forme {A, B} où A et B sont deux points distincts de la sphère. Le grand cercle est alors le cercle de centre le centre de la sphère, et passant par A et B. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `style=<style courant de ligne>`,
- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `out=nil`, si on affecte une variable de type liste à ce paramètre ⟨*out*⟩, alors la fonction ajoute à cette liste les deux points correspondant aux extrémités de l'arc caché, s'il y en a un, ce qui permet de les récupérer sans avoir à les calculer.
- options pour ajouter un label : voir page 159.

Ajouter un arc de grand cercle : g:DSarc

La méthode **g:DSarc(AB, sens, options)** permet d'ajouter un arc de grand cercle sur la sphère, l'argument ⟨*AB*⟩ est une table de la forme {A, B} où A et B sont deux points distincts de la sphère, on trace alors l'arc de grand cercle allant de A vers B. L'argument ⟨*sens*⟩ vaut 1 ou −1 pour indiquer le sens de l'arc. Lorsque A et B ne sont pas diamétralement opposés, le plan OAB (où O est le centre de la sphère) est orienté avec $\vec{OA} \wedge \vec{OB}$. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `style=<style courant de ligne>`,
- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `arrows=0`, trois valeurs possibles : 0 (pas de flèche), 1 (une flèche en B), 2 (flèche en A et en B).
- `normal=nil`, permet de préciser un vecteur normal au plan OAB lorsque ces trois points sont alignés.
- options pour ajouter un label : voir page 159.

Ajouter un angle : g:DSangle

La méthode **g:DSangle(B, A, C, r, sens, options)** où ⟨*A*⟩, ⟨*B*⟩ et ⟨*C*⟩ sont trois points de la sphère, permet de dessiner un arc de grand cercle sur la sphère pour représenter l'angle $(\vec{AB}, \vec{AC})$ avec un rayon de ⟨*r*⟩. L'argument ⟨*sens*⟩ vaut 1 ou −1 pour indiquer le sens de l'arc, le plan ABC est orienté avec $\vec{AB} \wedge \vec{AC}$. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `style=<style courant de ligne>`,
- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `arrows=0`, trois valeurs possibles : 0 (pas de flèche), 1 (une flèche en B), 2 (flèche en A et en B).

- `normal=nil`, permet de préciser un vecteur normal au plan OAB lorsque ces trois points sont alignés.
- Options pour ajouter un label :
  - `label=""` : texte qui sera affiché.
  - `node_options=""` : chaîne définissant les options pour le label.
  - `pos="auto"` : précise la position du label par rapport au point d'ancrage, les autres valeurs possibles sont les valeurs habituelles pour positionner un label : `"center"`, `"N"`, `"NW"`, etc. Par défaut, le point d'ancrage est situé à l'intersection de l'arc de cercle et de la bissectrice de l'angle.
  - `dist=r` : distance entre le point d'ancrage et le centre du cercle (A), par défaut c'est le rayon $\langle r \rangle$ du cercle.
  - `angle=0` : angle (en degrés) de la rotation que doit effectuer le point d'ancrage par défaut autour du centre du cercle (A) et dans le plan du cercle.
  - `rotate="none"` : indique si le label doit être tourné autour de son point d'ancrage **dans le plan de l'écran**. Les autres valeurs possibles sont : `"auto"`, dans ce cas le label est écrit en suivant la direction de la bissectrice, ou bien `"ortho"`, et dans ce cas le label est écrit perpendiculairement à la bissectrice.
  - `rotate3d="none"` : indique si le label doit être tourné autour de son point d'ancrage **dans le plan** (ABC). Les autres valeurs possibles sont : `"auto"`, dans ce cas le label est écrit en suivant la direction de la bissectrice, ou bien `"ortho"`, et dans ce cas le label est écrit perpendiculairement à la bissectrice (toujours dans le plan (ABC)). Avec la valeur `"none"` le label est écrit dans le plan de l'écran.
  - `fill=""` et `fillopacity=0.5` : lorsque l'option `fill` désigne un nom de couleur, le secteur angulaire (sphérique) est peint avec cette couleur et l'opacité contenue dans l'option `fillopacity`.

Ajouter une facette sphérique : g:DSfacet

La méthode **g:DSfacet(F, options)** où $\langle F \rangle$ est une liste de points de la sphère, permet de dessiner la facette représentée par $\langle F \rangle$, les arêtes étant des arcs de grands cercles. L'argument $\langle options \rangle$ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `style=<style courant de ligne>`,
- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `fill=""`, chaîne représentant la couleur de remplissage (aucune par défaut),
- `fillopacity=0.3`, opacité de la couleur de remplissage.
- `front=nil`, `back=nil` : si on affecte une variable de type table à ces options, alors la fonction ajoute à ces tables le chemin correspondant au contour de la partie visible pour la première option (front), et le chemin correspondant au contour de la partie cachée pour la deuxième option (back).
- options pour ajouter un label : voir page 159.

Ajouter une courbe sphérique : g:DScurve

La méthode **g:DScurve(L, options)** où $\langle L \rangle$ est une liste de points de la sphère, permet de dessiner la courbe représentée par $\langle L \rangle$. L'argument $\langle options \rangle$ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `style=<style courant de ligne>`,
- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `out=nil`, si on affecte une variable de type table à cette option `out`, alors la fonction ajoute à cette liste les points correspondant aux extrémités des parties cachées.
- options pour ajouter un label : voir page 159.

Ajouter une région sphérique : g:DSregion

La méthode **g:DSregion(L, options)** où ⟨*L*⟩ est une liste de points 3D représentant une courbe sphérique fermée simple. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `style=<style courant de ligne>`,
- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `fill=""`, chaîne représentant la couleur de remplissage (aucune par défaut),
- `fillopacity=0.3`, opacité de la couleur de remplissage.
- `dir={1,1}`, permet de changer le sens des arcs de cercle qui "remplacent" les parties cachées (pour la première valeur), ou les parties visibles (pour la seconde valeur). La valeur 1 indique le sens direct, et la valeur -1 le sens indirect.
- `out=nil`, si on affecte une variable de type table à cette option `out`, alors la fonction ajoute à cette liste les points correspondant aux extrémités des parties cachées.
- `front=nil`, `back=nil` : si on affecte une variable de type table à ces options, alors la fonction ajoute à ces tables le chemin correspondant au contour de la partie visible pour la première option (front), et le chemin correspondant au contour de la partie cachée pour la deuxième option (back).
- options pour ajouter un label : voir page 159.

Nous allons maintenant traiter d'objets qui ne sont pas forcément sur la sphère, mais qui peuvent la traverser, ou être à l'intérieur, ou à l'extérieur.

Ajouter un segment : g:DSseg

La méthode **g:DSseg(AB, options)** permet d'ajouter un segment, l'argument ⟨*AB*⟩ est une table de la forme {A, B} où A et B sont deux points de l'espace. La fonction traite les interactions avec la sphère. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `style=<style courant de ligne>`,
- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `arrows=0`, trois valeurs possibles : 0 (pas de flèche), 1 (une flèche en B), 2 (flèche en A et en B).
- options pour ajouter un label : voir page 159.

Ajouter une droite : g:DSline

La méthode **g:DSline(d, options)** permet d'ajouter une droite, l'argument ⟨*d*⟩ est une table de la forme {A, *u*} où A et un point de la droite et *u* un vecteur directeur (deux points 3D). La fonction traite les interactions avec la sphère. Le segment tracé est obtenu en intersectant la droite avec la fenêtre 3D, il peut être vide si la fenêtre est trop étroite. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `style=<style courant de ligne>`,
- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `arrows=0`, trois valeurs possibles : 0 (pas de flèche), 1 (une flèche en B), 2 (flèche en A et en B).
- `scale=1`, permet de modifier la taille du segment tracé.
- options pour ajouter un label : voir page 159.

Ajouter une ligne polygonale : g:DSpolyline

La méthode **g:DSpolyline(L, options)** permet d'ajouter une ligne polygonale, l'argument ⟨*L*⟩ est une liste de points de l'espace, ou une liste de listes de points de l'espace. La fonction traite les interactions avec la sphère. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `style=<style courant de ligne>`,
- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `arrows=0`, trois valeurs possibles : 0 (pas de flèche), 1 (une flèche en B), 2 (flèche en A et en B).
- `close=false`, indique si la ligne doit être refermée.
- options pour ajouter un label : voir page 159.

Ajouter des axes : g:DSaxes

La méthode **g:DSaxes(O, options)** permet d'ajouter les axes : (⟨*O*⟩,*vecI*), (⟨*O*⟩,*vecJ*) et (⟨*O*⟩,*vecK*), l'argument ⟨*O*⟩ est un point 3D représentant le point d'intersection des trois axes. La fonction traite les interactions avec la sphère. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut)

- `style=<style courant de ligne>`,
- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `arrows=0`, trois valeurs possibles : 0 (pas de flèche), 1 (une flèche à la fin), 2 (flèche en début et fin).
- `xyzlimits=nil` : par défaut l'intervalle considéré sur chacun des trois axes est celui défini par la fenêtre 3D à la création du graphe, mais celui-ci peut être changé (sans modifier la fenêtre), par exemple avec `xyzlimits={0,5}`, les trois intervalles seront égaux à [0;5]. Cependant ces trois intervalles peuvent être changés individuellement avec les options `xlimits`, `ylimits` et `zlimits`. Les axes ne sont pas gradués.
- `legend=true` : permet d'ajouter automatiquement une légende à l'extrémité de chaque axe, ces légendes sont gérées par l'option `labels={"$x$","$y$","$z$"}`.
- Lorsque `legend=true`, les labels sont positionnés à l'extrémité finale du segment plus une longueur égale à 1/40 fois la longueur du segment. Cette distance peut être modifiée, soit globalement avec l'option `xyzdist=0` (longueur nulle par défaut), soit individuellement avec les options `xdist`, `ydist` et `zdist`, par défaut ces trois options prennent la valeur de `xyzdist`.

Ajouter un plan : g:DSplane

La méthode **g:DSplane(P, options)** permet d'ajouter le contour d'un plan, l'argument ⟨*P*⟩ est une table de la forme {A, *n*} où A est un point du plan et *n* un vecteur normal. La fonction dessine un parallélogramme représentant le plan ⟨*P*⟩ en traitant les interactions avec la sphère. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `style=<style courant de ligne>`,
- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `scale=1`, permet de changer la taille du parallélogramme,
- `angle=0`, angle en degrés, permet de faire pivoter le parallélogramme autour de la droite perpendiculaire passant par le centre de la sphère.
- `trace=true`, permet de dessiner ou non, l'intersection du plan avec la sphère lorsqu'elle n'est pas vide.
- options pour ajouter un label : voir page 159.

Ajouter un label : g:DSlabel

La méthode **g:DSlabel(text1, anchor1, options1, text2, anchor2, options2, ...)** permet d'ajouter un ou plusieurs labels sur le même principe que la méthode **g:Dlabel3d()**, sauf qu'ici la fonction traite les cas où le point d'ancrage est à l'intérieur de la sphère, derrière la sphère ou devant la sphère. Dans le cas où il est à l'intérieur la couleur du label est donnée par l'option de la sphère `insidelabelcolor` qui vaut `"darkgray"` par défaut.

Ajouter des points : g:DSdots et g:DSstars

La méthode **g:DSdots(dots, options)** permet d'ajouter des points dans la scène, l'argument ⟨*dots*⟩ est une liste de points 3D. La fonction dessine les points en gérant les interactions avec la sphère. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `hidden=ld.Hiddenlines`,
- `mark_options=""`, chaîne qui sera passée directement à l'instruction *\draw*.

Dans le cas où un point est à l'intérieur de la sphère, ou sur la face cachée, la couleur du point est donnée par l'option de la sphère `insidelabelcolor` qui vaut `"darkgray"` par défaut.

La méthode **g:DSstars(dots, options)** permet d'ajouter des points **sur** la sphère, l'argument ⟨*dots*⟩ est une liste de points 3D qui seront projetés sur la sphère. La fonction dessine ces points en forme d'astérisque. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `style=<style courant de ligne>`,
- `color=<couleur courante des lignes>`,
- `width=<épaisseur courante des lignes en dixième de point>`,
- `opacity=<opacité courante des lignes>`,
- `hidden=ld.Hiddenlines`,
- `scale=1`, permet de changer la taille du point,
- `circled=false`, permet d'ajouter une cercle autour de l'étoile,
- `fill=""`, chaîne représentant une couleur, lorsqu'elle n'est pas vide, l'astérisque est remplacée par une facette hexagonale cerclée et remplie avec la couleur donnée par cette option.

Les points qui sont sur la face cachée de la sphère ont la couleur donnée par l'option de la sphère `insidelabelcolor` qui vaut `"darkgray"` par défaut.

Stéréographie inverse : g:DSinvstereo_curve et g:DSinvstereo_polyline

La méthode **g:DSinvstereo_curve(L, options)**, où ⟨*L*⟩ est une ligne polygonale 3D représentant une courbe tracée sur un plan d'équation z =cte, dessine sur la sphère l'image de ⟨*L*⟩ par stéréographie inverse, le pôle étant le point *C+r*vecK*, où C est le centre de la sphère et r le rayon.

La méthode **g:DSinvstereo_polyline(L, options)**, où ⟨*L*⟩ est une ligne polygonale 3D tracée sur un plan d'équation z =cte, dessine sur la sphère l'image de L par stéréographie inverse, le pôle étant le point *C+r*vecK*, où C est le centre de la sphère et r le rayon.

Dans les deux cas, les ⟨*options*⟩ sont les mêmes que pour la méthode **g:DScurve()**.

Ajouter des chemins

Le module affiche les éléments de la scène en quatre étapes

- Étape 1 : affichage des éléments qui sont derrière la sphère.
- Étape 2 : affichage des éléments qui sont à l'intérieur de la sphère.
- Étape 3 : affichage des éléments qui sont devant la sphère.
- Étape 4 : affichage des éléments "cachés".

Les méthodes suivantes permettent d'ajouter d'autres éléments (chemins), mais ces éléments ne sont pas "vérifiés" par le module.

- La méthode **g:DSaddback(L [, draw_options, hidden, hidden_options])** permet d'afficher le chemin, ou la ligne polygonale, ⟨*L*⟩ après l'étape 1 et avant l'étape 2, ce peut être par exemple un chemin dessiné sur la face cachée de la sphère. ⟨*draw_options*⟩ est une chaîne de caractères qui sera transmise à l'instruction *\draw*. ⟨*hidden*⟩ est un booléen

(`false` par défaut) indiquant si la partie cachée doit être dessinée. ⟨*hidden_options*⟩ est une chaîne de caractères qui sera transmise à l'instruction *\draw*.

- La méthode **g:DSaddinside(L [, draw_options, hidden, hidden_options])** permet d'afficher le chemin, ou la ligne polygonale, ⟨*L*⟩ en même temps que l'étape 2 (à l'intérieur de la sphère). Les options sont les mêmes que pour la méthode précédente.
- La méthode **g:DSaddfront(L [, draw_options])** permet d'afficher le chemin, ou ligne polygonale, ⟨*L*⟩ après l'étape 2 et avant l'étape 3, ce peut être par exemple un chemin dessiné sur la face visible de la sphère. ⟨*draw_options*⟩ est une chaîne de caractères qui sera transmise à l'instruction *\draw*.

Méthodes supplémentaires : g:Dhalf_sphere, g:Dquater_sphere, g:Deighth_sphere

Ces trois méthodes sont indépendantes de la sphère crée par la méthode **g:Define_sphere()**, mais elles utilisent certaines fonctions de ce module.

- **g:Dhalf_sphere(O, r, dir, options)** : dessine une demi-sphère, ⟨*O*⟩ est le centre, ⟨*r*⟩ est le rayon, ⟨*dir*⟩ est un vecteur 3D non nul, la demi-sphère est obtenue en coupant la sphère avec le plan {O, dir}, et en conservant la partie dans le demi-espace qui contient le vecteur ⟨*dir*⟩. Il y a une face sphérique et une face plane circulaire.
- **g:Dquater_sphere(O, r, dir1, dir2, options)** : dessine un quart de sphère, ⟨*O*⟩ est le centre, ⟨*r*⟩ est le rayon, ⟨*dir1*⟩ et ⟨*dir2*⟩ sont deux vecteurs 3D non nuls et orthogonaux, le quart de sphère est obtenu en coupant la sphère avec les plans {O, dir1} et {O, dir2}, et en conservant la partie dans le quart d'espace qui contient les vecteurs ⟨*dir1*⟩ et ⟨*dir2*⟩. Il y a une face sphérique et deux faces planes semi-circulaires.
- **g:Deighth_sphere(O, r, dir1, dir2, dir3, options)** : dessine un huitième de sphère, ⟨*O*⟩ est le centre, ⟨*r*⟩ est le rayon, ⟨*dir1*⟩, ⟨*dir2*⟩ et ⟨*dir3*⟩ sont trois vecteurs 3D non nuls et orthogonaux deux à deux, le huitième de sphère est obtenu en coupant la sphère avec les plans {O, dir1}, {O, dir2} et {O, dir3}, et en conservant la partie dans le huitième d'espace qui contient les vecteurs ⟨*dir1*⟩, ⟨*dir2*⟩ et ⟨*dir3*⟩. Il y a une face sphérique et trois faces planes en quart de cercle.

Pour ces trois méthodes, l'argument ⟨*options*⟩ est une table dont les champs définissent les options, celles-ci sont (avec leur valeur par défaut) :

- `color="white"` : couleur de remplissage des faces.
- `mode=ld.mShaded` : mode de représentation, par défaut les faces sont peintes et dessinées avec les arêtes visibles. Les autres modes possibles sont :
  - `ld.mShadedHidden` : faces peintes, avec les arêtes visibles et les arêtes cachées.
  - `ld.mShadedOnly` : faces peintes mais pas d'arêtes.
  - `ld.mWireframe` : seulement les arêtes, les faces ne sont pas peintes.
- `opacity=opacité par défaut` : opacité pour le remplissage des faces.
- `edgecolor=couleur par défaut` : couleur des arêtes.
- `edgewidth=épaisseur par défaut` : épaisseur des arêtes (en dixième de point).
- `edgestyle=style par défaut` : style de ligne pour les arêtes visibles (chaîne de caractères).
- `hiddenstyle=ld.Hiddenlinestyle` : style de ligne pour les arêtes cachées lorsqu'elles sont dessinées.
- `outline=false` : avec la valeur `true` le contour de la partie sphérique visible est dessiné avec les attributs des arêtes visibles.

Exemples

```
\begin{luadraw}{name=cube_in_sphere}
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window={-9,9,-4,5},viewdir={25,70},size={16,8}}
require 'luadraw_spherical'
g:Linewidth(6); ld.Hiddenlinestyle = "dashed"
local a = 4
local O = Origin
local cube = ld.parallelep(O,a*vecI,a*vecJ,a*vecK)
local G = pt3d.isobar3d(cube.vertices)
cube = ld.shift3d(cube,-G) -- pour centrer le cube à l'origine
```

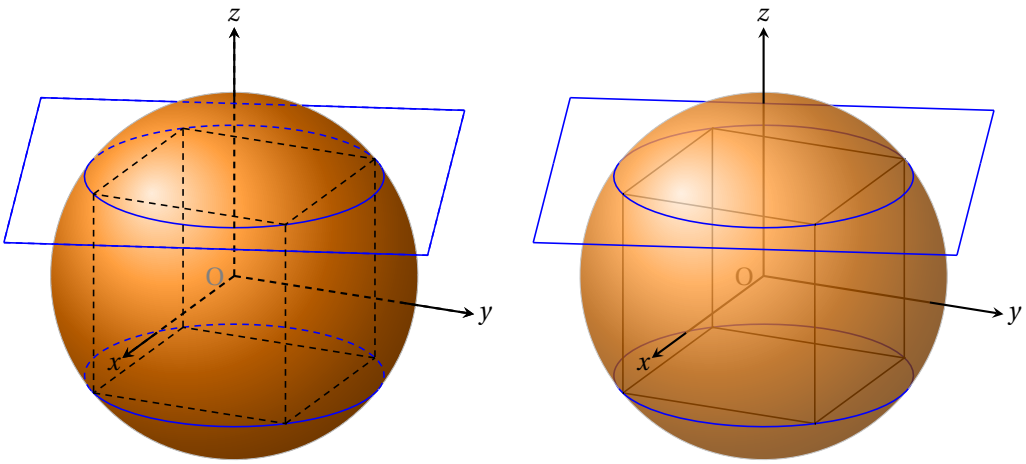
```
local R = pt3d.abs(cube.vertices[1])

local dessin = function()
    g:DSaxes(Origin, {width=8, arrows=1, xyzlimits={0,5}, xdist=0.1})
    g:DSplane({a/2*vecK,vecK},{color="blue",scale=0.9,angle=20});
    g:DScircle({-a/2*vecK,vecK},{color="blue"})
    g:DSpolyline( ld.facetedges(cube) ); g:DSlabel("$O$",O,{pos="W"})
    g:Dspherical()
end

g:Saveattr(); g:Viewport(-9,0,-4,5); g:Coordsystem(-5,5,-5,5)
ld.Hiddenlines = true; g:Define_sphere({radius=R, arrowBstyle = "-stealth"})
dessin()
g:Dlabel("Hiddenlines=true",0.5-4.5*cpx.I,{})
g:Restoreattr()

g:Saveattr(); g:Viewport(0,9,-4,5); g:Coordsystem(-5,5,-5,5)
ld.Hiddenlines = false; g:Define_sphere({radius=R,opacity=0.8, arrowBstyle = "-stealth"} )
dessin()
g:Dlabel("Hiddenlines=false, opacity=0.8",0.5-4.5*cpx.I,{})
g:Restoreattr()
g:Show()
\end{luadraw}
```

FIGURE 2 : Cube dans une sphère



Hiddenlines=true

Hiddenlines=false, opacity=0.8

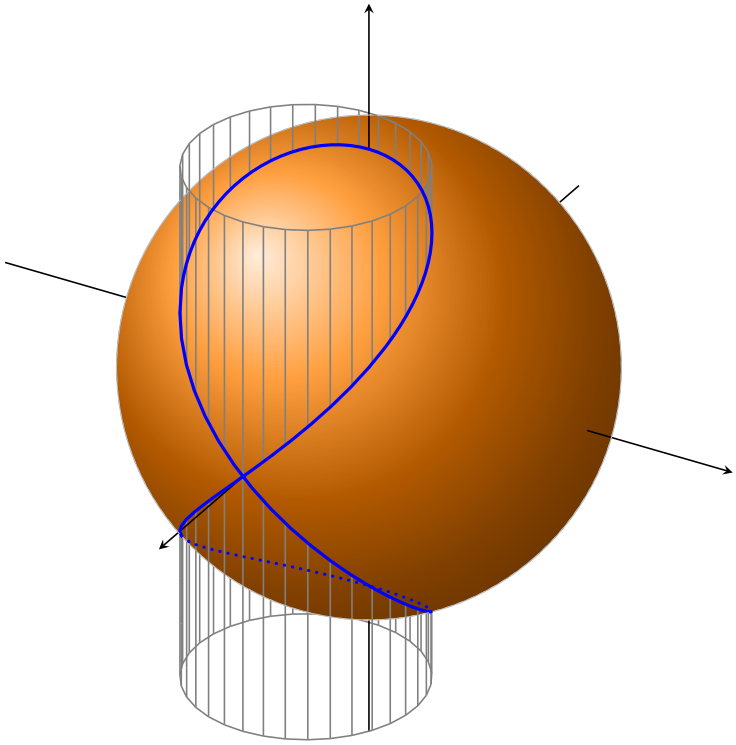
Courbe sphérique

```
\begin{luadraw}{name=courbe_spherique}
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Origin, vecI, vecJ, vecK, M, Ms = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M, pt3d.Ms

local g = ld.graph3d:new{window={-4.5,4.5,-4.5,4.5},viewdir={30,60},margin={0,0,0,0},size={10,10}}
require 'luadraw_spherical'
g:Linewidth(6); ld.Hiddenlinestyle = "dotted"
ld.Hiddenlines = false;
local C = ld.cylinder(M(1.5,0,-3.5),1.5,M(1.5,0,3.5),35,true)
local L = ld.parametric3d( function(t) return Ms(3,t-math.pi/2,t) end, -math.pi,math.pi) -- la courbe
g:Define_sphere({arrowBstyle = "-stealth"})
g:DSpolyline(ld.facetedges(C),{color="gray"}) -- affichage cylindre
g:DSpolyline({{-5*vecI,5*vecI},{-5*vecJ,5*vecJ},{-5*vecK,5*vecK}},{arrows=1}) --axes
ld.Hiddenlines=true; g:DScurve(L,{width=12,color="blue"}) -- courbe avec partie cachée
g:Dspherical()
g:Show()
```

```
\end{luadraw}
```

FIGURE 3 : Fenêtre de Viviani



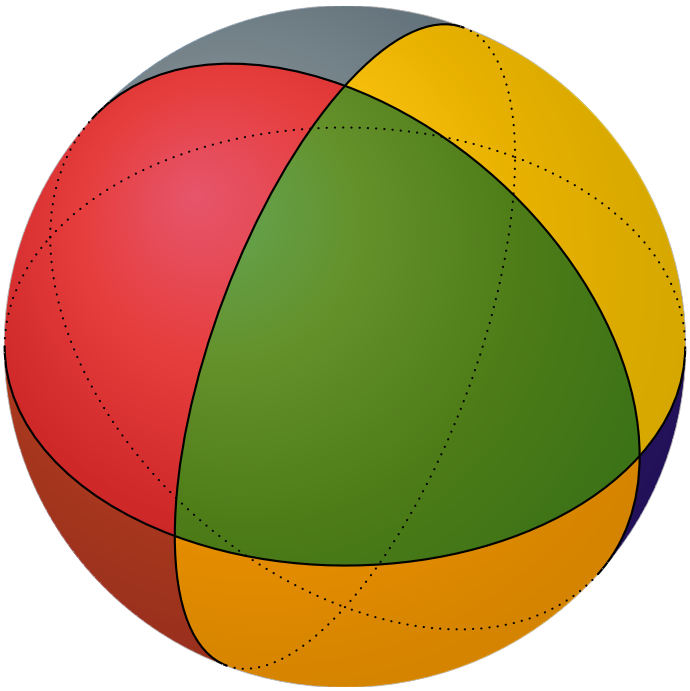
Pour ne pas nuire à la lisibilité du dessin, les parties cachées n'ont pas été affichées sauf celle de la courbe.

Un pavage sphérique

```
\begin{luadraw}{name=pavage_spherique}
local ld = luadraw
local Origin = ld.pt3d.Origin

local g = ld.graph3d:new{window={-3,3,-3,3},viewdir={30,60},size={10,10}}
require 'luadraw_spherical'
local poly = require "luadraw_polyhedrons"
g:Linewidth(6); ld.Hiddenlines = true; ld.Hiddenlinestyle = "dotted"
local P = ld.poly2facet( poly.octahedron(Origin, ld.sM(30,10)) )
local colors = {"Crimson","ForestGreen","Gold","SteelBlue","SlateGray","Brown","Orange","Navy"}
g:Define_sphere()
for k,F in ipairs(P) do
    g:DSfacet(F,{fill=colors[k],style="noline",fillopacity=0.7})  -- facettes sans les bords
end
for _, A in ipairs(ld.facetedges(P)) do
    g:DSarc(A,1,{width=8}) -- chaque arête est un arc de grand cercle
end
g:Dspherical()
g:Show()
\end{luadraw}
```

FIGURE 4 : Un pavage sphérique



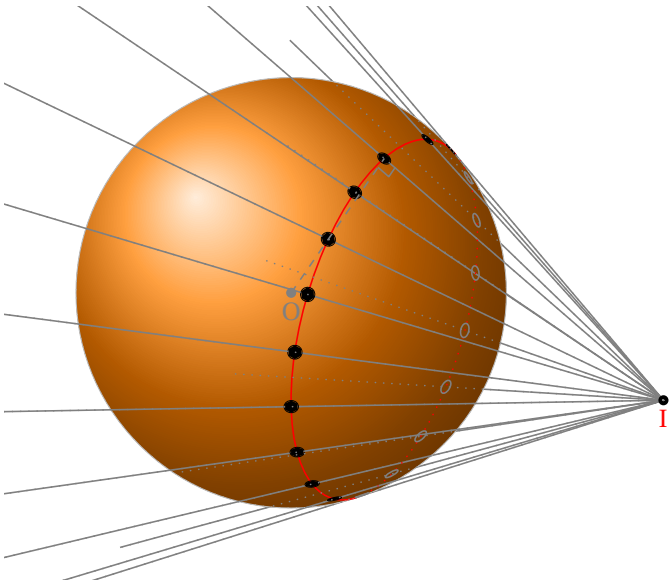
Pour ce pavage sphérique, on a choisi un octaèdre régulier de centre identique celui de la sphère et avec un sommet sur la sphère (et donc tous les sommets sont sur la sphère).

Tangentes à la sphère issues d'un point

```
\begin{luadraw}{name=tangent_to_sphere}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, M = pt3d.Origin, pt3d.M

local g = ld.graph3d:new{window={-4,5.5,-4,4},viewdir={30,60},size={10,10}}
require 'luadraw_spherical'
ld.Hiddenlines=true; g:Linewidth(6)
local O, I = Origin, M(0,6,0)
local S,S1 = {O, 3}, {(I+O)/2,pt3d.abs(I-O)/2}
-- le cerlce de tangence est l'intersection entre S et S1
local C,r,n = ld.interSS(S,S1)
local L = ld.circle3d(C,r,n)[1] -- liste de points du cercle
local dots, lines = {}, {}
-- draw
g:Define_sphere({opacity=1})
g:DScircle({C,n},{color="red"})
for k = 1, math.floor(#L/4) do
    local A = L[4*(k-1)+1]
    table.insert(dots,A)
    table.insert(lines,{I, 2*A-I})
end
g:DSpolyline(lines ,{color="gray"})
g:DSstars(dots) -- dessin de points sur la sphère
g:DSdots({O,I});  -- points dans la scène
g:DSlabel("$I$",I,{pos="S",node_options="red"},"$O$",O,{})
g:Dspherical()
g:Dseg3d({O,dots[1]},"gray,dashed"); g:Dangle3d(O,dots[1],I,0.2,"gray")
g:Show()
\end{luadraw}
```

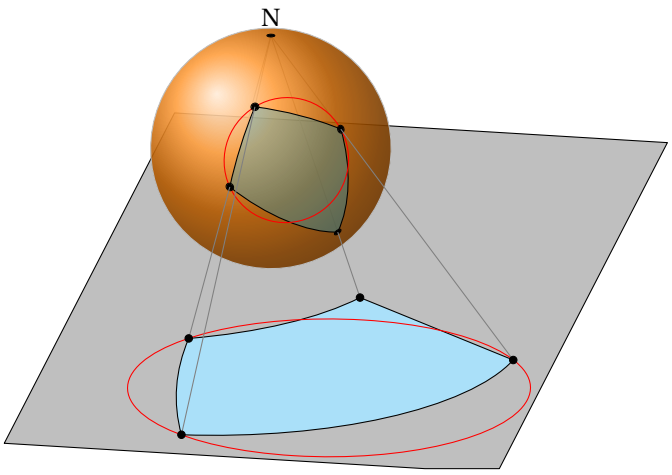
FIGURE 5 : Tangentes à la sphère issues d'un point



Projection stéréographique

```
\begin{luadraw}{name=projstereo_Sfacet}
local ld = luadraw
local pt3d = ld.pt3d
local M = pt3d.M
require 'luadraw_spherical'
local sM = ld.sM
local g = ld.graph3d:new{window={-7,10,-8,4}, size={10,10}, viewdir={10,70}}
local O, R = M(0,0,0), 3
g:Define_sphere( {color="orange", opacity=0.9} )
local A, B, C, D, N = sM(-10,90), sM(0, 50), sM(50,65), sM(50,120), sM(0,0)
local F = {A,B,C,D}
local p = ld.projstereo(F, {O,R}, N, -R)
local T1 = ld.projstereo_Sfacet(F, N, -R)
local T2 = ld.projstereo_Scircle(ld.plane(A,B,C), N, -R)
g:Dplane({M(2,2,-R), M(0,0,1)}, M(0,1,0), 12.5,24.5, "fill=lightgray")
g:DSpolyline( {{N,A},{N,B},{N,C},{N,D}}, {color="gray", hidden=false})
g:DSfacet(F, {fill="cyan"})
g:DScircle(ld.plane(A,B,C), {color="red"})
g:DSstars({A,B,C,D,N}, {fill="black"})
g:Dspherical()
g:Dpath3d(T1, "fill=cyan!30")
g:Dpath3d(T2, "red")
for k,A in ipairs(F) do
    g:Dpolyline3d( {A,p[k]}, "gray")
end
g:Ddots3d(p)
g:Dlabel3d("$N$",N,{pos="N"})
g:Show()
\end{luadraw}
```

FIGURE 6 : Projection stéréographique d'un cercle et d'une facette sphériques

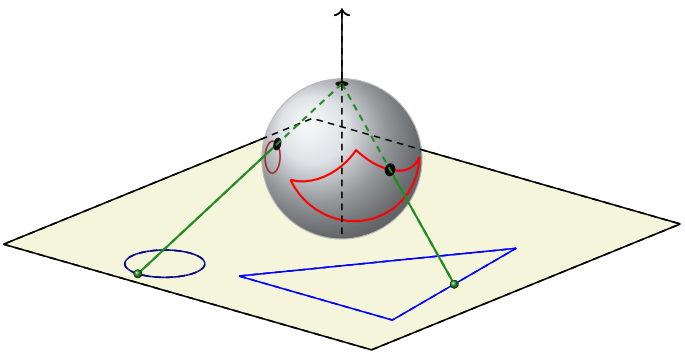


Stéréographie inverse

```
\begin{luadraw}{name=stereographic_curve}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, M, vecJ, vecK = pt3d.Origin, pt3d.M, pt3d.vecJ, pt3d.vecK

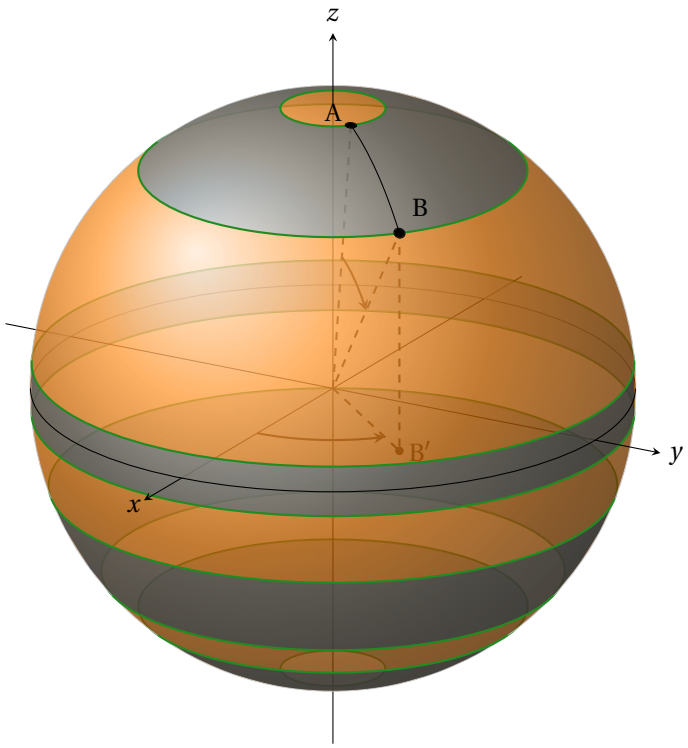
local g = ld.graph3d:new{window3d={-5,5,-2,2,-2,2},window={-4.25,4.25,-2.5,2},size={10,10}, viewdir={40,70}}
ld.Hiddenlines = true; ld.Hiddenlinestyle="dashed"; g:Linewidth(6)
require 'luadraw_spherical'
local C, R = Origin, 1
local a = -R
local P = ld.planeEq(0,0,1,-a)
local L = {M(2,0,a), M(2,2.5,a), M(-1,2,a)}
local L2 = ld.circle3d(M(2.25,-1,a),0.5,vecK)[1]
local A, B = (L[2]+L[3])/2, L2[20]
local a,b = table.unpack( ld.inv_projstereo({A,B},{C,R},C+R*vecK) )
g:Dplane(P,vecJ,6,6,15,"draw=none,fill=Beige")
g:Define_sphere( {center=C,radius=R, color="SlateGray!30", show=true} )
g:DSpolyline(L,{color="blue",close=true}); g:DSinvstereo_polyline(L,{color="red",width=8,close=true})
g:DSpolyline(L2,{color="Navy"}); g:DSinvstereo_curve(L2,{color="Brown",width=6})
g:DSplane(P,{scale=1.5})
g:DSpolyline({{C+R*vecK,A},{C+R*vecK,B}}, {color="ForestGreen",width=8})
g:DSpolyline({{-vecK,2*vecK}}, {arrows=1})
g:DSstars({C+R*vecK,a,b}, {scale=0.75})
g:Dspherical()
g:Dballdots3d({A,B},"ForestGreen",0.75)
g:Show()
\end{luadraw}
```

FIGURE 7 : Méthodes *DSinvstereo_curve* et *DSinvstereo_polyline*



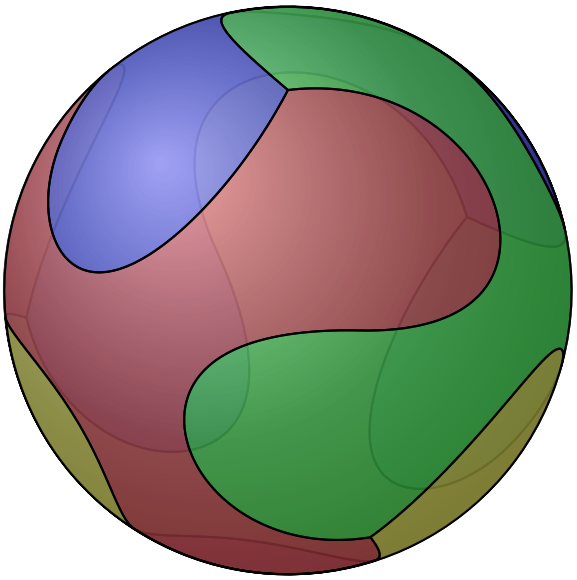
```
\begin{luadraw}{name=spherical_strip}
local ld = luadraw
local M, Ms = ld.pt3d.M, ld.pt3d.Ms
local g = ld.graph3d:new{ viewdir={30,70} }
require 'luadraw_spherical'
local R = 4
g:Define_sphere({radius=R, arrowBstyle="-stealth", opacity=0.8})
local sphere_strip = function(p1,p2,n)
    n = n or math.ceil((p2-p1)/5)
    p1 = p1*ld.deg; p2 = p2*ld.deg
    return ld.surface( function(u,v) return Ms(R,u,v) end, 2*math.pi, 0, p1, p2,{50,n})
end
local color1, color2 = "SteelBlue", "orange"
local strips ={ {10,40}, {85,95}, {110, 130}, {140,170} } -- angles
local O, I, J, K, A, B = M(0,0,0), M(1,0,0), M(0,1,0), M(0,0,1), ld.sM(50,10), ld.sM(50,40)
local B1 = ld.pxy(B)
local S = {}
for _, p in ipairs(strips) do
    ld.insert(S, sphere_strip(p[1],p[2]))
end
local Sv, Sh = g:Classifyfacet(S)
local back = ld.polyline2path3d( ld.border(Sh) )
local front = ld.polyline2path3d( ld.border(Sv) )
--drawing
g:DSaddback(back, "draw=none,even odd rule,left color="..color1.."!50,right color="..color1.."!80, fill opacity=0.6")
g:DSaddfront(front, "draw=none,even odd rule,ball color="..color1..", fill opacity=0.6")
for _, p in ipairs(strips) do
    g:DScircle({ld.sM(0,p[1]), M(0,0,1)}, {color="ForestGreen", width=8})
    g:DScircle({ld.sM(0,p[2]), M(0,0,1)}, {color="ForestGreen", width=8})
end
g:DSaddinside({A,O,B,B1,O}, "line width=0.8,dashed") -- polyline
g:DSaddinside({I,O,B1,2,1,"ca"}, "line width=0.8,-stealth") -- path
g:DSaddinside({A,O,B,2,1,"ca"}, "line width=0.8,-stealth")
g:DSstars({A,B}, {fill="black", scale=0.75}); g:DSdots(B1)
g:DSlabel("$B'$",B1,{pos="E"}, "$A$",1.05*A,{pos="W"}, "$B$",1.05*B,{pos="NE"})
g:DSarc({A,B},1)
g:DSaxes(O, {arrows=1})
g:DScircle( {O,K})
g:Dspherical()
g:Show()
\end{luadraw}
```

FIGURE 8 : Ajouter des chemins




```
\begin{luadraw}{name=DSregion}
local ld = luadraw
local pt3d = ld.pt3d
local Ms, O = pt3d.Ms, pt3d.Origin
local g = ld.graph3d:new{window={-4,4,-4,4},margin={0,0,0,0},viewdir={30,65},size={10,10}}
ld.Hiddenlines = false
ld.Hiddenlinestyle = "dashed"
g:Linewidth(8)
require 'luadraw_spherical'
local poly = require 'luadraw_polyhedrons'
local R = 3
local T = poly.tetrahedron(O, Ms(R,30*ld.deg,20*ld.deg))
local A, B, C, D = table.unpack(T.vertices)
local function edge(A,B,cut) -- cut = true/false, to remove the first point
    cut = cut or false
    local m = (A+B)/2
    local u = pt3d.normalize(B-A)
    local v = ld.rotate3d(u,90,{O,m})
    local L = ld.path3d({A,"m",A+2*v+u/2, m+2*v+u/2,m,"b",m-2*v-u/2,B-2*v-u/2,B,"b"},100)[1]
    if cut then table.remove(L,1) end
    return ld.map(ld.toSphere, L)
end
local ABD = ld.concat(edge(A,B), edge(B,D,true), edge(D,A,true) )
local ABC = ld.concat(edge(A,B), edge(B,C,true), edge(C,A,true) )
local ADC = ld.concat(edge(A,D), edge(D,C,true), edge(C,A,true) )
local BCD = ld.concat(edge(B,C), edge(C,D,true), edge(D,B,true) )
g:Define_sphere({radius=R, color="SlateGray", opacity=0.9})
g:DSregion(ABD,{fill="red"})
g:DSregion(ABC,{fill="green"})
g:DSregion(ADC,{fill="blue"})
g:DSregion(BCD,{fill="yellow"})
g:Dspherical()
g:Show()
\end{luadraw}
```

FIGURE 9 : Peindre une région sphérique



```
\begin{luadraw}{name=Split_facet}
local ld = luadraw
local pt3d = ld.pt3d
local M, Origin, vecI, vecJ, vecK = pt3d.M, pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK
local g = ld.graph3d:new{ size={10,10}, margin=0 }
require 'luadraw_spherical'
local r1, dr = 4.5, 0.75
local r2, r3, r4 = r1-dr, r1-2*dr, r1-3*dr
-- teal
g:Dhalf_sphere(Origin, r1, -vecJ, {color="teal", mode=ld.mShadedOnly})
-- yellow
g:Dpath3d({r2*vecI,Origin,-r2*vecI,r2,-1,vecJ,"ca"}, "draw=none,fill=yellow!80!black")
g:Dquater_sphere(Origin, r2, vecJ, -vecK, {color="yellow", mode=ld.mShadedOnly})
-- gray
g:Dpath3d({r3*vecI,Origin,-r3*vecI,r3,-1,vecJ,"ca"}, "draw=none,fill=gray")
g:Dpath3d({r3*vecI,Origin,-r3*vecI,r3,1,vecK,"ca"},"draw=none,fill=gray")
g:Deighth_sphere(Origin, r3, -vecI, vecJ, vecK, {color="gray", mode=ld.mShadedOnly})
-- orange
g:Deighth_sphere(Origin, r4, vecI, vecJ, vecK, {color="orange", mode=ld.mShadedOnly})
g:Show()
\end{luadraw}
```

FIGURE 10 : Exemple avec **g:Dhalf_sphere** et compagnie



3) Le module *luadraw_palettes*

Le module **luadraw_palettes**[1] définit 261 palettes de couleurs portant chacune un nom. Une palette est une liste (table) de couleurs qui sont elles-mêmes des listes de trois valeurs numériques entre 0 et 1 (composantes rouge, verte et bleue). La liste de ces palettes ainsi que leur rendu, peuvent être visualisés dans ce document.

Ce module n'ajoute pas de nouvelles méthodes graphiques, mais il renvoie une table de définitions de couleurs, plus la fonction **getPal(palname, options)** Par conséquent, il s'utilise ainsi (exemple) :

```
local pal = require 'luadraw_palettes'
local color = pal.Blackbody
local BlackbodyTransformed = pal.getPal( -- renvoie une nouvelle palette
    color, -- nom d'une palette
    {
    extract = {2, 5, 8, 9}, -- numéros des couleurs à extraire
    shift = 1, -- décalage parmi les couleurs extraites, ce qui donne ici: 5,8,9,2
    reverse = true -- inversion de l'ordre, ce qui donne ici: 2,9,8,5
    }
)
```

4) Le module *luadraw_compile_tex*

Attention :

- ce module ajoute de nouvelles méthodes graphiques aux classes ld.graph et *ld.graph3d* ainsi que plusieurs fonctions dans l'espace de noms *luadraw*, mais il ne renvoie rien.
- ce module a été testé sous linux, mais pas (encore) sous windows, ni macOS,
- ce module nécessite que soit installés les programmes : *pdflatex*, *pdf2ps* et *pstoedit* sur votre système,

La fonction **ld.compile_tex_default(options)** permet de modifier les réglages par défaut, ces paramètres sont (avec leur valeur par défaut) :

- `pdflatexcmd="pdflatex"`,
- `pstoeditcmd="pstoedit"`,
- `pdf2pscmd="pdf2ps"`,
- `preamble="\\documentclass[12pt]{article}\n"`,
- `usepackage="\\usepackage{amsmath,amssymb}\n\\usepackage{fourier}\n"`.

1. Ce module est une contribution de Christophe BAL.

En fonction de votre système d'exploitation, il vous faudra peut être modifier ces variables pour ajouter le chemin d'accès au programme, par exemple :

```
ld.compile_tex_default( {pstoeditcmd="/usr/bin/pstoedit"} )
```

Ce module permet de :

1. compiler un fragment de texte en TEX,
2. de convertir le fichier obtenu en un fichier *eps* contenant du *flattened postscript*,
3. de lire le contenu du fichier *eps* et renvoyer son contenu sous forme d'une liste de chemins, avec l'épaisseur de ligne en tête de chaque chemin, et l'instruction de remplissage à la fin.
4. La liste ainsi obtenue peut être :
   (a) dessinée à l'écran,
   (b) convertie en chemins 3D dans un plan donné et être dessinée,
   (c) convertie en lignes polygonales 3D dans un plan donné (on perd alors l'épaisseur et la commande de remplissage) et être dessinée.

Première partie : compilation et lecture

Attention : cette étape nécessite une compilation du document avec l'option *-shell-escape* ou *-enable-write18*. Sans cette option, le fragment ne sera pas compilé, ce qui n'est pas un problème si le fichier *<filename>.eps* existe déjà et que l'on ne souhaitait pas le modifier.

La première étape est le rôle de la fonction **ld.compile_tex(text [, filename, conv_stroke])**, l'argument ⟨*text*⟩ est une chaîne de caractères, c'est le fragment à compiler, l'argument optionnel ⟨*filename*⟩ est aussi une chaîne de caractères, c'est le nom du fichier qui sera créé, ce nom ne doit contenir **ni chemin, ni extension**, par défaut ce nom est *"tex2FlatPs"*, il est créé dans le dossier courant (mais sera ensuite effacé). L'argument optionnel ⟨*conv_stroke*⟩ est un booléen qui indique si les lignes polygonales (instructions *stroke*) doivent être converties en bandes remplies avec un *fill*, ou pas (`false` par défaut).

Le processus se déroule en plusieurs étapes :

1. création du fichier tex. Celui-ci utilise les paramètres `preamble` et `usepackage`. La compilation se fait avec *pdflatex*.
2. Le fichier obtenu est transformé en postscript avec l'utilitaire *pdf2ps*.
3. Le fichier *ps* obtenu est à son tour transformé avec l'utilitaire *pstoedit* en un fichier *eps* en *flattened postscript* (tout le contenu est sous forme de chemins).
4. Le fichier *<filename>.eps* ainsi obtenu est copié dans le dossier de travail de *luadraw* (le nom de ce dossier est dans la variable globale `ld.cachedir`), et tous les résidus de la compilation sont effacés.
5. Le contenu du fichier ainsi créé est automatiquement lu par la fonction **ld.read_compiled_tex(filename)**, qui renvoie une liste de chemins, chaque chemin est une liste commençant par l'épaisseur de ligne, suivi de points et d'instructions comme un chemin ordinaire, et se terminant par la commande de remplissage (*"fill"*, ou *"eofill"* ou *"stroke"*).

Deuxième partie : exploitation du résultat

En 2D Le résultat peut être dessiné avec la méthode **g:Dcompiled_tex(L, anchor, options)** où ⟨*L*⟩ est le résultat renvoyé par la fonction **compile_tex()**. L'argument ⟨*anchor*⟩ est un nombre complexe, il représente le centre de la boîte englobante du dessin contenu dans ⟨*L*⟩. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `pos="center"`, indique la position du contenu de ⟨*L*⟩ par rapport au point d'ancrage, les valeurs possibles sont les mêmes que pour l'option `pos` des labels, c'est-à-dire : `"center"`, `"N"`, `"NE"`, `"E"`, `"SE"`, `"S"`, `"SW"`, `"W"`, `"NW"`,
- `scale=1`, permet de jouer sur la taille du dessin, cette option peut être un nombre ou bien une table de deux nombres : *{scaleX, scaleY}*,
- `color=<couleur courante par défaut>`,
- `dir=nil`, direction de l'écriture (`nil` signifie le sens habituel). De manière générale, c'est une table constituée de deux vecteurs `dir={v1,v2}` indiquant le sens de l'écriture,

- `hollow=false`, permet d'activer ou désactiver le remplissage des formes. Avec la valeur `true` seuls les contours sont dessinés,
- `drawbox=false` : permet de dessiner ou non la boîte englobante,
- `draw_options=""` : chaîne contenant les options qui seront passées directement à la commande *\draw*.

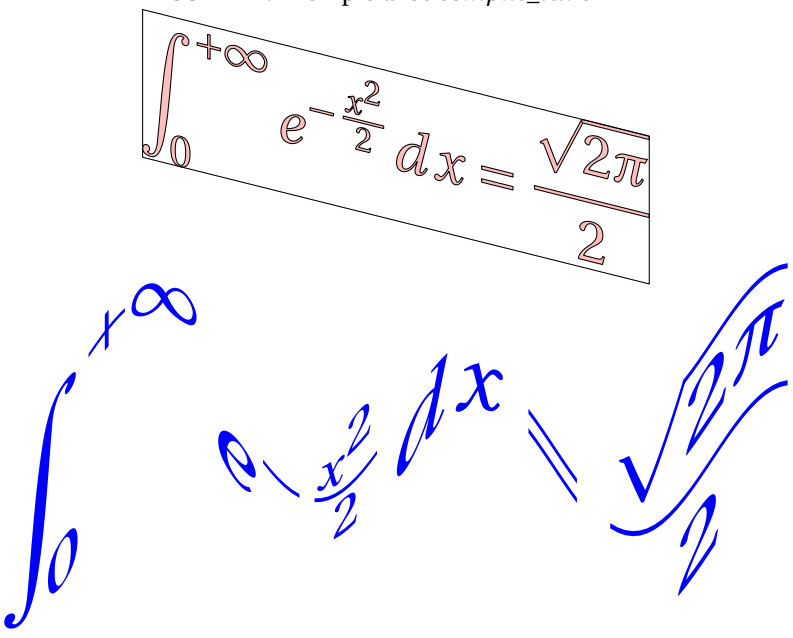
Le résultat peut être transformé en ligne polygonale (liste de listes de complexes) avec la fonction **ld.compiled_tex2polyline(L [, scale])** où ⟨*L*⟩ est le résultat renvoyé par la fonction **ld.compile_tex()**. L'argument optionnel ⟨*scale*⟩ permet de jouer sur la taille, ce peut être un nombre ou bien une table de deux nombres : *{scaleX, scaleY}*.

```
\begin{luadraw}{name=compile_tex2d}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{bbox=false}
require 'luadraw_compile_tex'
local i = cpx.I
local text = "\\[\\int_0^{+\\infty} e^{-\\frac{x^2}2}dx = \\frac{\\sqrt{2\\pi}}2\\]" -- text to compile
-- compiler avec l'option -shell-escape une première fois pour créer le fichier gauss_integral.eps
local L = ld.compile_tex(text,"gauss_integral",true) -- avec le true les lignes (stroke) sont changées en bandes
g:Shift(2*i) -- un premier dessin
g:Dcompiled_tex(L,0,{scale=2,hollow=true, drawbox=true, draw_options="fill=pink", dir={1-i/4,i}}) -- on dessine L

g:Shift(-4*i) -- un second dessin
L = ld.compiled_tex2polyline(L,{3,3}) -- L est convertie en ligne polygonale
local f = function(z) return Z(z.re,z.im+math.sin(z.re*1.5)) end  -- cette fonction produit des vagues sinusoïdales
L = ld.ftransform(L,f) -- on applique la fonction f à L
g:Dpath( ld.polyline2path(L), 'draw=none,fill=blue') -- on dessine L en tant que chemin
g:Show()
\end{luadraw}
```

FIGURE 11 : Exemple avec *compile_tex* en 2D



En 3D Le résultat peut être converti en 3D avec la méthode **g:Compiled_tex2path3d(L, options)** où ⟨*L*⟩ est le résultat renvoyé par la fonction **ld.compile_tex()**. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `scale=1`, permet de jouer sur la taille du dessin, cette option peut être un nombre ou bien une table de deux nombres : *{scaleX, scaleY}*,
- `anchor=pt3d.Origin`, point 3D qui représente le centre de la boîte englobante du dessin,
- `pos="center"`, indique la position du contenu de ⟨*L*⟩ par rapport au point d'ancrage, les valeurs possibles sont les mêmes que pour l'option `pos` des labels, c'est-à-dire : `"center"`, `"N"`, `"NE"`, `"E"`, `"SE"`, `"S"`, `"SW"`, `"W"`, `"NW"`,
- `color=<couleur courante par défaut>`,

- `dir={pt3d.vecJ,pt3d.vecK}`, base du plan dans lequel sera le résultat (ce plan contiendra également le point `anchor`), ces deux vecteurs indiquent le sens de l'écriture,
- `polyline=false`, avec la valeur `true` le résultat renvoyé sera une liste de listes de points 3D et pourra donc être dessiné avec la méthode **g:Dpolyline3d()**, par contre les informations : épaisseur de ligne et commande de remplissage, sont perdues. Avec la valeur `false` le résultat est une liste de chemins, chaque chemin est une liste commençant par l'épaisseur de ligne, suivi de points 3D et d'instructions comme un chemin 3D ordinaire, et se terminant par la commande de remplissage (*"fill"*, ou *"eofill"* ou *"stroke"*).

Avec l'option `polyline=false` (valeur par défaut), le résultat envoyé peut être dessiné avec la méthode **g:Dcompiled_tex3d(L, options)** où ⟨*L*⟩ est le résultat de la méthode **g:Compiled_tex2path3d()**. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `color=<couleur courante par défaut>`,
- `hollow=false`, permet d'activer ou désactiver le remplissage des formes. Avec la valeur `true` seuls les contours sont dessinés.
- `drawbox=false` : permet de dessiner ou non la boîte englobante,
- `draw_options=""` : chaîne contenant les options qui seront passées directement à la commande *\draw*.

```
\begin{luadraw}{name=compile_tex3d}
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Origin, vecI, vecJ, vecK, M, Mc = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M, pt3d.Mc

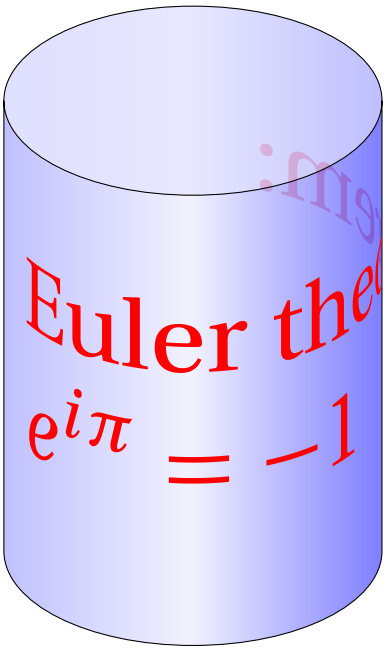
local g = ld.graph3d:new{ window={-3,3,-4,4}, margin={0,0,0,0}, size={10,10}, viewdir={-50,60}}
require 'luadraw_compile_tex'

function curve_on_cylinder(curve,cylinder,screenNormal)
-- curve est une ligne polygonale sur le cylindre,
-- cylinder = {A,r,B} estle cylindre
-- cette fonction sépare la partie visible de a partie cachée de la courbe
    local A,r,B = table.unpack(cylinder)
    local U = B-A
    local visibility_function = function(N)
        local I = ld.dproj3d(N,{A,U})
        return (pt3d.dot(N-I,screenNormal) >= 0)
    end
    return ld.split_points_by_visibility(curve,visibility_function)
end

local A, r, B = -3*vecK, 2, 2.5*vecK -- the cylinder
local text = "Euler theorem: \\par \\(e^{i\\pi}=-1\\)"
local L = ld.compile_tex(text, "essai") -- compiler avec -shell-escape la 1ière fois pour créer le fichier "essai.eps"
local C = g:Compiled_tex2path3d(L,{scale=3, anchor=M(r,0,0), dir={vecJ,vecK}, polyline=true})
-- C est le texte converti en ligne polygonale 3D sur le plan passant par anchor, de base (vecJ,vecK) et grossie 3 fois
 ↪  (scale)

local f = function(A) return Mc(r,A.y/r,A.z) end -- renvoie l'image du point sur le cylindre par enroulement
C = ld.ftransform3d(C,f) -- conversion: courbe plane -> courbe cylindrique
local Cv, Ch = curve_on_cylinder(C, {A,r,B}, g.Normal) -- parties visible et cachée de C (peut être un peu long)
Ch = ld.polyline2path3d(Ch) -- partie cachée, conversion en chemins
g:Dpath3d(Ch, "draw=none,fill=red!30")
g:Dcylinder(A,r,B,{color="blue",opacity=0.5})
Cv = ld.polyline2path3d(Cv) -- partie visible, conversion en chemins
g:Dpath3d(Cv, "draw=none,fill=red")
g:Show()
\end{luadraw}
```

FIGURE 12 : Écrire sur un cylindre



5) Le module *luadraw_cvx_polyhedra_nets*

Ce module ajoute une méthode graphique à la classe *ld.graph3d*, ainsi que plusieurs fonctions dans l'espace de noms *luadraw*, mais il ne renvoie rien.

La fonction de base

Le module *luadraw_cvx_polyhedra_nets* permet de « déplier » un polyèdre **convexe** afin d'en obtenir un patron. La fonction réalisant le dépliage est :

ld.unfold_polyhedron(P, options)

L'argument ⟨*P*⟩ doit être un polyèdre convexe. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `opening=1`, valeur comprise entre 0 et 1 représentant le "taux" d'ouverture. Avec la valeur 1 le polyèdre est totalement déplié, les facettes renvoyées par la fonction seront donc toutes dans un même plan. Avec la valeur 0, la fonction renvoie les facettes du polyèdre sans modification.
- `root=1`, numéro de la facette du polyèdre qui servira de racine, car la fonction met le polyèdre sous la forme d'un arbre en déterminant pour chaque facette quelles sont les voisines (facettes adjacentes) ainsi que les arêtes communes et les angles. Cette option permet de choisir la facette qui servira de point de départ.
- `model=nil`, liste de listes de numéros de facettes pour imposer un modèle de patron, par exemple :

  `model={ {1,6},{1,3},{1,4},{1,5,2}}`,

  la sous-liste *{1,5,2}* signifie que la facette 1 est l'ancêtre de la facette 5, et que la facette 5 est l'ancêtre de la facette 2, c'est à dire que les facettes 5 et 1 sont adjacentes, et la facette 5 tournera autour de son arête commune avec la facette 1 (même chose 5 et 2). Pour que le modèle soit cohérent, toutes les facettes du polyèdre SAUF une (qui sera la facette `root`), doivent avoir un et un seul ancêtre ; si dans le polyèdre les facettes 1 et 5 ne sont pas adjacentes, la fonction s'arrête et affiche une erreur dans le terminal. Lorsque l'option `model` vaut `nil` (valeur par défaut), l'algorithme calcule lui-même un modèle cohérent.
- `to2d=false`, booléen qui permet d'obtenir une version 2D du patron dans le repère du plan de l'écran. Avec la valeur `true`, l'option `opening` prend automatiquement la valeur 1 et les facettes renvoyées par la fonction auront des sommets exprimés en nombres complexes.
- `tabs=false`, booléen qui permet d'ajouter ou non des languettes au patron dans la version 2D. Avec la valeur `true`, l'option `to2d` prend automatiquement la valeur `true` également, les facettes renvoyées par la fonction auront

des sommets exprimés en nombres complexes dans le repère de l'écran, et la fonction renvoie en plus une ligne polygonale 2D représentant des languettes pour certaines arêtes (celles-ci sont déterminées automatiquement).

- `tabs_wd=0.2`, valeur numérique représentant l'épaisseur des languettes lorsque l'option `tabs` a la valeur `true`.
- `tabs_lg=0.5`, valeur numérique entre 0 et 1, permettant de déterminer la longueur du petit côté des languettes celle-ci est égale à la longueur de l'arête (qui est le grand côté) multipliée par `tabs_lg` (lorsque l'option `tabs` a la valeur `true`).
- `rotate=0`, lorsque l'option `to2d` a la valeur `true`, le dessin est tourné d'un angle égal à `rotate` (en degrés) autour de son centre. Dans la version 3D, le dessin est tourné d'un angle égal à `rotate` (en degrés) autour de l'axe passant par le centre de gravité de la facette `root` et orienté par un vecteur normal à cette facette dirigé vers l'extérieur du polyèdre.

La fonction renvoie en résultat une table constituée des champs suivants :

- Le champ *facets* : qui contient la liste des facettes, avec les sommets en 2D (nombres complexes) si l'option `to2d` vaut `true`, ou sommets en 3D (points 3D) dans le cas contraire.
- Le champ *tree* : qui est une liste de la forme :

  { {ancestor, n1, n2, angle, vertices}, …}

  chaque élément de cette liste représente une facette avec pour chacune d'elles les informations suivantes :
  - *ancestor* : numéro de la facette ancêtre, c'est son rang dans la liste *tree* (la facette qui a servi de racine a pour ancêtre le numéro 0 qui ne correspond à aucune facette).
  - *n1, n2* : numéro des sommets la facette ancêtre représentant l'arête commune.
  - *angle* : angle en degré avec la facette ancêtre.
  - *vertices* : liste des sommets (points 3D) de la facette.
- Le champ *bounds* : qui contient sous la forme d'une liste, la bounding box des facettes (soit en 2D soit en 3D)
- Lorsque l'option `tabs` a la valeur `true`, alors il y a deux champs supplémentaires dans le résultat :
  - Le champ *tabs* : qui contient une ligne polygonale 2D (liste de listes de nombres complexes) représentant les languettes, ceci uniquement lorsque l'option `tabs` a la valeur `true`.
  - Le champ *twins* : qui contient une liste de la forme *{ {{a1,b1}, {a2,b2}}, …}* représentant la liste des paires d'arêtes jumelles (les arêtes jumelles sont confondues lorsque le polyèdre est refermé), *a1*, *b1*, *a2*, *b2*, sont des nombres complexes représentant les extrémités des arêtes dans le patron du polyèdre version 2D. Cette liste est calculée uniquement lorsque l'option `tabs` a la valeur `true`.

La méthode de dessin

Celle-ci est la méthode **g:Dpolyhedron_net(P, options)** où ⟨*P*⟩ désigne un polyèdre convexe. Les options sont celles de la fonction précédentes, à celles-ci s'ajoutent :

- Dans le cas d'un patron 2D (lorsque l'option `to2d`, ou l'option `tabs`, a la valeur `true`) :
  - `facet_name=false`, avec la valeur `true` le numéro des facettes (précédé de la lettre 'F') sera affiché au centre de chaque facette.
  - `edge_name=false`, avec la valeur `true` le numéro des arêtes (précédé de la lettre 'e') sera affiché au centre de chaque arête, ce qui permet de repérer les arêtes jumelles et donc les facettes voisines.
  - `tabs_options=""`, chaîne représentant des options de dessin TikZ pour les languettes si l'option `tabs` a la valeur `true`.
  - `facet_options=""`, chaîne représentant des options TikZ de dessin pour la méthode **g:Dpolyline()** qui dessinera les facettes.
- Dans le cas d'un patron 3D il y a uniquement en plus :
  - `facet_options={}`, liste d'options de dessin pour la méthode **g:Dfacet()** qui dessinera les facettes.

Le dessin est accompagné d'un affichage dans le terminal de la bounding box 2D de celui-ci.

Exemples

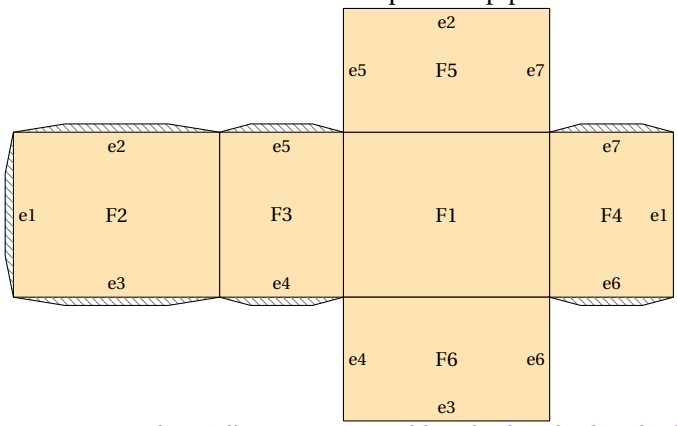
Dans cet exemple, on affiche le patron par défaut, version 2D, d'un parallélépipède P avec les languettes hachurées, le numéro des facettes (ce numéro est le rang dans la liste *P.facets*) ainsi que le numéro des arêtes afin de voir celles qui doivent être collées ensemble :

```
\begin{luadraw}{name=parallelep_net}
```

```
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Origin, vecI, vecJ, vecK = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK

local g = ld.graph3d:new{viewdir={30,60},window={-8.5,8,-5,5},bbox=false, size={10,10}}
require 'luadraw_cvx_polyhedra_nets'
P = ld.parallelep(Origin, 4*vecI,5*vecJ,3*vecK)
g:Dpolyhedron_net(P, {tabs=true, tabs_options="pattern=north west lines, pattern color=gray",
  facet_options="fill=Orange!30", facet_name=true, edge_name=true})
g:Show()
\end{luadraw}
```

FIGURE 13 : Patron d'un parallélépipède

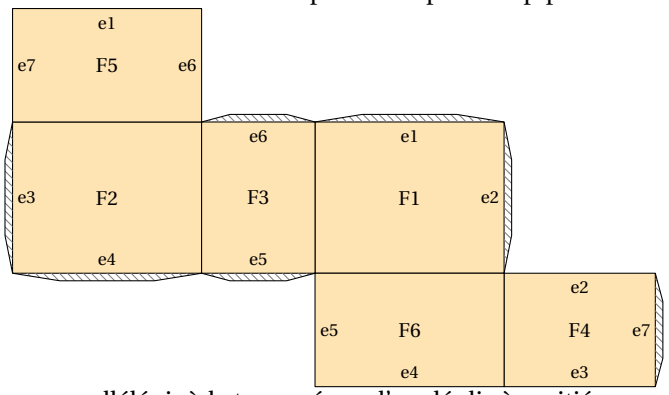


Le patron par défaut ici correspondrait à l'option `model={{1,3},{1,4},{1,5},{1,6},{3,2}}`[2], mais on peut vouloir imposer un autre modèle, par exemple, avec le même parallélépipède :

```
\begin{luadraw}{name=parallelep_net2}
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Origin, vecI, vecJ, vecK = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK

local g = ld.graph3d:new{viewdir={30,60},window={-9,9,-5,5},bbox=false,size={10,10}}
require 'luadraw_cvx_polyhedra_nets'
P = ld.parallelep(Origin, 4*vecI,5*vecJ,3*vecK)
g:Dpolyhedron_net(P, {model={{4,6,1,3,2,5}},tabs=true,
  tabs_options="pattern=north west lines, pattern color=gray",
  facet_options="fill=Orange!30", facet_name=true,
 edge_name=true, rotate=-90})
g:Show()
\end{luadraw}
```

FIGURE 14 : Patron imposé d'un parallélépipède



Voici un exemple avec un parallélépipède tronqué que l'on déplie à moitié :

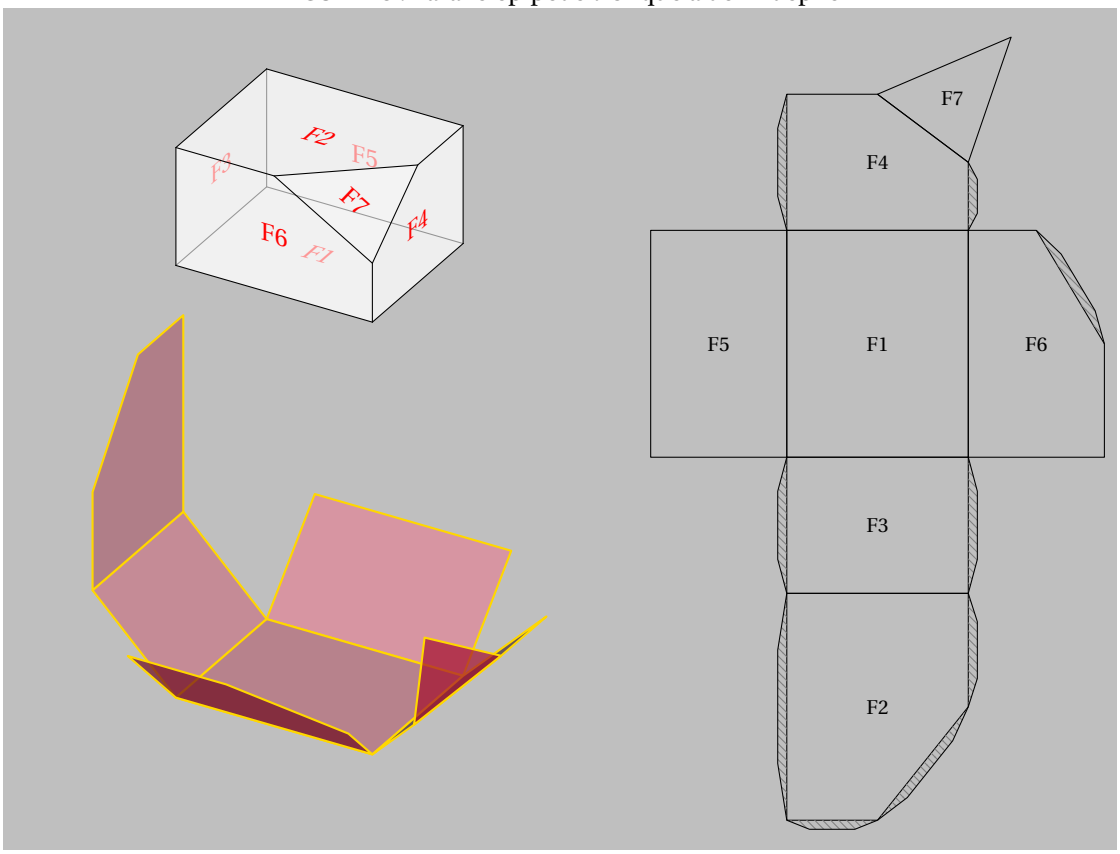
```
\begin{luadraw}{name=parallelep_net3}
local ld = luadraw
```

2. L'algorithme prend la première facette, puis cherche ses voisines, puis les voisines de la première voisine, etc.

```
local cpx, pt3d = ld.cpx, ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window={-9,15,-9,9,0.6,0.6},bg="lightgray", viewdir={30,60}, margin={0,0,0,0}}
require 'luadraw_cvx_polyhedra_nets'
P = ld.parallelep(Origin, 4*vecI,5*vecJ,3*vecK)
local A, B, C = M(4,2.5,3), M(2,5,3), M(4,5,1.5)
P = ld.cutpoly(P, ld.plane(A,B,C), true) -- P est tronqué avec le plan
g:Shift3d(M(0,-4,5))
g:Dpolynames(P,"facet") -- cette méthode montre les numéros des facettes de P
-- demi-ouverture de P
g:Shift3d(M(0,0,-11))
g:Dpolyhedron_net(P,{opening=0.5, facet_options={color="Crimson", opacity=0.7, edgecolor="Gold", edgewidth=8}})
-- patron 2D
g:Shift(10)
g:Dpolyhedron_net(P,{tabs=true, tabs_options="pattern=north west lines, pattern color=gray",
 facet_name=true, rotate=90})
g:Show()
\end{luadraw}
```

FIGURE 15 : Parallélépipède tronqué à demi déplié



NB : Les fonctions **ld.unfold_polyhedron()** et **g:Dpolyhedron_net()** s'appliquent à tout polyèdre convexe, mais elles ne donneront pas le résultat escompté avec un polyèdre non convexe.

La fonction *unfold_tree()*

Il peut être utile de récupérer l'arbre fabriqué par la fonction **ld.unfold_polyhedron()** afin d'éviter de recalculer celui-ci plusieurs fois, lors d'une animation par exemple. La fonction **ld.unfold_tree(tree, opening [, num])** permet aussi de déplier le polyèdre. L'argument⟨*tree*⟩ est l'arbre fourni par la fonction **ld.unfold_polyhedron()**, l'argument optionnel ⟨*opening*⟩ est un nombre entre 0 et 1 qui représente le taux d'ouverture (1 par défaut), l'argument optionnel ⟨*num*⟩ est le numéro de la facette que l'on souhaite ouvrir (et toute la descendance de la facette tournera de la même façon), lorsque cet argument est omis, toutes les facettes tournent.

Exemple d'animation :

```
\begin{luacode*}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK

ld.nbimages = 70
-- images creation
local g = ld.graph3d:new{ viewdir={"central",30,60}, bg="gray", size={10,10}, margin={0,0,0,0} }
-- declarations
local poly = require 'luadraw_polyhedrons'
require 'luadraw_cvx_polyhedra_nets'
local p = ld.linspace(0,1,36)
local T = ld.linspace(0,360, ld.nbimages+1)
local P = poly.dodecahedron(Origin, -2*vecI)
local net = ld.unfold_polyhedron(P)
local tree = net.tree
-- create the image number k, this function must be global
function ld.makeframe(k) -- do not modify this line
    local r = k
    if k > 36 then r = 72-k end
    local P1 = ld.rotate3d( ld.unfold_tree(tree,p[r]), T[k],{Origin,vecK})
    g:Dfacet(P1, {color="Crimson", edgecolor="Gold", edgewidth=8})
    -- send image number k
    g:Sendtotex()  -- do not modify
    g:Cleargraph() -- do not modify
end
\end{luacode*}
```

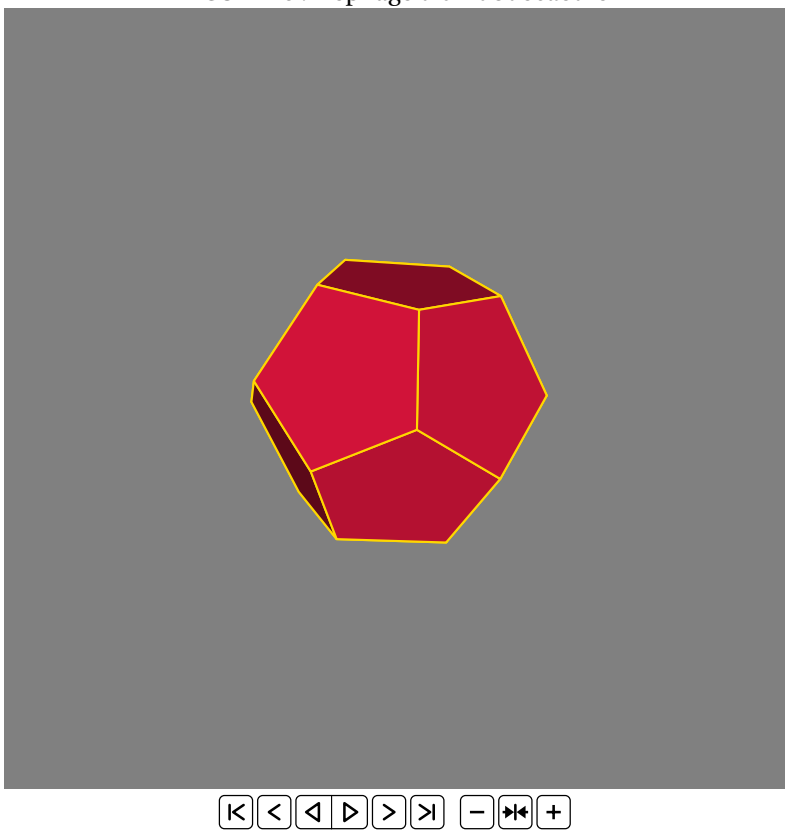
Le code TEX (avec le paquet *animate*) :

```
\newcommand*\nb{\directlua{tex.print(luadraw.nbimages)}}
\newcommand*\makeframe[1]{\directlua{luadraw.makeframe(#1)}}%

\begin{animateinline}[poster=first,controls,loop]{8}
\multiframe{\nb}{ik=1+1}{%
\makeframe{\ik}%
}%
\end{animateinline}
```

Le résultat :

FIGURE 16 : Dépliage d'un dodécaèdre



6) Le module *luadraw_fields*

Ce module ne renvoie rien, il ajoute de nouvelles méthodes graphiques à la classe *ld.graph* et de nouvelles fonctions dans l'espace de noms *luadraw*.

Champs 2D

Ce sont les méthodes qui ont été données à la fin du chapitre 1 pour le dessin des champs de vecteurs et des champs de gradient. On y trouve :

- La fonction **ld.field(f, x1, x2, y1, y2 [, grid, length])** qui renvoie le champ de vecteurs (liste de segments). L'argument $\langle f\rangle$ est une fonction $\langle f\rangle : (x,y)\mapsto f(x,y)\in \mathbf{R}^2$ ($f(x,y)$ est une liste de deux réels). Le champ est calculé sur le pave $[x_1;x_2]\times[y_1;y_2]$. L'argument $\langle grid\rangle$ vaut par défaut $\{25,25\}$, c'est le nombre de subdivisions de l'intervalle $[x_1;x_2]$ et celui de l'intervalle $[y_1;y_2]$. L'argument $\langle length\rangle$ permet d'imposer la longueur des vecteurs (tous les vecteurs ont la même longueur), par défaut une longueur est calculée en fonction des pas sur chacun des deux axes.
- La méthode **g:Dvectorfield(f, options)** fait le dessin du champ de vecteurs associé à la fonction $\langle f\rangle$. L'argument $\langle options\rangle$ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :
  - `view=<fenêtre par défaut>`, liste de la forme `view={x1,x2,y1,y2}` permettant de définir le pavé $[x_1;x_2]\times[y_1;y_2]$. Par défaut c'est la fenêtre choisie lors de la création du graphe.
  - `grid={25,25}`, liste de deux entiers qui permet de définir les subdivisions suivant les deux axes.
  - `length=nil`, permet d'imposer la longueur des vecteurs, par défaut une longueur est calculée en fonction des pas sur chacun des deux axes.
  - `draw_options=""`, chaîne contenant les options de dessin qui seront passées à TikZ.
- La méthode **g:Dgradientfield(f, options)** fait le dessin du champ de gradient associé à la fonction **numérique** $\langle f\rangle : (x,y)\mapsto f(x,y)\in\mathbf{R}$. L'argument $\langle options\rangle$ est identique à celui de la méthode **Dvectorfield()**.

Un exemple de ces deux méthodes a déjà été donné page 79.

Champs 3D

Il s'agit de champs de vecteurs tangents à une surface 3D.
La surface est définie par un paramétrage $p : (u,v)\mapsto p(u,v)\in\mathbb{R}^3$ avec $(u,v)\in[u_1;u_2]\times[v_1;v_2]$.

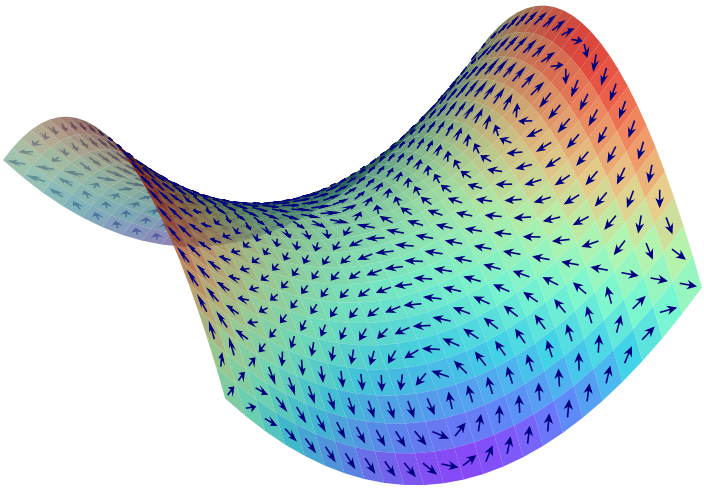
Le champ est définie par une fonction $f : (u,v) \mapsto (f_1(u,v); f_2(u,v)) \in \mathbb{R}^2$, avec $(u,v) \in [u_1;u_2] \times [v_1;v_2]$.
Les vecteurs sont calculés au centre de chaque maille et sont définis par la formule :

$$f_1(u,v)\frac{\partial p}{\partial u}(u,v) + f_2(u,v)\frac{\partial p}{\partial v}(u,v), \text{ puis ceux-ci sont normalisés.}$$

- La fonction **ld.surfacefield(p, f, u1, u2, v1, v2 [, grid, length, arrows])** renvoie le champ de vecteurs (liste de listes de points 3D). L'argument ⟨*p*⟩ est le paramétrage de la surface, l'argument ⟨*f*⟩ définit le champ de vecteurs, les calculs sont faits dans le domaine $[u_1;u_2] \times [v_1;v_2]$. L'argument ⟨*grid*⟩ vaut par défaut {25, 25}, c'est le nombre de subdivisions de l'intervalle $[u_1;u_2]$ et celui de l'intervalle $[v_1;v_2]$. L'argument ⟨*length*⟩ permet d'imposer la longueur des vecteurs (tous les vecteurs ont la même longueur), par défaut une longueur est calculée en fonction des pas sur chacun des deux axes. L'argument ⟨*arrows*⟩ est un booléen (`false` par défaut), avec la valeur `true` une flèche (basique) est ajoutée à l'extrémité de chaque vecteur, celles-ci est dessinée dans le plan tangent à la surface.
- La méthode **g:Dsurfacefield(p, f, options)** fait le dessin du champ de vecteurs associé à la fonction ⟨*f*⟩, et (éventuellement) celui de la surface associé à ⟨*p*⟩. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :
  - `domain={u1,u2,v1,v2}`, permet de définir le domaine $[u_1;u_2] \times [v_1;v_2]$. Par défaut ce sont les intervalles des abscisses et des ordonnées de la fenêtre 3D choisie lors de la création du graphe.
  - `grid={25,25}`, liste de deux entiers qui permet de définir les subdivisions du domaine.
  - `length=nil`, permet d'imposer la longueur des vecteurs, par défaut une longueur est calculée en fonction de la grille choisie.
  - `color=<couleur courante des lignes>`, couleur des vecteurs.
  - `width=<épaisseur courante des lignes>`, épaisseur de trait des vecteurs.
  - `arrows="auto"`, style de flèche pour les vecteurs, par défaut c'est une flèche dessinée dans le plan tangent à la surface. Avec par exemple `arrows="-stealth"` les flèches seront dessinées par TikZ, mais dans le plan de l'écran. Avec `arrows="-"` il n'y aura pas de flèches, seulement des segments.
  - `clip=false`, permet de clipper ou non le dessin avec la fenêtre 3D courante.
  - `field_options=""`, chaîne contenant les options de dessin des vecteurs qui seront passées à TikZ. En principe cette option ne devrait pas être utile.
  - `surface_options=nil`, avec la valeur `nil` la surface est ni calculée, ni dessinée, il n'y aura que les vecteurs. Dans les autres cas, cette option doit être une table d'options pour la méthode **g:Dfacet()** qui dessinera la surface. Notons que si cette table contient l'option `opacity=0` alors les facettes de la surface ne seront pas rendues.

```
\begin{luadraw}{name=surface-quiver}
local ld = luadraw
local pt3d, cpx = ld.pt3d, ld.cpx
local M, Z = pt3d.M, cpx.Z
require 'luadraw_fields'
local surface = function(x, y) return M(x, y, 0.15*x^2 - 0.15*y^2) end
local field = function(x, y) return {math.sin(y), math.sin(x)} end
local g = ld.graph3d:new{ window={-5.5, 6, -6, 4}, viewdir={65, 60}, margin=0, size={10, 10} }
g:Dsurfacefield(surface, field, {
    domain = {-4, 4, -4, 4},
    color = "Navy",
    width = 6,
    surface_options = {mode=ld.mShadedOnly, usepalette={ld.palRainbow,"z"}, opacity=0.8}
    })
g:Dlabel("$z=0.15x^2-0.15y^2$,\\quad $f(x,y)=(\\sin y,\\sin x)$", Z(0,-4.5), {pos="S"})
g:Show()
\end{luadraw}
```

FIGURE 17 : La méthode *g:Dsurfacefield()*

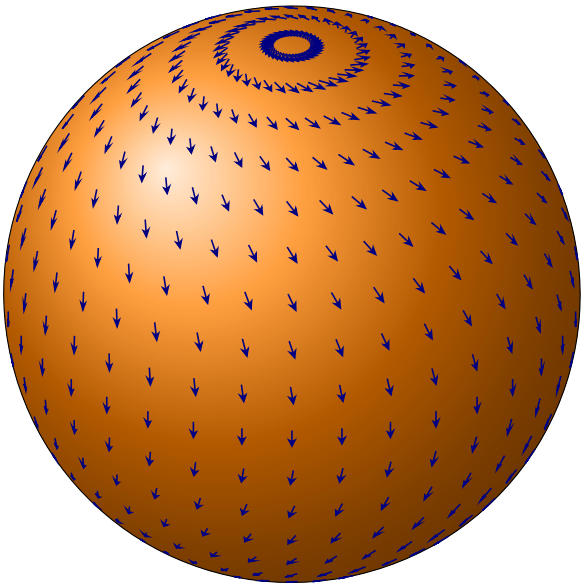


$$z = 0.15x^2 - 0.15y^2, \quad f(x,y) = (\sin y, \sin x)$$

Dans le second exemple, on utilise le calcul de la surface pour éliminer les facettes non visibles (et donc les vecteurs associés également) avec l'option `backcull=true`, mais on ne dessine pas la surface, grâce à l'option `opacity=0`, pour mettre à la place une version plus jolie de la sphère.

```
\begin{luadraw}{name=surface-quiver2}
local ld = luadraw
local pt3d, cpx = ld.pt3d, ld.cpx
local M, Ms, Z, sin, cos, pi = pt3d.M, pt3d.Ms, cpx.Z, math.sin, math.cos, math.pi
require 'luadraw_fields'
local surface = function(u,v) return Ms(4,u,v) end
local field = function(u,v) return {cos(v)/(1-cos(u)), sin(v)/(1-cos(u))} end
local g = ld.graph3d:new{ window={-5, 5.5, -5.5, 5}, viewdir={65, 60}, margin=0, size={10, 10} }
g:Dsphere( M(0,0,0), 4, {color="orange", mode=ld.mBorder})
g:Dsurfacefield(surface, field, {
    domain = {pi, -pi, 0, pi},
    grid = {37,20},
    color = "Navy",
    width = 6,
    surface_options = {backcull=true, opacity=0}
    })
g:Dlabel("$f(u,v)= (\\frac{\\cos(v)}{1-\\cos(u)},\\frac{\\sin(v)}{1-\\cos(u)})$", Z(0,-4.5), {pos="S"})
g:Show()
\end{luadraw}
```

FIGURE 18 : Sur une sphère



$$f(u,v) = \left(\frac{\cos(v)}{1-\cos(u)}, \frac{\sin(v)}{1-\cos(u)}\right)$$

7) Le module *luadraw_shadedforms*

Ce module ne renvoie rien, il ajoute de nouvelles méthodes graphiques à la classe *ld.graph*. Il permet de dessiner des lignes polygonales ou de remplir une forme en utilisant un gradient de couleurs.

Dshadedpolyline

La méthode **g:Dshadedpolyline(L, palette, options)** permet de dessiner la ligne polygonale 2D $\langle L\rangle$ avec un dégradé de couleurs en fonction de la méthode de calcul et de la $\langle palette\rangle$ choisies. $\langle L\rangle$ est une liste de nombres complexes ou une liste de listes de nombres complexes, $\langle palette\rangle$ est une liste de couleurs, chaque couleur est elle-même une liste de trois nombres entre 0 et 1 représentant les trois composantes : rouge, vert et bleu de la couleur. L'argument $\langle options\rangle$ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `values="x"`, pour chaque point de $\langle L\rangle$ on calcule une valeur numérique qui permettra de déterminer la couleur de ce point dans la palette choisie. C'est l'option `values` qui détermine le mode de calcul, cette option peut être égale à :
  - `"x"` (valeur par défaut), dans ce cas pour chaque point de $\langle L\rangle$ la valeur sera l'abscisse.
  - `"y"`, dans ce cas pour chaque point de $\langle L\rangle$ la valeur sera l'ordonnée.
  - une fonction `f` : $(x,y)\mapsto f(x,y)\in\mathbf{R}$, dans ce cas chaque point (x,y) de $\langle L\rangle$ la valeur sera donnée par $f(x,y)$.
- `width=<épaisseur courante>`, permet de définir l'épaisseur de ligne en dixième de point.
- `close=false`, booléen indiquant si la ligne polygonale doit être refermée ou non.
- `clip=nil`, cette option est soit `nil` (valeur par défaut), soit une table *{x1,x2,y1,y2}*, dans le premier cas la ligne est clippée par la fenêtre 2D courante **après** sa transformation par la matrice 2D du graphe, dans le second cas la ligne est clippée par la fenêtre $[x_1;x_2]\times[y_1;y_2]$ **avant** d'être transformée par la matrice du graphe.

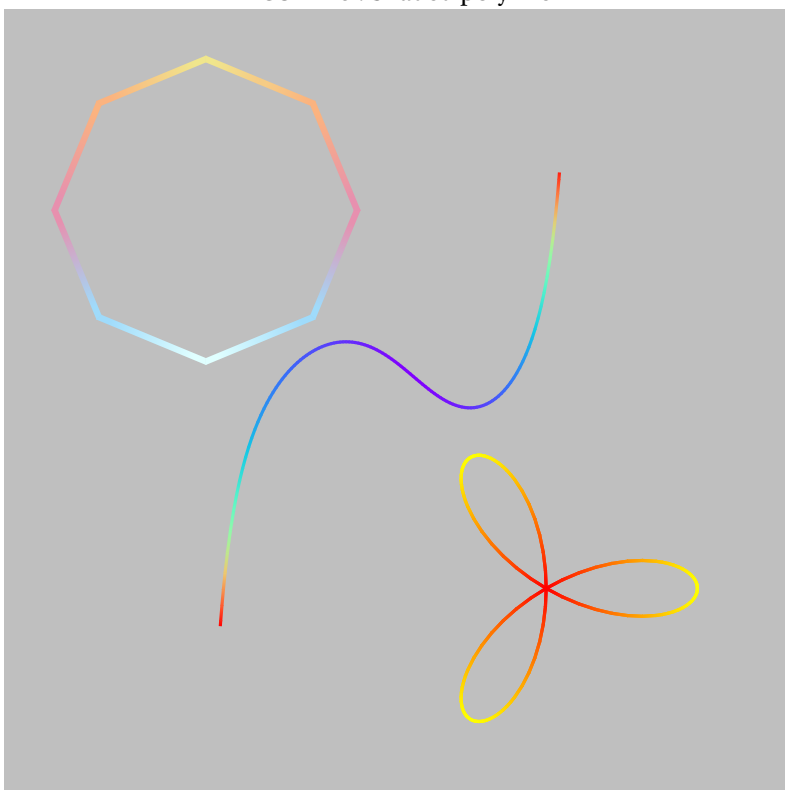
Cette méthode convertit $\langle L\rangle$ en une succession de trapèzes qui sont ensuite remplis avec un dégradé de couleur.

```
\begin{luadraw}{name=shading_polyline}
local ld = luadraw
local cpx = ld.cpx
local i, Z = cpx.I, cpx.Z

local g = ld.graph:new{size={10,10},bg="lightgray", margin={0,0,0,0}}
require 'luadraw_shadedforms'
-- 1ier exemple equation diff. y'= x^2+y^2-1 (=f(x,y))
local x1,x2,y1,y2 = -3,3,-3,3
local A = Z(0,1/2) -- condition initiale
```

```
local f = function(x,y)
    return x^2+y^2-1
end
local S = ld.odesolve(f, A.re, A.im, x1, x2, 150) -- S est une matrice {X,Y}
local L = {} -- convertit {X,Y} en une liste de complexes L
for k = 1, #S[1] do table.insert(L, Z(S[1][k],S[2][k])) end
L = ld.clippolyline(L,-2.5,2.4,y1,y2)[1] -- L est la courbe solution (approchée)
g:Dshadedpolyline(L, ld.palRainbow, {values=f, width=12}) -- la couleur est calculée avec la fonction f
-- 2ième exemple
L = ld.polar(function(t) return 2*math.cos(3*t) end, -math.pi, math.pi)
local f = function(x,y) return cpx.abs(Z(x,y)) end -- ici la valeur sera le module
g:Shift(2-2.5*i)
g:Dshadedpolyline(L, ld.palAutumn, {values=f, width=12})
-- 3ième exemple
g:Shift(-4.5+5*i)
g:Dshadedpolyline( ld.polyreg(0,2,8), ld.palGasFlame, {values="y", width=24, close=true})
g:Show()
\end{luadraw}
```

FIGURE 19 : Shaded polyline



Dcolorbar

La méthode **g:Dcolorbar(A, pal, options)** permet de dessiner un rectangle avec un gradient de couleurs à partir d'une palette, et éventuellement une graduation. L'argument ⟨*A*⟩ est un nombre complexe, c'est le point de référence pour la construction du rectangle. L'argument ⟨*pal*⟩ est une palette de couleurs (liste de listes de la forme $\{r,g,b\}$ avec r, g et b entre 0 et 1). L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `minmax={0,1}`, liste contenant la valeur minimale (qui sera affectée à la première couleur de la palette) et la valeur maximale (qui sera affectée à la dernière couleur de la palette), ainsi, à toute valeur numérique entre *min* et *max* correspond une couleur de la palette.
- `dir=cpx.I`, direction du grand côté du rectangle (ce vecteur est automatiquement normalisé), par défaut le rectangle est donc vertical.
- `length=8`, longueur du rectangle.
- `width=0.5`, largeur du rectangle. Les sommets du rectangle sont :

  `A, A+length*dir, A+length*dir+width*cpx.I*dir, A+width*cpx.I*dir`
- `values=0`, cette option permet de définir soit le nombre de valeurs numériques affichées équiréparties dans l'inter-

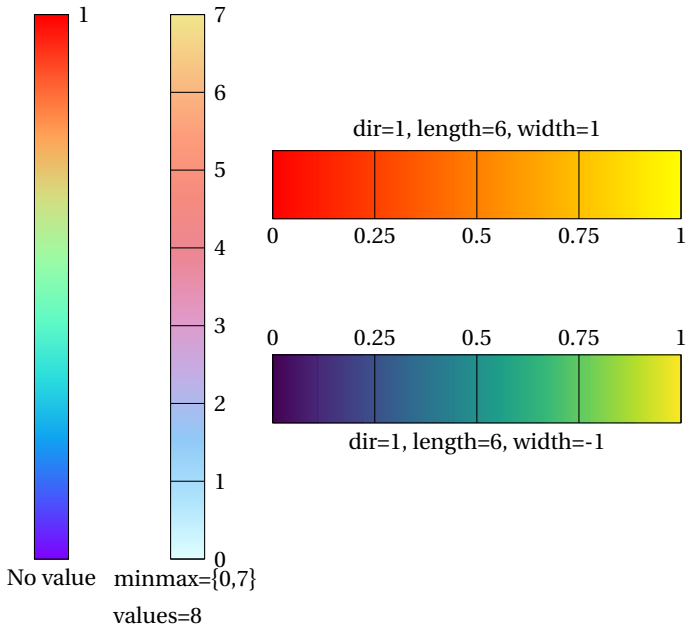
valle `minmax` (aucune par défaut), soit la liste des valeurs numériques affichées (dans ce cas `values` doit être une liste de valeurs numériques comprises dans l'intervalle `minmax`).

- `addvalues=nil`, cette option permet de définir une liste des valeurs numériques à afficher en plus des valeurs *min* et *max*. Cette option prime sur la précédente.
- `digits=2`, nombre de décimales pour les affichages numériques.
- `labelpos="E"`, permet de positionner les labels par rapport aux points d'ancrage (`"N"`, `"NE"`, `"E"`, `"SE"`, `"S"`, `"SW"`, "W", `"NW"`). Les labels sont positionnés le long de l'axe *(A, dir)*.

```
\begin{luadraw}{name=Dcolorbar}
local ld = luadraw
local Z = ld.cpx.Z

local g = ld.graph:new{size={10,10}, bbox=false}
g:Labelsize("small")
require 'luadraw_shadedforms'
local pal = require 'luadraw_palettes'
g:Dcolorbar(Z(-4,-4), pal.Rainbow)
g:Dcolorbar(Z(-2,-4), pal.GasFlame, {minmax={0,7},values=8})
g:Dcolorbar(Z(-1,1), pal.Autumn, {dir=1,length=6,width=1,addvalues={0.25,0.5,0.75},labelpos="S"})
g:Dcolorbar(Z(-1,-1), pal.Viridis, {dir=1,length=6,width=-1,addvalues={0.25,0.5,0.75},labelpos="N"})
g:Dlabel("No value", Z(-4.25,-4),{pos="S"},
    "\\parbox{1.95cm}{minmax=\\{0,7\\}\\\\values=8}", Z(-2.25,-4), {},
    "dir=1, length=6, width=1", Z(2,2), {pos="N"},
    "dir=1, length=6, width=-1", Z(2,-2),{pos="S"})
g:Show()
\end{luadraw}
```

FIGURE 20 : Color bars



Dshadedrectangle

NB : l'utilisation de cette méthode avec l'option `bilinear=true` nécessite la librairie *shadings*.

La méthode **g:Dshadedrectangle(x1, x2, y1, y2, pal, options)** permet de remplir le rectangle $[x_1;x_2]\times[y_1;_y 2]$ avec un gradient de couleurs extraites de la palette ⟨*pal*⟩. Chaque point (x,y) du rectangle a une couleur issue de la palette, calculée à partir d'une valeur $f(x,y)$ où f est une fonction à valeurs numériques définie sur le rectangle. L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

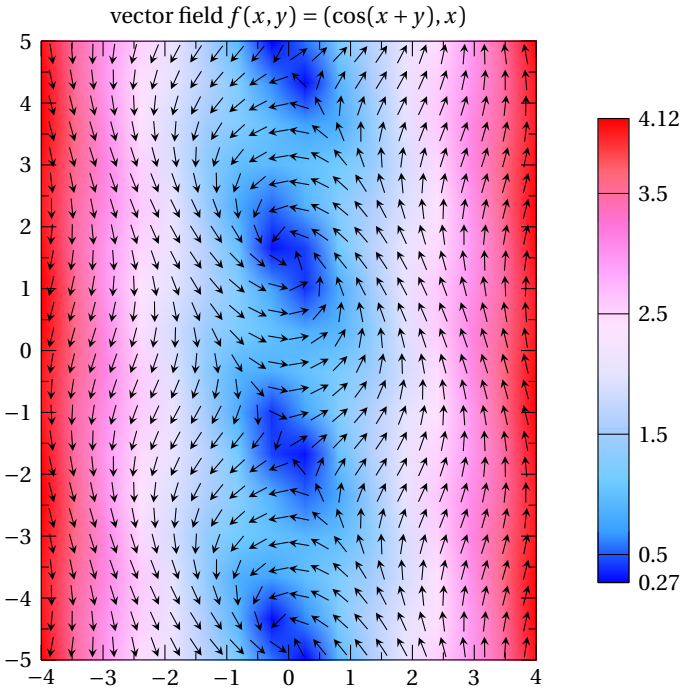
- `values=function(x,y) return cpx.abs(Z(x,y)) end`, cette option définit la fonction f utilisée pour calculer la couleur de chaque point. La valeur maximale de f sur le rectangle correspondra à la dernière couleur de la palette, et la valeur minimale de f correspondra à la première couleur de la palette. Par défaut, cette fonction est le module.
- `grid={15,15}`, cette option définit le nombre de subdivisions pour l'intervalle $[x_1;x_2]$ et pour l'intervalle $[y_1;y_2]$. Plus la subdivision est fine plus l'affichage sera long.

- `bilinear=true`, avec la valeur `true` (valeur par défaut) un ombrage bilinéaire est utilisé sur chaque maille. Avec la valeur `false`, chaque maille est peinte de manière unie.
- `bar="none"`, permet d'ajouter ou non une légende en faisant appel à la méthode **Dcolorbar()**, cette option peut valoir : `"none"`, `"right"`, `"bottom"`, `"left"`, ou `"top"`.
- `bardist=1`, distance entre le rectangle et la légende s'il y en a une.
- `baroptions={}`, liste d'options pour la méthode **Dcolorbar()** s'il y a une légende.
- `out=nil`, si on affecte une variable de type liste à ce paramètre `out`, alors la méthode ajoute à cette liste les deux valeurs *min* et *max* de la fonction f (ce qui permet de récupérer éventuellement ces deux valeurs).

```
\begin{luadraw}{name=Dshadedrectangle}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{window={-4,6,-5.5,5.5}, size={10,10}, bbox=false}
require 'luadraw_shadedforms'
require 'luadraw_fields'
local pal = require 'luadraw_palettes'
g:Labelsize("small")
local f = function(x,y) return {math.cos(x+y), x} end -- champ de vecteurs
local Nf = function(x,y) local A = f(x,y); return cpx.abs(Z(A[1],A[2])) end -- module de f(x,y)
g:Dshadedrectangle(-4,4,-5,5, pal.Picnic, {values=Nf, bar="right", baroptions={addvalues={0.5,1.5,2.5,3.5}}})
g:Dvectorfield(f,{view={-4,4,-5,5},draw_options="-stealth"})
g:Dgradbox({Z(-4,-5),Z(4,5),1,1}, {title="vector field $f(x,y)=(\\cos(x+y),x)$"})
g:Show()
\end{luadraw}
```

FIGURE 21 : Shaded rectangle



Dshadedregion

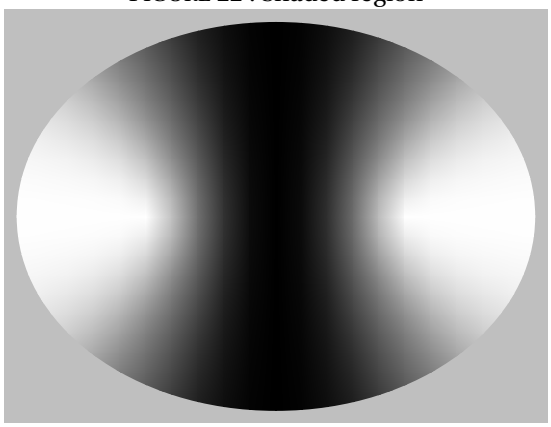
La méthode **g:Dshadedregion(apath, pal, options)** permet de remplir la région définie par le chemin ⟨*apath*⟩ avec un gradient de couleurs extraites de la palette ⟨*pal*⟩. Cette méthode utilise la précédente (**Dshadedrectangle()**) sur le rectangle définie par la boîte englobante du chemin. Chaque point (x,y) de ce rectangle a une couleur issue de la palette, calculée à partir d'une valeur $f(x,y)$ où f est une fonction à valeurs numériques définie sur ce rectangle. Le dessin est clippé par le chemin. L'argument ⟨*options*⟩ est identique à celui de la méthode précédente *Dshadedrectangle*.

```
\begin{luadraw}{name=Dshadedregion}
local ld = luadraw
```

```
local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{window={-4.5,6,-4.5,5}, size={10,10},bg="lightgray", bbox=false}
g:Labelsize("small")
require 'luadraw_shadedforms'
local pal = require 'luadraw_palettes'
local L1, L2 = -2, 2
local f = function(x,y)
    local z = Z(x,y)
    return (cpx.abs(z-L1)-cpx.abs(z-L2))^2
end
g:Dshadedregion({4,0,4,3,"e"}, pal.getPal(pal.Grays,{reverse=true}), {values=f, grid={20,20}})
g:Show()
\end{luadraw}
```

FIGURE 22 : Shaded region



8) Le module *luadraw_povray*

Ce module ne renvoie rien, il ajoute de nouvelles méthodes graphiques à la classe *ld.graph3d* ainsi qu'une fonction dans l'espace de noms *luadraw*.

Prérequis et introduction

Avant toute chose, le logiciel POV-Ray doit être installé sur votre ordinateur en version ligne de commande (version non graphique).

- Sous linux : POV-Ray est forcément en ligne de commande et s'installe avec le gestionnaire de paquets.
- Sous macOS : la version de POV-Ray en ligne de commande peut être installée via Homebrew.
- Sous windows : l'installateur est à télécharger sur le site de POV-Ray.

Ce module permet de créer des fichiers sources (relativement basiques) pour le logiciel POV-Ray (en ligne de commande) et de les compiler à la volée. Une fois créée, l'image (format *png*) peut être incluse automatiquement dans le graphique en cours, ce qui permet de dessiner dessous et dessus. L'image s'insérera parfaitement, c'est à dire que les repères 3D du graphique et de l'image seront parfaitement identiques, à condition d'être en mode de **projection orthographique**. Les fichiers sources et images sont créés dans le dossier de travail de *luadraw*. Les fichiers sources peuvent être évidemment compilés à part par l'utilisateur, les options à transmettre à POV-Ray sont écrites en commentaires au début du source. Le source est en trois parties : un préambule, la déclaration des objets, et le rendu des objets.

Réglages par défaut

L'utilisation de ce module nécessite certains réglages par défaut qui peuvent dépendre du système d'exploitation, ces valeurs par défaut peuvent être modifiées avec la fonction :

ld.povray_default(options)

L'argument ⟨*options*⟩ est une table dont les champs définissent les options, qui sont (avec leur valeur par défaut) :

- `bg=""`, cette option permet de définir le fond, soit c'est une chaîne vide (valeur par défaut) alors le fond sera transparent (indispensable si on doit dessiner sous l'image), soit c'est une chaîne non vide et alors ce doit un nom de

couleur connu de POV-Ray, soit c'est une table de la forme $\{r, g, b\}$ représentant une couleur avec les valeurs r, g et b entre 0 et 1.

- `shadow=true`, ce booléen indique si les objets doivent créer une ombre ou pas (ce booléen peut être modifié localement pour chaque objet créé).
- `imagescale=1`, cette option permet de modifier la taille de l'image *png* produite, pour des valeurs supérieures à 1 cela augmente évidemment son poids.
- `param="-V +A +FN"`, ce sont les paramètres de base transmis à POV-Ray, à ceux-ci s'ajouteront automatiquement la largeur et la hauteur de l'image, ainsi que l'option "+UA" si le fond doit être transparent.
- `pov_cmd=`, nom de la commande pour exécuter POV-Ray, sous unix la valeur par défaut est `"povray"`, sous windows la valeur par défaut est `"pvengine64.exe"`.
- `pov_cmd_ext=""`, chaîne de caractères qui est ajoutée à la fin de chaque commande envoyée à POV-Ray (vide par défaut).
- `win_param_ext="/RENDER /EXIT"`, paramètres supplémentaires qui ne concernent que windows.
- `include={}`, cette option permet de demander à POV-Ray d'inclure des fichiers **.inc*, par exemple :

  `include={"textures.inc", "colors.inc"}`

  ces deux fichiers sont normalement distribués avec POV-Ray, le premier définit des textures, le second des couleurs. Par défaut, aucune inclusion n'est faite.
- `arrowscale={1,1}`, table de deux nombres, le premier est un facteur d'échelle pour le rayon de la base des flèches (celles-ci sont des cônes), le second est un facteur d'échelle pour la hauteur des flèches.

Si vous modifiez une de ces options, la valeur que vous lui donnez devient la nouvelle valeur par défaut pour toute la suite du document, mais il n'est pas nécessaire de répéter cette fonction dans chaque graphique de votre document utilisant POV-Ray, il suffit de le faire une seule fois dans le préambule, par exemple, sous macOS si vous avec installé POV-Ray via *Homebrew*, alors ceci convient :

```
\documentclass{standalone}%
\usepackage[svgnames]{xcolor}
\usepackage[3d]{luadraw}
\directlua{%
require "luadraw_povray"
luadraw.povray_default( {pov_cmd = "/opt/homebrew/bin/povray"} )
}%
...
```

Bien sûr, si les valeurs par défaut conviennent à votre système (cela devrait être le cas sous linux), vous n'avez pas besoin d'utiliser la fonction **ld.povray_default()**.

Avant la création des objets

Il faut **obligatoirement initialiser** le dessin POV-Ray avec la méthode :

g:Pov_new(options).

L'argument ⟨*options*⟩ est exactement le même que pour la fonction précédente **ld.povray_default()**, sauf que l'effet sera uniquement local au graphique en cours.

Création des objets

Les objets sont créés à partir de méthodes sur le format **g:Pov_<command>(<data>, options)**. Chaque objet est d'abord déclaré dans le source POV-Ray en portant un nom (comme une variable), puis cet objet est ensuite "rendu" dans la troisième partie du source avec une texture définie à partir des options.

L'argument *options* est une table dont les champs définissent les options de l'objet, voici les **options communes à tous les objets** avec leur valeur par défaut :

- `name="object<num>"`, chaîne de caractères représentant le nom de l'objet créé, par défaut c'est le mot "object" suivi d'un numéro (ordre d'apparition). Donner un nom est utile lorsqu'on a besoin de réutiliser l'objet par la suite.
- `shadow=true`, booléen indiquant si l'objet crée une ombre.

- `render=true`, booléen indiquant si l'objet doit être affiché. On peut ne pas vouloir afficher un objet lorsque celui-ci intervient dans la construction d'un autre objet par exemple.
- Options définissant la texture de base, celle-ci est composée d'un *pigment* (couleur + opacité) et d'un *finish* (ambient + diffuse + phong) :
  - `color=ld.White`, cette option peut être soit une table au format $\{r, g, b\}$ (avec r, g et b entre 0 et 1), soit une chaîne de caractères, auquel cas celle-ci doit représenter une couleur connue de POV-Ray, ou bien une texture définie uniquement avec un *pigment* (par exemple `color="Blue_Sky3"`, ce nom est défini dans le fichier *textures.inc*).
  - `usepalette=nil`, cette option permet d'utiliser une palette de couleurs, la syntaxe est `usepalette={palette, func, minmax}` où $\langle palette \rangle$ est la liste (table) des couleurs, chacune d'elles étant une liste au format $\{r, g, b\}$ avec r, g et b entre 0 et 1, l'argument $\langle func \rangle$ est une chaîne qui définit le type de gradient, ce peut être soit `"x"` ou `"y"` ou `"z"` (dans ce cas, l'argument $\langle mimax \rangle$ doit être une liste contenant la valeur minimale et la valeur maximale), soit une chaîne de la forme `"function { <expression dépendant de x, y, z> }"`, l'expression qui dépend de x, y et z est une fonction (pour POV-Ray) qui doit renvoyer une valeur entre 0 et 1 (dans ce cas, l'argument $\langle minmax \rangle$ est inutile).
  - `opacity=1`, nombre entre 0 et 1 définissant l'opacité de l'objet.
  - `ambient=0.35`, `diffuse=0.8` et `phong=0.5` : trois paramètres définissant le *finish*.
  - `mytexture=nil`, ce paramètre permet définir sous forme d'une chaîne sa propre texture, ou bien de donner le nom d'une texture déjà connue de POV-Ray, dans ce cas les paramètres précédents ne sont pas pris en compte. Par exemple : `mytexture="texture{Silver_Metal}"` (cette texture est déclarée dans le fichier *textures.inc*).
- `matrix=nil`, matrice de transformation 3D qui s'appliquera localement à l'objet. La matrice 3D de transformation globale du graphique est également prise en compte.
- `clipbox=nil`, cette option permet de définir une liste (table) d'objets pour clipper celui qui est en cours de construction, ces objets peuvent être : soit un objet déjà créé, dans ce cas on donne son nom sous forme d'une chaîne, soit une boite, dans ce cas on donne une table de la forme *{M(xinf,yinf,zinf), M(xsup,ysup,zsup)}* (cela représente une diagonale de la boîte), soit une sphère, dans ce cas on donne une table de la forme *{center,radius}*, où *center* est un point 3D et *radius* un nombre positif.
- `clipplane=nil`, cette option permet de définir une liste (table) de plans pour clipper l'objet en cours de construction, chaque plan doit être de la forme $\{A, n\}$ où A est un point du plan (point 3D) et n un vecteur normal au plan. Seule la partie de l'objet se situant dans le demi-espace contenant n est conservée.

Liste des objets prédéfinis

Voici la liste des objets que l'on peut dessiner avec les méthodes correspondantes :

- **Surfaces implicites** d'équation $f(x, y, z) = 0$. C'est la méthode :

  g:Pov_implicit(povfunction, luafunction, options).

  L'argument $\langle povfunction \rangle$ est une chaîne contenant l'expression de $f(x, y, z)$, POV-Ray connaît les fonctions mathématiques, il faut cependant savoir que la fonction puissance est la fonction $pow : x \mapsto \text{pow}(x, n)$, et que POV-Ray ne gère pas les erreurs de calculs comme la division par zéro par exemple.

  Les options sont celles qui ont déjà été données, plus l'option spécifique : `containedby=<fenêtre 3D courante>`, celle-ci indique dans quelle boîte (ou sphère) seront faits les calculs, par défaut cette boîte est la fenêtre 3D du graphique en cours. Une boîte est une table de la forme *{M(xinf,yinf,zinf), M(xsup,ysup,zsup)}* (cela représente une diagonale de la boîte), une sphère est une table de la forme $\{C, r\}$, où C est le centre (point 3D) et r le rayon. Exemple :

```
local f = function(x,y,z) return x^2+y^2+z^2 - 1 end
local r = 1.1
g:Pov_implicit("x*x+y*y+z*z-1", f, {color=ld.SteelBlue, containedby={M(-r,-r,-r), M(r,r,r)}})
```

- **Surfaces paramétrées**. Deux syntaxes possibles :

  1. La méthode : **g:Pov_surface(f, u1, u2, v1, v2, options)**

     dessine la surface (lissée) paramétrée par la fonction $\langle f \rangle : (u, v) \mapsto f(u, v) \in \mathbf{R}^3$. L'intervalle pour le paramètre u est donné par $\langle u1 \rangle$ et $\langle u2 \rangle$. L'intervalle pour le paramètre v est donné par $\langle v1 \rangle$ et $\langle v2 \rangle$. Les options sont celles qui ont déjà été données, plus deux options spécifiques :

     - `grid={25,25}`, cette option définit le nombre de points à calculer pour le paramètre u suivi du nombre de points à calculer pour le paramètre v (25 par défaut).

– clip=false, avec la valeur true la surface est clippée par la fenêtre 3D courante.

Exemple :

```
local f = function(x,y) return M(x,y,x^2+y^2) end
g:Pov_surface(f,-2,2,-2,2, {color=ld.SteelBlue, clip=true})
```

2. La méthode : **g:Pov_surface(xfunc, yfunc, zfunc, u1, u2, v1, v2, options)**

dessine la surface paramétrée par $(u,v) \mapsto (x(u,v), y(u,v), z(u,v))$. Les arguments $\langle xfunc \rangle$, $\langle yfunc \rangle$ et $\langle zfunc \rangle$ sont trois chaînes contenant respectivement les expressions $x(u,v)$, de $y(u,v)$ et de $z(u,v)$. Les arguments $\langle u1 \rangle$ et $\langle u2 \rangle$, respectivement $\langle v1 \rangle$ et $\langle v2 \rangle$, définissent les bornes de l'intervalle pour le paramètre u, respectivement pour le paramètre v. Les options sont celles qui ont déjà été données, plus deux options spécifiques :

– containedby=<fenêtre 3D courante>, celle-ci indique dans quelle boîte (ou sphère) seront faits les calculs, par défaut cette boîte est la fenêtre 3D du graphique en cours. Une boîte est une table de la forme *{M(xinf,yinf,zinf), M(xsup,ysup,zsup)}* (cela représente une diagonale de la boîte), une sphère est une table de la forme {C, r}, où C est le centre (point 3D) et r le rayon.

– max_grad=nil, valeur numérique (optionnelle) permettant d'optimiser les calculs, voici ce que dit l'aide de POV-Ray sur ce nombre :

```
The max_gradient is the maximum magnitude of all six partial derivatives
over the specified ranges of u and v.
Take dx/du, dx/dv, dy/du, dy/dv, dz/du, and dz/dv and calculate them over
the entire range.
The max_gradient should be at least the maximum (absolute value) of all of those values.
Choosing a too small of a value will create holes or artifacts in the object.
```

Exemple :

```
g:Pov_surface("x", "y", "x*x+y*y",-2,2,-2,2, {color=ld.SteelBlue, max_grad=4})
```

Remarque : il s'agit de la même surface dans les deux exemples, mais la deuxième méthode est plus longue, et si on ne précise par l'option max_grad=4 alors on n'a pas la surface en entier.

- **Polyèdre ou liste de facettes**. C'est la méthode :

g:Pov_facet(F, options).

L'argument $\langle F \rangle$ est soit un polyèdre, soit une liste de facettes. Les options sont les options communes, plus les options spécifiques suivantes (avec les valeurs par défaut) :

– edge=false, booléen indiquant si les arêtes doivent être dessinées.

– edgestyle=<style de ligne courant>, style des arêtes.

– edgecolor=ld.Black, couleur des arêtes.

– edgewidth=<épaisseur courante>, épaisseur des arêtes (en dixième de point).

– hidden=ld.Hiddenlines, booléen indiquant si les arêtes cachées doivent être dessinées.

– hiddenstyle=ld.Hiddenlinestyle, style des arêtes cachées.

– hiddenscale=ld.Hiddenlinescale, nombre représentant un pourcentage, l'épaisseur des lignes cachées est égale à celle des lignes visibles multipliée par ce nombre. ld.Hiddenlinescale est une variable globale qui vaut 2/3 par défaut.

– csg=false, avec la valeur true, l'objet 3D défini par $\langle F \rangle$ peut-être utilisé dans des opérations de CSG (union, intersection, différence, merge), **sous réserve** que cet objet soit fermé et correctement orienté.

Exemple :

```
local T1 = tetra(M(-1,-1,-1), 3*vecI, 3*vecJ, M(1,1,3))
g:Pov_facet(T1, {color=ld.SteelBlue, edge=true, hidden=true})
```

Remarque : le dessin des arêtes cachées n'est pas toujours optimal, il peut être parfois plus intéressant de les dessiner avec TikZ par dessus l'image.

- **Ligne polygonale**. C'est la méthode :

g:Pov_polyline(L, options).

L'argument $\langle L \rangle$ est soit une liste de points 3D, soit une liste de listes de points 3D. Les options sont les options communes, plus les options spécifiques suivantes (avec les valeurs par défaut) :

– style=<style de ligne courant>, style de la ligne.

- `width=<épaisseur courante>`, épaisseur de la ligne.
- `close=false`, booléen indiquant si la ligne doit être refermée.
- `clip=false`, booléen indiquant si la ligne doit être clippée par la fenêtre 3D courante.
- `arrows=0`, trois valeurs possibles : soit 0 (pas de flèche), soit 1 (une flèche à la fin), soit 2 (une flèche au début et à la fin).
- `arrowscale={1,1}`, permet de jouer sur la taille des flèches (largeur pour le premier nombre, hauteur pour le second). Lorsque `arrowscale` est un nombre, la deuxième valeur est considérée égale à la première.
- `hiddenstyle=ld.Hiddenlinestyle`, style des arêtes cachées.
- `hiddenscale=ld.Hiddenlinescale`, nombre représentant un pourcentage, l'épaisseur des lignes cachées est égale à celle des lignes visibles multipliée par ce nombre. `ld.Hiddenlinescale` est une variable globale qui vaut `2/3` par défaut.

Exemple : dessin des axes

```
g:Pov_polyline({{-5*vecI,5*vecI},{-5*vecJ,5*vecJ},{-5*vecK,5*vecK}}, {arrows=1,width=8})
```

- **Points 3D**. C'est la méthode :

  g:Pov_dots(L, options).

  L'argument ⟨*L*⟩ est une liste de points 3D. Les options sont les options communes, plus les options spécifiques suivantes (avec les valeurs par défaut) :
  - `style="ball"`, deux styles possibles : `"ball"` (sphère) ou bien `"box"` (boite aux faces parallèles à la fenêtre 3D).
  - `dotscale=1`, permet de jouer sur la taille des points.
- **Un plan**. C'est la méthode :

  g:Pov_plane(P, options).

  L'argument ⟨*P*⟩ est une table de la forme {A, *n*} où A est un point du plan ⟨*P*⟩ (point 3D) et *n* un vecteur normal au plan. Le plan est automatiquement clippé par la fenêtre 3D. Les options sont les options de **g:Pov_facet()**, plus l'option spécifique `scale=1`.
- **Un cercle**. C'est la méthode :

  g:Pov_circle(A, R, N, options).

  Elle dessine le cercle de centre ⟨*A*⟩, de rayon ⟨*R*⟩, l'argument ⟨*N*⟩ désigne un vecteur normal au plan du cercle. Ce sont les options communes.
- **Les axes**. C'est la méthode :

  g:Pov_axes(O, options).

  Elle trace les axes en prenant le point ⟨*O*⟩ (point 3D) comme point d'intersection. Les options sont les mêmes que pour **g:Pov_polyline()**. Il n'y a ni graduations, ni légende.
- **Les solides de base**. Ce sont les méthodes :
  - **g:Pov_sphere(center, radius, options)**, qui dessine la sphère de centre ⟨*center*⟩ (point 3D) et de rayon ⟨*radius*⟩. Ce sont les options communes.
  - **g:Pov_torus(center, R, r, N, options)**, qui dessine le tore de centre ⟨*center*⟩ (point 3D) et de grand rayon ⟨*R*⟩, de petit rayon ⟨*r*⟩, dans le plan normal au vecteur ⟨*N*⟩. Ce sont les options communes.
  - **g:Pov_cylinder(A, R, B, options)**, qui dessine le cylindre d'axe (AB) allant de ⟨*A*⟩ jusqu'à ⟨*B*⟩ (points 3D), et de rayon ⟨*R*⟩. Ce sont les options communes plus l'option spécifique `hollow=false` qui indique si le cylindre est creux ou non.
  - **g:Pov_cone(A, R, B [r, options])**, qui dessine le cône d'axe (AB) allant de ⟨*A*⟩ jusqu'à ⟨*B*⟩ (points 3D), et de rayon ⟨*R*⟩ à l'extrémité ⟨*A*⟩, et de rayon ⟨*r*⟩ à l'extrémité ⟨*B*⟩. Le rayon ⟨*r*⟩ est optionnel et vaut 0 par défaut, dans ce cas, ⟨*B*⟩ est le sommet du cône. Ce sont les options communes plus l'option spécifique `hollow=false` qui indique si le cône est creux ou non.
  - **g:Pov_box(A, B, options)**, qui dessine une boîte dont les faces sont parallèles à celles de la fenêtre 3D du graphique, les points 3D ⟨*A*⟩ et ⟨*B*⟩ représente une diagonale de cette boîte, plus précisément ⟨*A*⟩=M(xinf, yinf, zinf) et ⟨*B*⟩=M(xsup, ysup, zsup). Ce sont les options communes.
- **CSG géométrie**. Ce sont les méthodes suivantes (pour chacune d'elles les options sont les options communes) :
  - **g:Pov_union(list, options)**, où ⟨*list*⟩ est une liste d'objets POV-Ray, ces objets peuvent être, soit le nom d'un objet déjà créé, soit une commande POV-Ray sous forme de chaîne. Le résultat sera considéré comme un seul objet.
  - **g:Pov_intersection(list, options)**, où ⟨*list*⟩ est une liste d'objets POV-Ray, ces objets peuvent être, soit le nom

d'un objet déjà créé, soit une commande POV-Ray sous forme de chaîne. Le résultat est la partie commune à ces objets.

- **g:Pov_merge(list, options)**, où ⟨*list*⟩ est une liste d'objets POV-Ray, ces objets peuvent être, soit le nom d'un objet déjà créé, soit une commande POV-Ray sous forme de chaîne. Cette commande fonctionne comme l'union, mais supprime les surfaces internes (contrairement à l'union), ceci est utile en cas de transparence.
- **g:Pov_difference(list, options)**, où ⟨*list*⟩ est une liste d'objets POV-Ray, ces objets peuvent être, soit le nom d'un objet déjà créé, soit une commande POV-Ray sous forme de chaîne. Le résultat la différence : ((objet1 moins objet2) moins objet3) etc.

- **Écrire directement dans le source**, avec les méthodes suivantes :
  - **g:Pov_comment(comment)**, qui écrit la chaîne ⟨*comment*⟩ dans le source sous forme de commentaire.
  - **g:Pov_special(code)**, qui écrit telle quelle la chaîne ⟨*code*⟩ dans le source, ce doit donc être du code POV-Ray. Il faut savoir que le point 3D qui se note *M(x,y,z)* dans *luadraw*, s'écrit dans le code POV-Ray : <*-x,y,z*>, le repère POV-Ray n'est donc pas notre repère usuel et il est indirect...

Sauvegarde, exécution, inclusion

- **Sauvegarde et exécution**. Lorsque tous les objets ont été créés, la sauvegarde et l'exécution du fichier par POV-Ray se font avec la méthode :

  g:Pov_exec([filename]).

  L'argument ⟨*filename*⟩ doit être un nom de fichier sans chemin ni extension, mais il est optionnel, par défaut c'est le nom du graphique en cours qui est utilisé. Le fichier créé portera l'extension *pov* et sera enregistré dans le dossier de travail de *luadraw* (celui-ci est contenu dans la variable `cachedir`). La commande utilisée pour l'exécution s'affiche dans le terminal, ainsi que les retours du logiciel POV-Ray, ce qui permet de voir s'il a rencontré une erreur. La construction de l'image apparaît à l'écran, mais la fenêtre se ferme dès la fin du rendu[3]. L'image porte le même nom que le source, sauf bien sûr l'extension qui est *png* au lieu de *pov*.

  NB : l'exécution de POV-Ray nécessite une compilation du document avec l'option *-shell-escape* ou *-enable-write18*. Une fois l'image obtenue, l'option n'est plus nécessaire s'il n'y a pas eu de modification de celle-ci.
- **Enregistrement seul**. Celui-ci se fait avec la méthode :

  g:Pov_save([filename]).

  Avec les mêmes remarques que précédemment pour l'argument ⟨*filename*⟩.
- **Inclusion de l'image**. Une fois celle-ci obtenue, l'image peut être incluse dans le graphique avec la méthode :

  g:Pov_show([filename]).

  Si l'argument ⟨*filename*⟩ n'est pas précisé alors il s'agit du nom du graphique en cours, sinon ⟨*filename*⟩ doit être un nom complet de fichier image (avec extension). L'inclusion se fait avec `\includegraphics[]{filename}`.

Exemples

Intersection de deux surfaces implicites

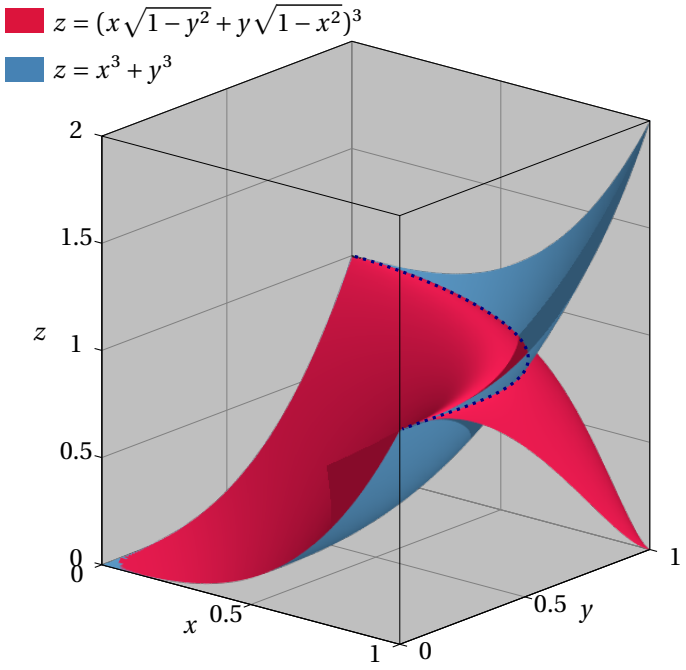
```
\begin{luadraw}{name=intersection_surf}
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Z, M = cpx.Z, pt3d.M

local g = ld.graph3d:new{window3d={0,1,0,1,0,2}, window={-0.25,1.5,-0.5,2.25}, size={10,10,0}, viewdir={-50,60}}
local sqrt = math.sqrt
require "luadraw_povray"
local f = function(x,y,z) return x^3+y^3-z end
local h = function(x,y,z) return (x*sqrt(1-y^2)+y*sqrt(1-x^2))^3-z end
local L = ld.implicit(function(x,y) return f(x,y,0)-h(x,y,0) end,0.01,1,0.01,1,{25,25})
L = ld.map(function(z) return M(z.re,z.im,z.re^3+z.im^3) end, L[1])  -- courbe intersection
-- utilisation de povray
g:Pov_new()
g:Pov_implicit("pow(x,3)+pow(y,3)-z", f, {color=ld.SteelBlue})
g:Pov_implicit("z-pow(x*sqrt(1-y*y)+y*sqrt(1-x*x),3)", h, {color=ld.Crimson, containedby={M(0,0,0),M(0.999,0.999,2)}})
```

3. Cela dépend de votre système d'exploitation.

```
g:Pov_exec()
-- dessin
g:Dboxaxes3d({grid=true, gridcolor="gray",fillcolor="lightgray",xyzstep=0.5, drawbox=true})
g:Pov_show() --Pov-Ray image
g:Dpolyline3d(L, "line width=1.2pt,dotted,Navy") --on dessine la courbe par dessus l'image
local d = 0.1
g:Dsquare(Z(-0.25,2.25), Z(-0.25,2.25-d),1,"draw=none,fill=Crimson")
g:Dsquare(Z(-0.25,2.25-2*d), Z(-0.25,2.25-3*d),1,"draw=none,fill=SteelBlue")
g:Dlabel("$z=(x\\sqrt{1-y^2}+y\\sqrt{1-x^2})^3$", Z(-0.25+d,2.25-d/2) , {pos="E"},
    "$z=x^3+y^3$", Z(-0.25+d,2.25-5*d/2),{})
g:Show()
\end{luadraw}
```

FIGURE 23 : Intersection de deux surfaces implicites



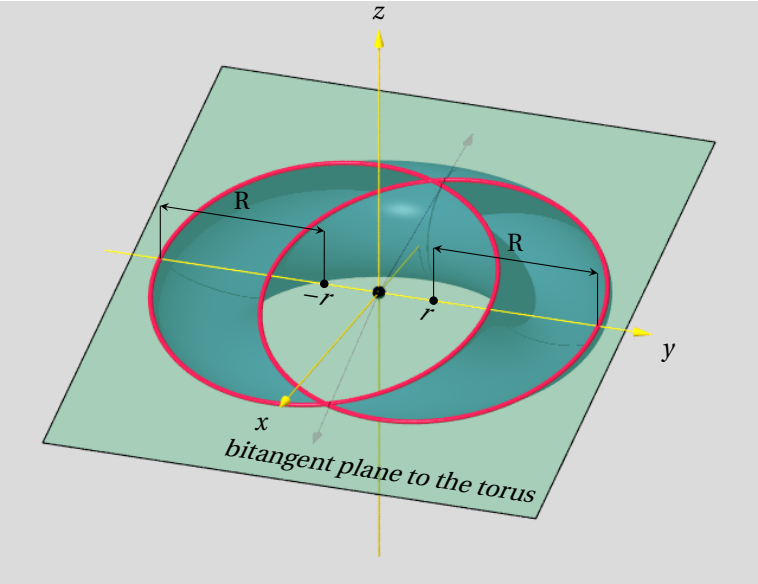
Cercles de Villarceau

```
\begin{luadraw}{name=Villarceau_circles}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window={-6.5,6.5,-5,5},margin={0,0,0,0}, size={10,10}, viewdir={20,65}}
require "luadraw_povray"
local R, r = 3, 1
local N = ld.rotate3d(vecK, math.asin(r/R)*ld.rad, {Origin, vecJ})
local P = {Origin, -N}
-- utilisation de povray
g:Pov_new({bg=ld.LightGray})
g:Pov_torus(Origin, R, r, vecK, {color=ld.SteelBlue, clipplane=P})
g:Pov_plane(P,{color=ld.SeaGreen, opacity=0.4, edge=true, scale=0.9})
g:Pov_axes(Origin,{color=ld.Gold,arrows=1})
g:Pov_dots(Origin, {dotscale=1.5})
g:Pov_circle( M(0,r,0),R,N,{color=ld.Crimson, width=12})
g:Pov_circle( M(0,-r,0),R,N,{color=ld.Crimson, width=12})
g:Pov_exec()
-- dessin
g:Pov_show()
local F = g:Plane2facet(P,0.9)
g:Dpolyline3d({{-r*vecJ,-r*vecJ+vecK},{-(R+r)*vecJ,-(R+r)*vecJ+vecK}})
g:Dpolyline3d({{r*vecJ,r*vecJ+vecK},{(R+r)*vecJ,(R+r)*vecJ+vecK}})
```

```
g:Dpolyline3d({{-r*vecJ+vecK,-(R+r)*vecJ+vecK}, {r*vecJ+vecK,(R+r)*vecJ+vecK}}, "stealth-stealth")
g:Ddots3d({-r*vecJ, r*vecJ})
g:Dlabel3d("$x$",5*vecI,{pos="SW"},"$y$", 5*vecJ,{pos="SE"}, "$z$",5*vecK,{pos="N"},
  "$R$", -(R/2+r)*vecJ+vecK, {}, "$R$", (R/2+r)*vecJ+vecK, {},
  "bitangent plane to the torus", F[1], {pos="NW", dir={vecJ,pt3d.prod(N,vecJ)}},
  "$-r$",-r*vecJ,{pos="S"}, "$r$", r*vecJ,{} )
g:Show()
\end{luadraw}
```

FIGURE 24 : Cercles de Villarceau



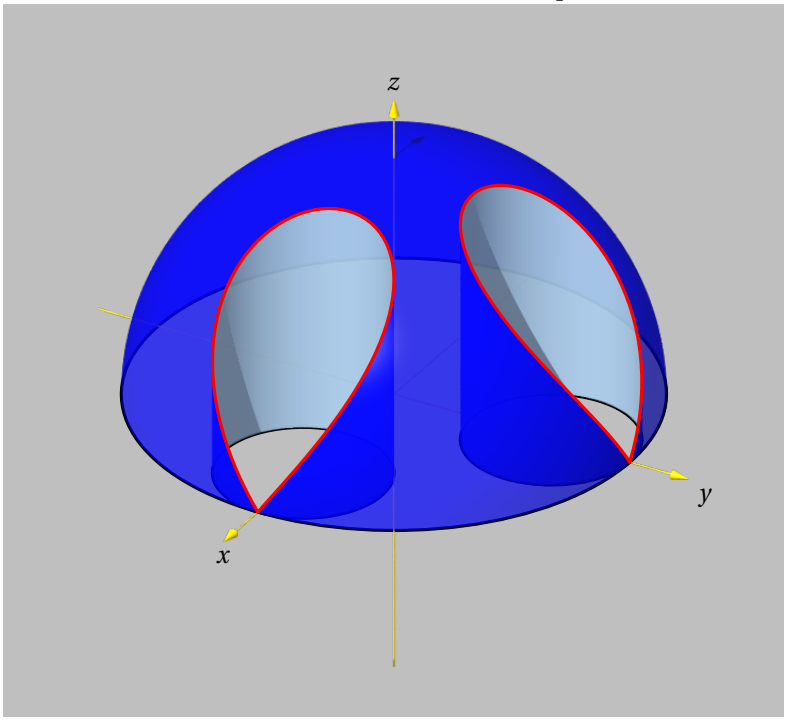
Trous dans une demi-sphère

```
\begin{luadraw}{name=holes_in_hemisphere}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window={-5,5,-4,5}, size={10,10}, bg="lightgray"}
require "luadraw_povray"
local O, R = Origin, 4
local A, r = 2*R/3*vecJ, R/3
local P = {Origin, vecK}
local base = ld.circle3d(A,r,vecK)[1] --base circulaire du cylindre (liste de points 3D)
local Cylborder = {}
for _, C in ipairs(base) do -- on projette chaque point C de la base sur la demi-sphère verticalement
    table.insert(Cylborder,C + math.sqrt(R^2-pt3d.abs2(C))*vecK)
end -- Cylborder est maintenant l'intersection entre la demi-sphère et le cylindre
-- utilisation de povray
g:Pov_new()
g:Pov_sphere(Origin, R, {name="sph", clipplane=P, render=false}) -- only declaration
g:Pov_cylinder(A-vecK, r, A+R*vecK, {name="cyl1", render=false })
g:Pov_union({"cyl1", "cyl1 rotate 90*z"}, {name="cylext", render=false})
g:Pov_cylinder(A-vecK, r-0.001, A+R*vecK, {name="cyl2", render=false })
-- le rendu
g:Pov_difference({"sph","cylext"}, {color=ld.Blue, opacity=0.7})
g:Pov_union( {"cyl2", "cyl2 rotate 90*z"}, {color=ld.LightBlue,opacity=0.8,clipbox="sph"})
g:Pov_axes(Origin,{color=ld.Gold,arrows=1})
g:Pov_exec()
-- dessin
g:Dcircle3d(O,R,vecK,"line width=1.2pt"); g:Dcircle3d(A,r,vecK,"line width=1.2pt")
g:Dcircle3d(ld.rotate3d(A,-90,{Origin,vecK}), r,vecK,"line width=1.2pt")
g:Pov_show()
g:Dpolyline3d( {Cylborder, ld.rotate3d(Cylborder,-90,{Origin,vecK})}, "red, line width=1.2pt")
g:Dlabel3d("$x$",5*vecI,{pos="S"}, "$y$", 5*vecJ, {pos="SE"}, "$z$", 5*vecK, {pos="N"})
g:Show()
```

```
\end{luadraw}
```

FIGURE 25 : Trous dans une demi-sphère



9) Le module *luadraw_log_axes*

Ce module ne renvoie rien, il ajoute de nouvelles méthodes graphiques à la classe *ld.graph* ainsi qu'une fonction dans l'espace de noms *luadraw*.

Le module *luadraw_log_axes* permet d'afficher une grille semi-logarithmique ou logarithmique, et de placer des points sur cette grille ou de dessiner des lignes polygonales. La création de cette grille entraîne une modification de la matrice 2D du graphe, pour cette raison le dessin se déroule en trois temps :

1. Initialisation de la grille avec la méthode **g:Beginlogview()**.
2. Le dessin avec les méthodes **g:Dlogpolyline()**, **g:Dlogdots()**, **Dlogline()** et **g:Dloglabel()**.
3. La fin de l'utilisation de la grille avec la méthode **g:Endlogview()**, ce qui restitue la matrice d'origine et permet d'ajouter de compléter éventuellement le dessin en coordonnées "normales" dans la fenêtre d'origine.

Initialisation : g:Beginlogview()

L'initialisation de la grille ainsi que son affichage sont gérés par la méthode :

g:Beginlogview(type, xmin, xmax, ymin, ymax, options).

- L'argument ⟨*type*⟩ permet de préciser le type de grille souhaitée, les valeurs possibles sont les chaînes : `"logx"`, `"logy"` ou bien `"logxy"`.
- Les quatre arguments ⟨*xmin*⟩, ⟨*xmax*⟩, ⟨*ymin*⟩ et ⟨*ymax*⟩ définissent les intervalles des valeurs sur les deux axes, celles-ci sont totalement indépendantes de la fenêtre 2D du graphe.
- L'argument⟨*options*⟩ est une table dont les champs définissent les options. Celles-ci sont (avec leur valeurs pas défaut) :
  - `viewport=<fenêtre 2D courante>`. Cette option est une table de la forme $\{x_1, x_2, y_1, y_2\}$, elle indique quelle zone de la fenêtre 2D du graphique doit être utilisée pour dessiner la grille. Par défaut, c'est la fenêtre courante en entier (celle définie par l'option `window` à la création du graphe).
  - `clip=true`. Booléen indiquant si la grille doit clipper les dessins. Le clipping est désactivé après l'exécution de la méthode **g:Endlogview()**.
  - `grid={true,true }`. Cette option permet d'afficher ou non les traits verticaux de la grille (correspondant à l'axe des x, ainsi que les traits horizontaux de la grille (correspondant à l'axe des y).

- `unit={"",""}`. Cette option permet de préciser de combien en combien vont les graduations sur les axes **non logarithmiques**. La valeur par défaut signifie qu'il faut prendre la valeur du pas (`xstep` sur Ox, ou `ystep` sur Oy), SAUF lorsque l'option `labeltext` n'est pas la chaîne vide, dans ce cas `unit` prend la valeur 1.
- `nbsubdiv={0 ou 3, 0 ou 3}`. Cette option permet de préciser le nombre de subdivisions entre deux graduations principales sur l'axe. Lorsque l'axe est non logarithmique la valeur par défaut est nulle, sinon la valeur par défaut est de 3.
- `xstep=nil ou 1` et `ystep=nil ou 1`. Cette option précise le pas sur les axes. Si l'axe est non logarithmique alors la valeur par défaut est de 1. Si l'axe est logarithmique alors cette option représente l'écart entre la première graduation (celle qui correspond à la valeur minimale) et la deuxième graduation principale ; s'il y a plusieurs décades, alors cette option vaut `nil` par défaut et chaque décade est découpée en 9 parties, s'il y a une seule décade cette option vaut par défaut le dixième de l'intervalle des valeurs.
- `defaultloglabels={2,3,5,10}`. Cette liste de valeurs est utilisée pour déterminer quelles sont les valeurs par décade sur les axes logarithmiques.
- `xdecadeloglabels=nil` et `ydecadeloglabels=nil`. Cette option s'applique aux axes logarithmiques. Elle précise la liste des valeurs de la **première décade**, ces valeurs donneront les graduations principales et les labels pour toutes les décades. Si la valeur minimale sur l'axe est *vmin*, alors la première décade est l'intervalle [vmin; 10 × vmin], par défaut cette liste de valeurs est donnée par le calcul : {2 × vmin, 3 × vmin, 5 × vmin, 10 × vmin} (il est inutile de préciser la première valeur *vmin*, celle-ci est automatiquement ajoutée). La liste {2, 3, 5, 10} est la valeur par défaut de l'option `defaultloglabels`.
- `xexponent=0` et `yexponent=0`. Cette option s'applique aux axes logarithmiques. Lorsque par exemple `xexponent=2` tous les labels de l'axe des x sont divisés par 10^2, et dans la légende sera ajoutée la chaîne : $(\times 10^2)$. C'est le même principe pour l'axe des y s'il est logarithmique.
- `xaddloglabels={}` et `yaddloglabels={}`. Cette option s'applique aux axes logarithmiques. Elle permet d'ajouter une liste de valeurs qui donneront des labels supplémentaires, ces valeurs doivent être comprises entre les valeurs minimale et maximale de l'axe.
- `tickpos={0.5,0.5}`. Cette option précise la position des graduations par rapport à chaque axe, ce sont deux nombres entre 0 et 1, la valeur par défaut de 0.5 signifie qu'ils sont centrés sur l'axe (0 et 1 représentent les extrémités).
- `xyticks={0.2,0.2}`. Cette option précise la longueur des graduations sur les axes.
- `xylabelsep={0,0}`. Cette option précise la distance entre les labels et les graduations sur les axes.
- `originloc=<coin inférieur gauche>`. Cette option précise le point origine des graduations pour l'axe non logarithmique.
- `originnum={minimum,minimum}`. Cette option précise la valeur au point origine des graduations (graduation numéro 0) pour l'axe non logarithmique.

  La formule qui définit le label à la graduation numéro n est :

  `(originnum + unit*n)"labeltext"/labelden.`
- `legend={"",""}`. Cette option permet de préciser une légende pour les axes. Par défaut la légende de l'axe des x est située sous l'axe et au milieu, et la légende de l'axe des y est située à gauche de l'axe, au milieu, et est écrite verticalement.
- `legendpos={0.5,0.5}`. Cette option précise la position (entre 0 et 1) de la légende par rapport à chaque axe.
- `legendsep={-0.5,-1}` . Cette option précise la distance entre la légende et l'axe. La légende est de l'autre côté de l'axe par rapport aux graduations.
- `legendangle={0,90}`. Cette option précise l'angle (en degrés) que doit faire la légende pour l'axe.
- `legendstyle={"S","N"}`. Précise le style de label pour les légendes, avec la valeur `"auto"` celui-ci est déterminé automatiquement, sinon on peut utiliser les valeurs : `"N"`, `"NW"`, `"W"`, `"SW"`, `"S"`, `"SE"`, `"E"`, `"NE"`.
- `labelpos={"bottom","left"}`. Cette option précise la position des labels par rapport à l'axe. Pour l'axe Ox, les valeurs possibles sont : `"none"`, `"bottom"` ou `"top"`, pour l'axe Oy c'est : `"none"`, `"right"` ou `"left"`.
- `labelstyle={"S","W"}`. Cette option définit le style des labels pour chaque axe. Les valeurs possibles sont : `"auto"`, `"N"`, `"NW"`, `"W"`, `"SW"`, `"S"`, `"SE"`, `"E"`, `"NE"`.
- `labelangle={0,0}`. Cette option définit pour chaque axe l'angle des labels en degrés par rapport à l'horizontale.
- `labelcolor={"",""}`. Cette option permet de choisir une couleur pour les labels sur chaque axe. La chaîne vide

représente la couleur par défaut.

- `labelden={1,1}`. Cette option précise le dénominateur des labels (entier) pour l'axe non logarithmique.
- `labeltext={"",""}`. Cette option définit le texte qui sera ajouté au numérateur des labels pour l'axe non logarithmique.
- `xynode_options=""`. Chaîne de caractères qui sera transmise telle quelle à l'instruction `\node{}` pour tous les labels sur les deux axes (mais pas pour les légendes).
- `xnode_options=xynode_options`. Chaîne de caractères qui sera transmise telle quelle à l'instruction `\node{}` pour tous les labels sur l'axe des x (mais pas pour la légende).
- `ynode_options=xynode_options`. Chaîne de caractères qui sera transmise telle quelle à l'instruction `\node{}` pour tous les labels sur l'axe des y (mais pas pour la légende).
- `use_siunitx={siunitx,siunitx}`. Cette option précise si les valeurs numériques doivent être formatées en utilisant le paquet *siunitx*, la valeur par défaut est celle de la variable globale `siunitx` qui vaut `false` par défaut.
- `mylabels=""`. Cette option permet d'imposer des labels personnels sur l'axe non logarithmique. Lorsqu'il y en a, la valeur passée à l'option doit être une liste du type : `{pos1,"text1",pos2,"text2",…}`. Le nombre ⟨*pos1*⟩ représente une abscisse dans le repère `(A,pas)` (A étant l'origine de l'axe), ce qui correspond au point d'affixe `A+pos1*pas`, le pas étant soit `xstep` soit `ystep` suivant l'axe.
- `gridstyle="solid"`. Cette option définit le style ligne pour la grille principale.
- `subgridstyle="solid"`. Cette option définit le style ligne pour la grille secondaire. Une grille secondaire apparaît lorsqu'il y a des subdivisions sur un des axes.
- `gridcolor="gray"`. Ceci définit la couleur de la grille principale.
- `subgridcolor="lightgray"`. Ceci définit la couleur de la grille secondaire.
- `gridwidth=4`. Épaisseur de trait de la grille principale (ce qui fait 0.4pt).
- `subgridwidth=2`. Épaisseur de trait de la grille secondaire (ce qui fait 0.2pt).

Méthodes de dessin et conversion

- La méthode **g:Dlogpolyline(L [, close, draw_options)]** permet de dessiner la ligne polygonale 2D ⟨L⟩ sur la grille, la conversion des coordonnées est automatique. L'argument ⟨*close*⟩ est un booléen indiquant si la ligne doit être refermée (`false` par défaut), et l'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera transmise à l'instruction *\draw*.
- La méthode **g:Dlogdots(L [, mark_options])** permet de dessiner sur la grille le nuage de points contenu dans ⟨L⟩ (nombre complexe, ou liste de nombres complexes, ou liste de listes de nombres complexes), la conversion des coordonnées est automatique. l'argument ⟨*mark_options*⟩ est une chaîne (vide par défaut) qui sera transmise à l'instruction *\draw*.
- La méthode **g:Dlogline(A, B [, draw_options])** permet de dessiner sur la grille la droite passant par les points ⟨A⟩ et ⟨B⟩ (la conversion des coordonnées est automatique). L'argument ⟨*draw_options*⟩ est une chaîne (vide par défaut) qui sera transmise à l'instruction *\draw*.
- La méthode **g:Dloglabel(text1, anchor1, options1, ...)** fait la même chose que la méthode **g:Dlabel()** sauf que les coordonnées sont d'abord converties pour s'adapter à la grille.
- Fonction de conversion : **ld.Zlog(z)** où ⟨z⟩ est un nombre complexe, renvoie l'affixe sur la grille du point correspondant.

Fin de l'utilisation de la grille : g:Endlogview()

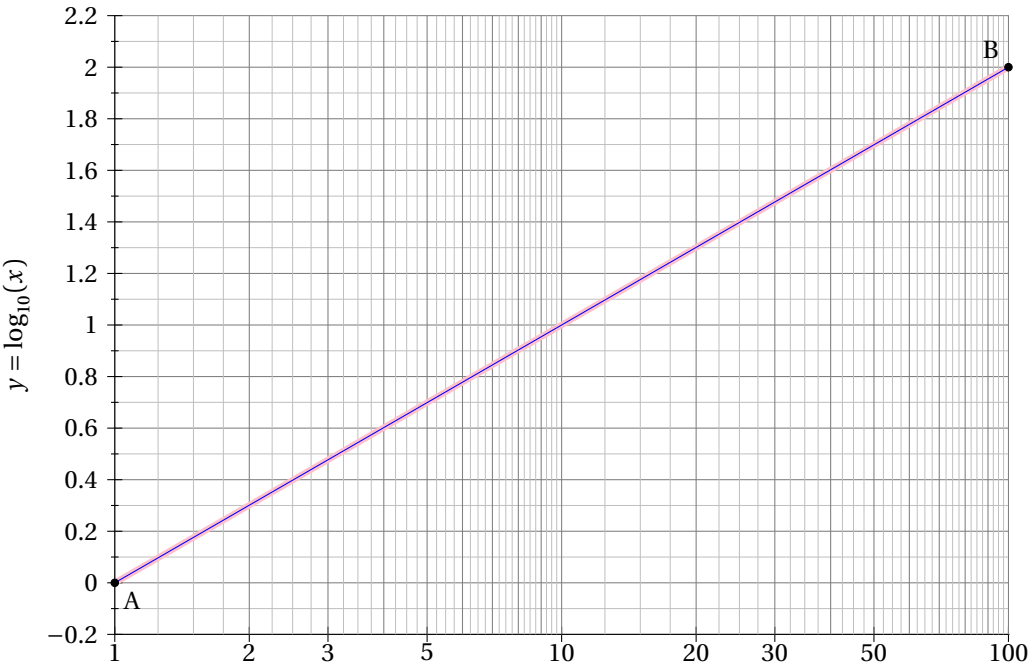
La méthode **g:Endlogview()** restaure la matrice initiale du graphe ainsi que la fenêtre 2D.

Exemples

```
\begin{luadraw}{name=logx_example}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z
local log10 = math.log10
```

```
local g = ld.graph:new{ size={14,10,0} }
require 'luadraw_log_axes'
g:Beginlogview("logx", 1, 100, -0.2, 2.2, {
   nbsubdiv={3,1},
   ystep = 0.2,
   legend={"La droite $(AB)$ et la courbe $y=\\log_{10}(x)$ sont confondues","$y=\\log_{10}(x)$"}
})
local L = ld.cartesian(function(x) return log10(x) end, 1,100)
local A, B = 1, Z(100,2)
g:Dlogline(A,B, "Pink,line width=2.4pt")
g:Dlogpolyline(L,"blue")
g:Dlogdots({A,B})
g:Dloglabel("$A$", A, {pos="SE"}, "$B$", B, {pos="NW"})
g:Endlogview()
g:Show()
\end{luadraw}
```

FIGURE 26 : Axe des x logarithmique



La droite (AB) et la courbe $y = \log_{10}(x)$ sont confondues

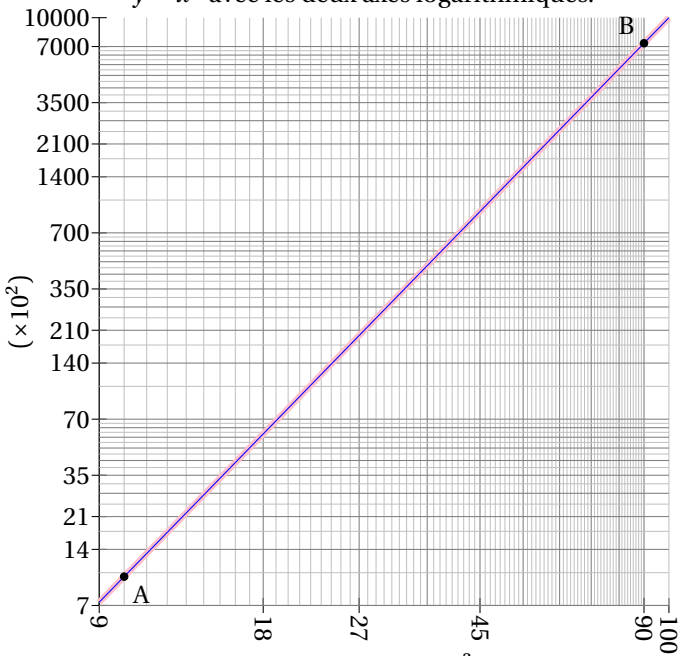
```
\begin{luadraw}{name=logy_and_logxy_example}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z
local  exp = math.exp

local g = ld.graph:new{ window={-10,10,-5,5}, size={18,10,0} }
require 'luadraw_log_axes'
g:Beginlogview("logy", -1, 5, 0.2, 200, { viewport={0.25,10,-5,4.5},
    nbsubdiv = {1,1},
    legendsep = {-0.5,-0.5},
    legend = {"La droite $(AB)$ et la courbe $y=\\exp(x)$ sont confondues",""}
})
local L = ld.cartesian(exp, -1, 5)
local A, B = Z(-1,exp(-1)), Z(5,exp(5))
g:Dlogline(A,B, "Pink,line width=2.4pt")
g:Dlogpolyline(L,"blue")
g:Dlogdots({A,B})
g:Dloglabel("$A$", A, {pos="SE"}, "$B$", B, {pos="NW"})
g:Endlogview()
```

```
g:Beginlogview("logxy", 9, 100, 7e2, 1e6, { viewport={-10,-0.25,-5,4.5},
    nbsubdiv = {8,1},
    xaddloglabels={100},
    legendsep = {-0.5,-0.75},
    yexponent=2,
    yaddloglabels={1e6},
    labelangle={-90,0}, labelstyle={"E","W"},
    legend = {"La droite $(AB)$ et la courbe $y=x^3$ sont confondues",""}
})
local L = ld.cartesian(function(x) return x^3 end, 9, 100)
local A, B = Z(10,1e3), Z(90,729e3)
g:Dlogline(A,B, "Pink,line width=2.4pt")
g:Dlogpolyline(L,"blue")
g:Dlogdots({A,B})
g:Dloglabel("$A$", A, {pos="SE"}, "$B$", B, {pos="NW"})
g:Endlogview()
-- drawing on the "normal" 2D window
g:Dlabel("$y=\\mathrm{e}^x$ avec l'axe $Oy$ logarithmique.",Z(5,4.5),{pos="N"},
  "$y=x^3$ avec les deux axes logarithmiques.",Z(-5,4.5),{})
g:Show()
\end{luadraw}
```

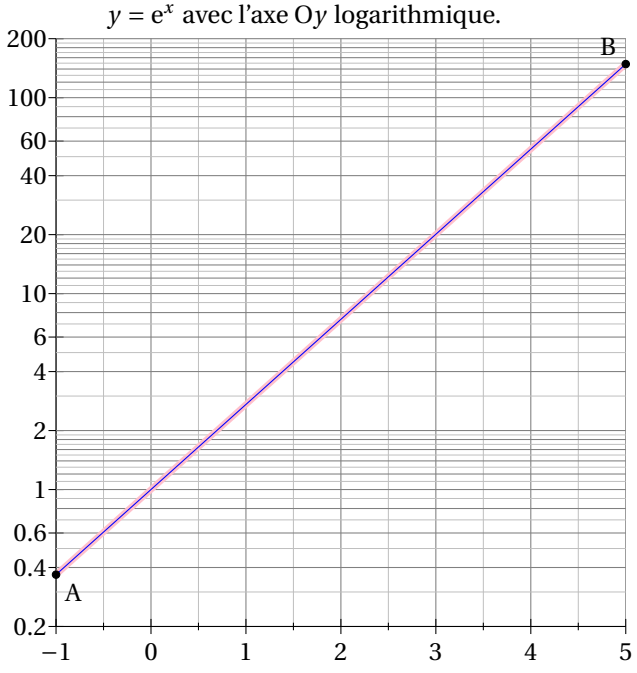
FIGURE 27 : Les deux autres cas de figure.

$y = x^3$ avec les deux axes logarithmiques.



La droite (AB) et la courbe $y = x^3$ sont confondues

$y = e^x$ avec l'axe Oy logarithmique.



La droite (AB) et la courbe $y = \exp(x)$ sont confondues

10) Le module *luadraw_decorations*

Le module **luadraw_decorations** redéfinit certaines méthodes graphiques pour leur ajouter des options comme par exemple : ajouter un label, mais pour chacune d'elles, **l'ancienne syntaxe est toujours valable**. Ce module ne renvoie rien.

Lignes polygonales 2D : g:Dpolyline()

La méthode **g:Dpolyline(L, options)** dessine la ligne polygonale $\langle L\rangle$ (liste de nombres complexes ou liste de listes de nombres complexes). L'argument $\langle options\rangle$ est une table dont les champs définissent les options possibles, celles-ci sont (avec leur valeur par défaut) :

- `close=false` : booléen indiquant si la ligne $\langle L\rangle$ doit être refermée.
- `clip=nil` : cette option vaut ou bien `nil` (valeur par défaut) ou bien une table de la forme `{x1,x2,y1,y2}`, dans le premier cas la ligne est clippée par la fenêtre 2D courante **après** sa transformation par la matrice 2D du graphe, dans le second cas la ligne est clippée par la fenêtre $[x_1;x_2]\times[y_1;y_2]$ **avant** d'être transformée par la matrice du graphe.

- `draw_options=""` : chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- Options pour l'ajout d'un label :
  - `label=""` : label à ajouter.
  - `anchor1d=nil` : nombre entre 0 et 1 indiquant la position du label le long de la ligne ⟨*L*⟩ (0 pour la début de ligne, 1 pour la fin de ligne).
  - `anchor=nil` : nombre complexe représentant le point d'ancrage du label dans le plan.
  - `anchor2d=cpx.Z(0.5,0.5)` : nombre complexe représentant la position du label dans le pavé $[0;1]\times[0;1]$ qui représente la boîte englobante de la ligne ⟨*L*⟩, donc par défaut le label est au centre de cette boîte. L'ordre de priorité est : `anchor`, `anchor1d`, `anchor2d` (si la première option vaut `nil`, la seconde est choisie, si elle vaut `nil` également, alors la troisième est choisie).
  - `pos="center"`, indique la position du label par rapport au point d'ancrage, il peut valoir `"center"` (centré), `"N"` (nord), `"NE"` (nord est), `"E"` (est), `"SE"` (sud est), `"S"` (sud), `"SW"` (sud ouest), `"W"` (ouest), `"NW"` (nord ouest). Par défaut, il vaut `center`, et dans ce cas le label est centré sur le point d'ancrage.
  - `dist=0`, distance en cm entre le label et son point d'ancrage lorsque `pos` n'est pas égal à `"center"`.
  - `dir={}`, cette option est une table de la forme `{dirX [,dirY,dep]}` qui indique la direction de l'écriture. Les 3 valeurs `dirX`, `dirY` et `dep` sont trois complexes représentant 3 vecteurs, les deux premiers indiquent le sens de l'écriture, le troisième un déplacement (translation) du label par rapport au point d'ancrage. Le vecteur `dep` est nul par défaut, et le vecteur `dirY`, s'il est absent, est égal au vecteur `dirX` tourné de 90 degrés dans le sens direct. Par défaut l'option `dir` est égale à la valeur courante de la direction de l'écriture.
  - `node_options=""` est une chaîne (vide par défaut) destinée à recevoir des options qui seront directement passées à TikZ dans l'instruction *node[]*.
  - `showanchor=false` : avec la valeur `true` le point d'ancrage est affiché ainsi que la boîte englobante du label.

Chemins 2D : g:Dpath()

La méthode **g:Dpath(P, options)** dessine chemin ⟨*P*⟩. Les ⟨*options*⟩ sont les mêmes que pour la méthode **g:Dpolyline()** (page 202) sauf les options `close` et `clip` qui ne sont pas prises en compte.

Droites 2D : g:Dline()

La méthode **g:Dline(d, options)** trace la droite ⟨*d*⟩, celle-ci est une liste du type {A, *u*} où A représente un point de la droite (un nombre complexe) et *u* un vecteur directeur (un nombre complexe non nul).

Variante : la méthode **g:Dline(A, B, options)** trace la droite passant par les points ⟨*A*⟩ et ⟨*B*⟩ (deux nombres complexes).

Dans les deux cas, les ⟨*options*⟩ sont les mêmes que pour la méthode **g:Dpolyline()** (page 202) sauf les options `close` et `clip` qui ne sont pas prises en compte, et plus l'option :

- `scale=1`. L'option `scale` (qui vaut 1 par défaut) est soit un nombre (pourcentage) permettant de faire varier la longueur du segment affiché (à partir de son milieu), soit une table de deux nombres (pourcentages) {scaleA, scaleB} permettant de faire varier la longueur du segment affiché à chacune de ses extrémités.

Segments 2D : g:Dseg()

La méthode **g:Dseg(seg, options)** où ⟨*seg*⟩={A, B} avec A et B deux nombres complexes, trace le segment [A; B]. Les ⟨*options*⟩ sont les mêmes que pour la méthode **g:Dpolyline()** (page 202) sauf les options `close` et `clip` qui ne sont pas prises en compte, et plus les deux options suivantes

- `scale=1`. L'option `scale` (qui vaut 1 par défaut) est soit un nombre (pourcentage) permettant de faire varier la longueur du segment affiché (à partir de son milieu), soit une table de deux nombres (pourcentages) {scaleA, scaleB} permettant de faire varier la longueur du segment affiché à chacune de ses extrémités.
- `ticks="none"` : permet d'ajouter on non des marques (graduations) sur le segment. La syntaxe pour en ajouter est : `ticks={nb [, length, space, draw_options]}` où ⟨*nb*⟩ est le nombre de traits à ajouter.

Arc 2D : g:Darc()

La méthode **g:Darc(B, A, C, r, direction, options)** dessine un arc de cercle de centre ⟨*A*⟩ (nombre complexe), de rayon ⟨*r*⟩, allant de ⟨*B*⟩ (nombre complexe) vers ⟨*C*⟩ (nombre complexe) dans le sens trigonométrique si l'argument ⟨*direction*⟩

vaut 1, le sens inverse sinon. L'argument ⟨*options*⟩ est une table dont les champs définissent les options possibles, celles-ci sont (avec leur valeur par défaut) :

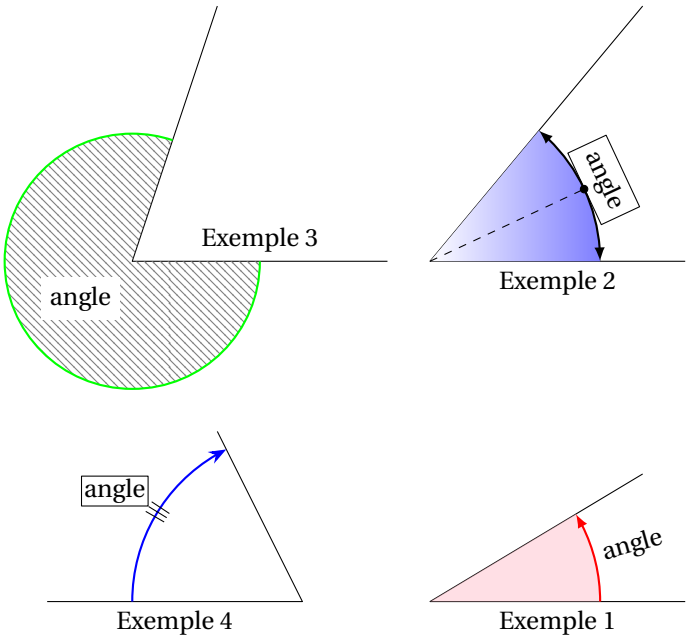
- `arc_options=""` : chaîne de caractères transmises à l'instruction *\draw* pour le dessin de l'arc de cercle.
- `sector_options=""` : chaîne de caractères définissant le mode de remplissage du secteur angulaire.
- `label=""` : texte qui sera affiché.
- `node_options=""` : chaîne définissant les options pour le label.
- `pos="auto"` : précise la position du label par rapport au point d'ancrage, les autres valeurs possibles sont les valeurs habituelles pour positionner un label : `"center"`, `"N"`, `"NW"`, etc. Par défaut, le point d'ancrage est situé à l'intersection de l'arc de cercle et de la bissectrice de l'angle.
- `dist=r` : distance entre le point d'ancrage et le centre du cercle, par défaut c'est le rayon ⟨*r*⟩ du cercle.
- `angle=0` : angle (en degrés) de la rotation que doit effectuer le point d'ancrage par défaut autour du centre du cercle.
- `rotate="none"` : indique si le label doit être tourné autour de son point d'ancrage, les autres valeurs possibles sont : `"auto"`, dans ce cas le label est écrit en suivant la direction de la bissectrice, ou bien `"ortho"`, et dans ce cas le label est écrit perpendiculairement à la bissectrice.
- `ticks="none"` : permet d'ajouter on non des marques (graduations) sur l'arc de cercle pour un angle aigu. La syntaxe pour en ajouter est : `ticks={nb [, length, space, draw_options]}` où ⟨*nb*⟩ est le nombre de traits à ajouter.
- `showanchor=false` : avec la valeur `true`, le point d'ancrage est dessiné, ainsi que la bissectrice et la boîte autour du label.

```
\begin{luadraw}{name=decoratedarc2D}
local ld = luadraw
local cpx = ld.cpx
local Z, i = cpx.Z, cpx.I

local g = ld.graph:new{window={-5,3,-3,5}, size={10,10}}
require 'luadraw_decorations'
g:Shift(Z(0,-2)) -- exemple 1
local b,a,c,r = 3, 0, 2.5+1.5*i, 2
g:Dpolyline({b,a,c})
g:Darc(b,a,c,r,1, {
    label="angle", rotate="auto",
    sector_options="fill=Pink,opacity=0.5",
    arc_options="red,line width=0.8pt,-latex"  })
g:Dlabel("Exemple 1",1.5,{pos="S"})
g:Shift(Z(0,4))  -- exemple 2
b,a,c,r = 3, 0, 2.5+3*i, 2
g:Dpolyline({b,a,c})
g:Darc(b,a,c,r,1, {
    label="angle", rotate="ortho",
    sector_options="left color=white, right color=blue!50",
    arc_options="line width=0.8pt,latex-latex",
    showanchor=true  })
g:Dlabel("Exemple 2",1.5,{pos="S"})
g:Shift(Z(-3.5,0))  -- exemple 3
b,a,c,r = 3, 0, 1+3*i, 1.5
g:Dpolyline({b,a,c})
g:Darc(b,a,c,r,-1, {
    label="angle", dist=r/2, pos="center",
    node_options="fill=white",
    sector_options="pattern=north west lines,pattern color=gray",
    arc_options="line width=0.8pt,green"   })
g:Dlabel("Exemple 3",1.5,{pos="N"})
g:Shift(Z(2,-4)) -- exemple 4
b,a,c,r = -3, 0, -1+2*i, 2
g:Dpolyline({b,a,c})
g:Darc(b,a,c,r,-1, {
    label="angle", dist=r+0.125,
    node_options="draw,inner sep=1pt",
    arc_options="line width=0.8pt,blue,-Stealth",
    ticks=3  })
g:Dlabel("Exemple 4",-1.5,{pos="S"})
```

```
g:Show()
\end{luadraw}
```

FIGURE 28 : Méthode *g:Darc()*



Arc 3D : g:Darc3d()

La méthode **g:Darc3d(B, A, C, r, direction, options)** dessine un arc de cercle de centre ⟨*A*⟩ (point 3D), de rayon ⟨*r*⟩, allant de ⟨*B*⟩ (point 3D) vers ⟨*C*⟩ (point 3D) dans le sens direct si l'argument ⟨*direction*⟩ vaut 1, le sens inverse sinon. Cet arc est tracé dans le plan contenant les trois points ⟨*A*⟩, ⟨*B*⟩ et ⟨*C*⟩, lorsque ces trois points sont alignés il faut préciser l'option `normal` (point 3D non nul) qui représente un vecteur normal au plan. Ce plan est orienté par le produit vectoriel $\vec{AB} \wedge \vec{AC}$ ou bien par le vecteur `normal` si celui-ci est précisé. L'argument⟨*options*⟩ est une table dont les champs définissent les options possibles, celles-ci sont (avec leur valeur par défaut) :

- `normal=nil` : point 3D non nul représentant un vecteur normal au plan (ABC), il est facultatif si les trois points sont non alignés.
- `arc_options=""` : chaîne de caractères transmises à l'instruction *\draw* pour le dessin de l'arc de cercle.
- `sector_options=""` : chaîne de caractères définissant le mode de remplissage du secteur angulaire.
- `label=""` : texte qui sera affiché.
- `node_options=""` : chaîne définissant les options pour le label.
- `pos="auto"` : précise la position du label par rapport au point d'ancrage, les autres valeurs possibles sont les valeurs habituelles pour positionner un label : `"center"`, `"N"`, `"NW"`, etc. Par défaut, le point d'ancrage est situé à l'intersection de l'arc de cercle et de la bissectrice de l'angle.
- `dist=r` : distance entre le point d'ancrage et le centre du cercle (A), par défaut c'est le rayon ⟨*r*⟩ du cercle.
- `angle=0` : angle (en degrés) de la rotation que doit effectuer le point d'ancrage par défaut autour du centre du cercle (A) et dans le plan du cercle.
- `rotate="none"` : indique si le label doit être tourné autour de son point d'ancrage **dans le plan de l'écran**. Les autres valeurs possibles sont : `"auto"`, dans ce cas le label est écrit en suivant la direction de la bissectrice, ou bien `"ortho"`, et dans ce cas le label est écrit perpendiculairement à la bissectrice.
- `rotate3d="none"` : indique si le label doit être tourné autour de son point d'ancrage **dans le plan** (ABC). Les autres valeurs possibles sont : `"auto"`, dans ce cas le label est écrit en suivant la direction de la bissectrice, ou bien `"ortho"`, et dans ce cas le label est écrit perpendiculairement à la bissectrice (toujours dans le plan (ABC)). Avec la valeur `"none"` le label est écrit dans le plan de l'écran.
- `ticks="none"` : permet d'ajouter on non des marques (graduations) sur l'arc de cercle pour un angle aigu. La syntaxe

pour en ajouter est : `ticks={nb [, length, space, draw_options]}` où ⟨*nb*⟩ est le nombre de traits à ajouter.

- `showanchor=false` : avec la valeur `true`, le point d'ancrage est dessiné, ainsi que la bissectrice et la boîte autour du label.
- `bezier=true` : pour dessiner l'arc avec des courbes de Bézier, avec la valeur `false` l'arc est une ligne polygonale. Lorsque TikZ ajoute une flèche au bout d'une courbe de Bézier celle-ci subit une légère déformation qui peut créer un artefact lorsque le secteur angulaire doit être peint, dans ce cas il vaut mieux prendre l'option `bezier=false`.

```
\begin{luadraw}{name=decorated_arcs3D}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{ window={-3,8,-5,8}, viewdir={"xy",0.8,60}, size={10,10} }
require 'luadraw_decorations'
local O, P = Origin, M(6.5, 6, 4)
local Px,Pz,Py,Pxz = ld.px(P), ld.pz(P), ld.py(P), ld.pxz(P)

g:Dpolyline3d({{O, 7*vecI}, {O, 7*vecJ}, {O, 6*vecK}}, "-Stealth,solid,black, line width=1pt")
g:Dlabel3d("$x$", 7*vecI, {pos="E"}, "$y$", 7*vecJ, {pos="N"}, "$z$", 6*vecK, {pos="S"})
g:Dpolyline3d({{P,Py}, {P,Pxz}, {Pxz,Px}, {O,Pxz}, {Pxz,Pz}}, "dashed,blue,thick")
g:Dseg3d({O, P}, "-latex,RosyBrown,line width=4pt")
g:Dpolyline3d({{O, 2.5*vecI}, {O, 2.5*vecJ}, {O, 2.5*vecK}}, "-latex,cyan,line width=3pt")

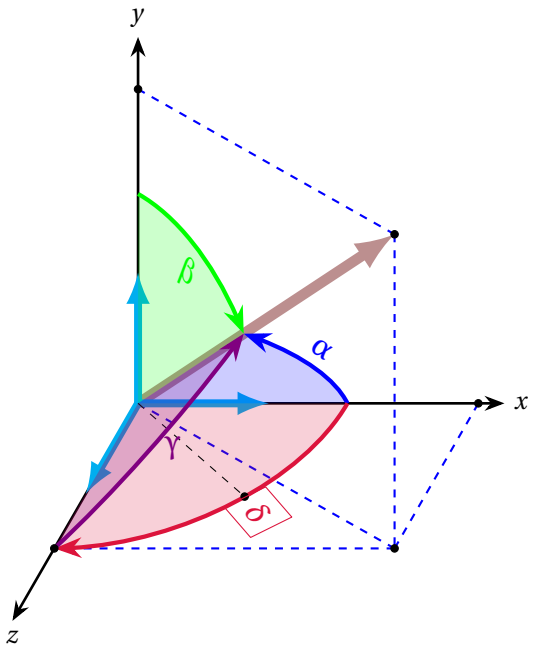
g:Darc3d(Px, O, P, 4, 1, {
    arc_options = "-Stealth,blue,ultra thick",
    sector_options = "fill=blue,opacity=0.2",
    label = "$\\alpha$", node_options="blue,scale=1.25",
    rotate = "ortho", pos="N" })

g:Darc3d(Py, O, P, 4, 1, {
    arc_options = "-Stealth,green,ultra thick",
    sector_options = "fill=green,opacity=0.2",
    label = "$\\beta$", node_options="green,scale=1.25",
    pos = "S", rotate3d = "ortho"  })

g:Darc3d(Px, O, Pz, 4, 1, {
    arc_options = "-Stealth,Crimson,ultra thick",
    sector_options = "fill=Crimson,opacity=0.2",
    label = "$\\delta$", node_options="Crimson,scale=1.25",
    rotate3d = "auto", angle=-5,
    showanchor=true  })

g:Darc3d(Pz, O, P, 4, 1, {
    arc_options = "-Stealth,violet,ultra thick",
    sector_options = "fill=violet,opacity=0.2",
    label = "$\\gamma$", node_options="violet,scale=1.25",
    pos="E"  })
g:Ddots3d({P,Pxz,Px,Py,Pz})
g:Show()
\end{luadraw}
```

FIGURE 29 : Méthode *g:Darc3d()*



Angle 3D : g:Dangle3d()

La méthode **g:Dangle3d(B, A, C, options)** dessine l'angle BAC avec un parallélogramme (deux côtés seulement sont dessinés). Le parallélogramme est dans le plan défini par les points ⟨*A*⟩, ⟨*B*⟩ et ⟨*C*⟩, ceux-ci ne doivent donc pas être alignés. L'argument⟨*options*⟩ est une table dont les champs définissent les options possibles, celles-ci sont (avec leur valeur par défaut) :

- `radius=0.25` : définit la longueur des côtés du parallélogramme.
- `arc_options=""` : chaîne de caractères transmises à l'instruction *\draw* pour le dessin du contour du parallélogramme (deux côtés seulement sont dessinés).
- `sector_options=""` : chaîne de caractères définissant le mode de remplissage du parallélogramme.
- Ajout d'un label : les options sont les mêmes que pour la méthode **g:Dpolyline()** (page 202) sauf l'option `close` qui n'est pas prise en compte.

Conclusion :

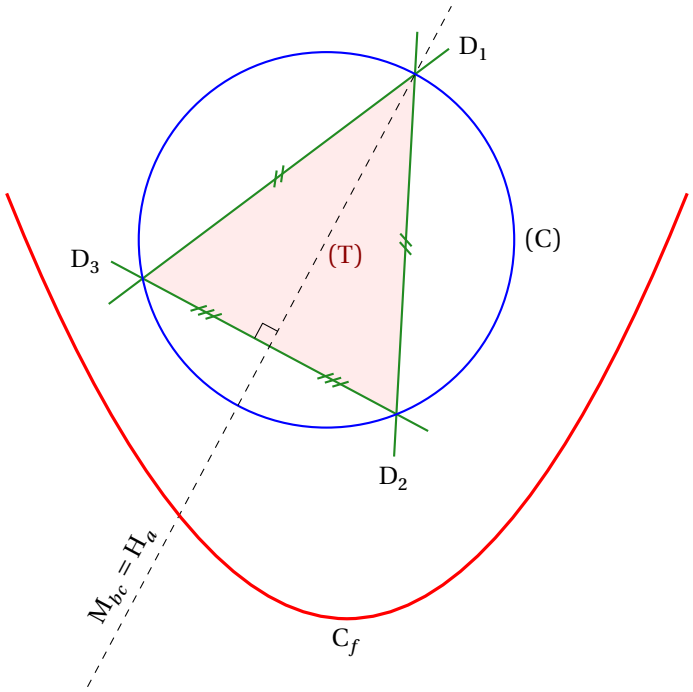
1. Dans toutes les méthodes graphiques 2D ou 3D qui font du dessin de lignes font appel à une des méthodes précédentes, pour celles qui ont un argument ⟨*draw_options*⟩, qui normalement est une chaîne de caractères, celui-ci peut être remplacé par une table ⟨*options*⟩ comme pour la nouvelle méthode **g:Dpolyline(L, options)**.
2. Dans toutes les méthodes graphiques 2D ou 3D qui font du dessin de lignes, qui ont déjà un argument ⟨*options*⟩ (ou ⟨*args*⟩), et qui parmi ces options en ont une une qui s'appelle `draw_options` ,qui normalement est une chaîne de caractères, celle-ci peut être remplacée par une table ⟨*options*⟩ comme pour la nouvelle méthode **g:Dpolyline(L, options)** (page 202).

```
\begin{luadraw}{name=decorations2d}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{size={10,10}}
require 'luadraw_decorations'
local A, B = Z(1,4), Z(-3,1)
local C = ld.rotate(B,50,A)
local A1 = (B+C)/2
g:Dpolyline({A,B,C}, {close=true, draw_options="draw=none,fill=pink,fill opacity=0.3",
```

```
    label="$(T)$", anchor=(A+B+C)/3, pos="E",node_options="DarkRed"})
g:Lineoptions(nil,"ForestGreen",8)
g:Dseg({A,B}, {scale=1.25,ticks=2,label="$D_1$",anchor1d=0,pos="E"})
g:Dseg({A,C}, {scale=1.25,ticks=2,label="$D_2$",anchor1d=1,pos="S"})
g:Dseg({C,B}, {scale=1.25,label="$D_3$",anchor1d=1,pos="W"})
g:Dmarkseg(B,A1,3); g:Dmarkseg(A1,C,3); g:Dangle(B,A1,A,0.25,"black,thin")
g:Dcircle({A,B,C}, {label='$(C)$', anchor1d=0, pos="E", draw_options="blue"})
g:Dmed(B,C,{label="$M_{bc}=H_a$",anchor1d=0.15,pos="N",dir=cpx.I*(C-B), draw_options="dashed,black,thin"})
g:Dcartesian( function(x) return (x/2)^2-4 end,{x={-5,5},
    draw_options={label="$C_f$", anchor2d=Z(0.5,0), pos="S", draw_options="red,line width=1.2pt"} })
g:Show()
\end{luadraw}
```

FIGURE 30 : Décorations 2D

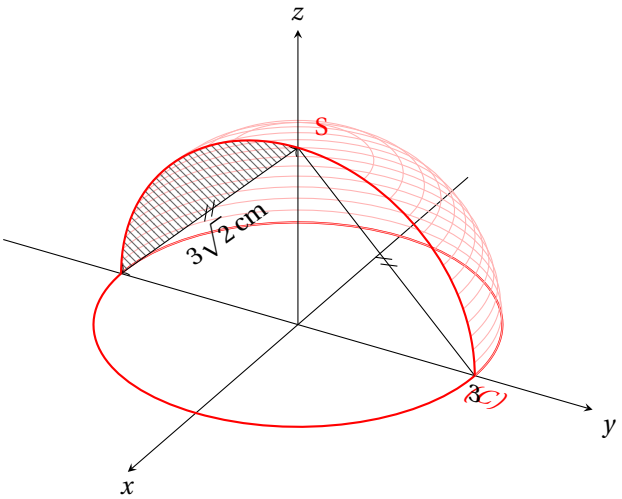



```
\begin{luadraw}{name=decorations3d}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M, Z = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M, ld.cpx.Z

local g = ld.graph3d:new{size={10,10}}
require 'luadraw_decorations'
g:Dline3d(-5*vecI,5*vecI, {label="$x$",anchor=5*vecI,pos="S",draw_options="-stealth"},true)
g:Dline3d(-5*vecJ,5*vecJ, {label="$y$",anchor1d=1,pos="SE",draw_options="-stealth"},true)
g:Dcircle3d(Origin, 3, vecK, {label="$(C)$",anchor1d=0,pos="SE",
    dir={vecJ,-vecI}, draw_options="red,thick", node_options="red"})
local nb = 15
local H = ld.linspace(0,3,nb)
for k = 1, nb-1 do
    local z = H[k]
    local r = math.sqrt(9-z^2)
    g:Darc3d(M(0,-r,z),z*vecK,M(0,r,z),r,-1,vecK,"thin, red!30")
end
g:Dpath3d({-3*vecJ,Origin,3*vecK,3,-1,vecI,"ca",3*vecK,"l","cl"},
    "draw=none,pattern color=black!60,pattern=north west lines")
g:Dseg3d({Origin,5*vecK},
    {label="$z$",anchor2d=Z(0,1),pos="N",draw_options="-stealth"})
g:Dseg3d({-3*vecJ,3*vecK}, {label="$3\\sqrt 2$\\,cm", pos="S", dir={M(0,1,1),M(0,-1,1)},
    ticks=2, draw_options="<->"})
g:Dseg3d({3*vecJ,3*vecK},
```

```
    {ticks=2, label="$3$", anchor=3*vecJ,pos="S",dir={vecJ,vecK}})
g:Dpath3d({-3*vecJ,Origin,3*vecJ,3,-1,vecI,"ca"},
    {label="$S$", anchor1d=0.5, pos="N",dist=0.15,draw_options="red,thick", node_options="red"})
g:Show()
\end{luadraw}
```

FIGURE 31 : Décorations 3D



11) Le module *luadraw_coils_chains*

Le module *luadraw_coils_chains* permet de dessiner des ressorts (deux types), ainsi que des chaînes (deux types). Il ajoute de nouvelles méthodes graphiques à la classe *ld.graph* mais ne renvoie rien.

Les ressorts

- La méthode **g:Dcoil(list, radius, options)** permet de dessiner un ressort. L'argument ⟨*radius*⟩ est le rayon du ressort. L'argument ⟨*list*⟩ a deux formes possibles :
  - Forme 1 : ⟨*list*⟩ = {a1, n1, a2, n2,..., aN}, où a_1, ..., a_N sont des points (nombres complexes), n_1 est un entier, c'est le nombre de spires entre a_1 et a_2, n_2 est le nombre de spires entre a_2 et a_3, etc.
  - Forme 2 : ⟨*list*⟩ = {C, n }, où C est une ligne polygonale (par exemple une courbe) et n un entier représentant le nombre total de spires. Le ressort sera dessiné le long de la courbe C.

  L'argument ⟨*options*⟩ est une table dont les champs sont les options qui vont modifier l'aspect esthétique du ressort. Ces options sont (avec leur valeur par défaut) :
  - `direction=1` : sens de rotation des spires (1 pour le sens direct, −1 pour le sens inverse).
  - `wire_dia=<2*radius/15>` : diamètre du fil.
  - `color="gray"` : couleur de remplissage des spires.
  - `colorB=color` : couleur de remplissage des spires "arrière", par défaut c'est la même valeur que l'option `color`.
  - `border=<couleur courante>` : couleur du contour des spires.
  - `border_width=<épaisseur courante>` : épaisseur du contour des spires.
  - `tension=1` : les spires sont légèrement courbées pour donner un effet 3D, cette option permet d'accentuer ou diminuer la courbure. Avec la valeur 0 il n'y a aucune courbure.
  - `start_angle=nil`, `end_angle=nil` : angles (en degrés) pour la première demi spire et pour la dernière. Ces angles sont déterminés de manière automatique, mais ils peuvent être modifiés avec ces options.
  - `wireframe=false` : avec la valeur `true`, les spires sont remplies d'une couleur unie, mais avec la valeur `false` (valeur par défaut) elles sont remplies avec un gradient, plus précisément avec la formule suivante :

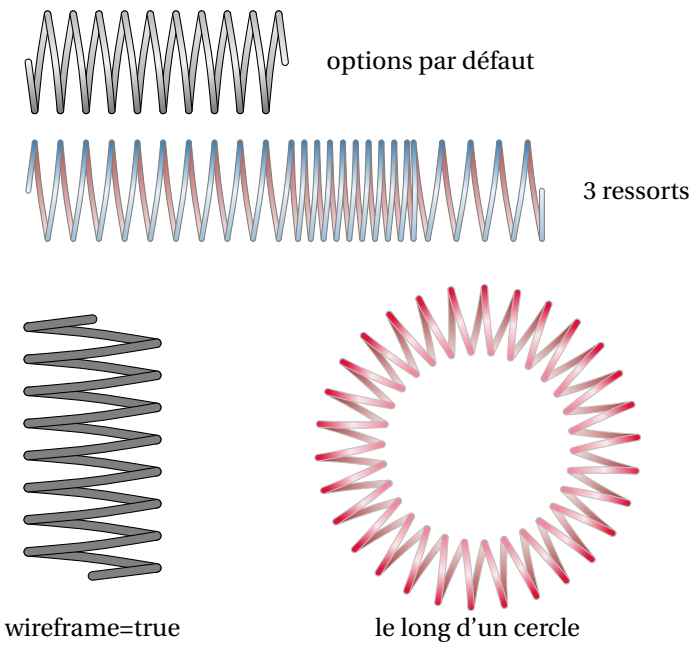
`left color=<color>!<leftC>, right color=<color>!<rightC>, middle color=<color>!<midC>`
avec les trois coefficients qui sont des options : `leftC=100`, `rightC=50`, `midC=10`.

- `holes=false` : avec la valeur `true` un point est dessiné aux extrémités du ressort.
- `reverse=false` : permet d'inverser l'ordre d'affichage du ressort lorsque celui-ci est constitué de plusieurs morceaux.

```
\begin{luadraw}{name=Dcoil_examples}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{window={-5.6,5,-5,5}, size={10,10}}
require 'luadraw_coils_chains'
local a, b, c, d = Z(-5,4), Z(-1,4), Z(1,4), Z(3,4)
g:Dcoil({a,10,b}, 0.75, {}); g:Dlabel("options par défaut", b+0.5, {pos="E"})
g:Shift(Z(0,-2))
g:Dcoil({a,10,b,10,c,5,d}, 0.75, {color="SteelBlue", direction=-1, wire_dia=0.075,
    border="gray", colorB="Brown"})
g:Dlabel("3 ressorts", d+0.5, {pos="E"})
g:Shift(Z(0,-2))
g:Dcoil({Z(-4,4),8, Z(-4,0)}, 1, {wireframe=true}); g:Dlabel("wireframe=true",Z(-4,-0.5),{pos="S"})
local C = ld.circle(Z(2,2),2)
g:Dcoil( {C,30}, 0.5, {color="Crimson", wire_dia=0.1, border="lightgray", colorB="DarkRed"})
g:Dlabel("le long d'un cercle",Z(2,-0.5),{pos="S"})
g:Show()
\end{luadraw}
```

FIGURE 32 : Méthode *g:Dcoil()*



- La méthode **g:Dcoil2(list, radius, options)** permet de dessiner un ressort en fil de fer. L'argument ⟨*radius*⟩ est le rayon du ressort. L'argument ⟨*list*⟩ a deux formes possibles :
  - Forme 1 : ⟨*list*⟩ = {a1, n1, a2, n2,..., aN}, où a_1, ..., a_N sont des points (nombres complexes), n_1 est un entier, c'est le nombre de spires entre a_1 et a_2, n_2 est le nombre de spires entre a_2 et a_3, etc.
  - Forme 2 : ⟨*list*⟩ = {C, n }, où C est une ligne polygonale (par exemple une courbe) et n un entier représentant le nombre total de spires. Le ressort sera dessiné le long de la courbe C.

  L'argument ⟨*options*⟩ est une table dont les champs sont les options qui vont modifier l'aspect esthétique du ressort. Ces options sont (avec leur valeur par défaut) :
  - `direction=1` : sens de rotation des spires (1 pour le sens direct, −1 pour le sens inverse).
  - `color=<couleur courante>` : couleur des spires.
  - `width=<épaisseur courante>` : épaisseur des spires (en dixième de point).

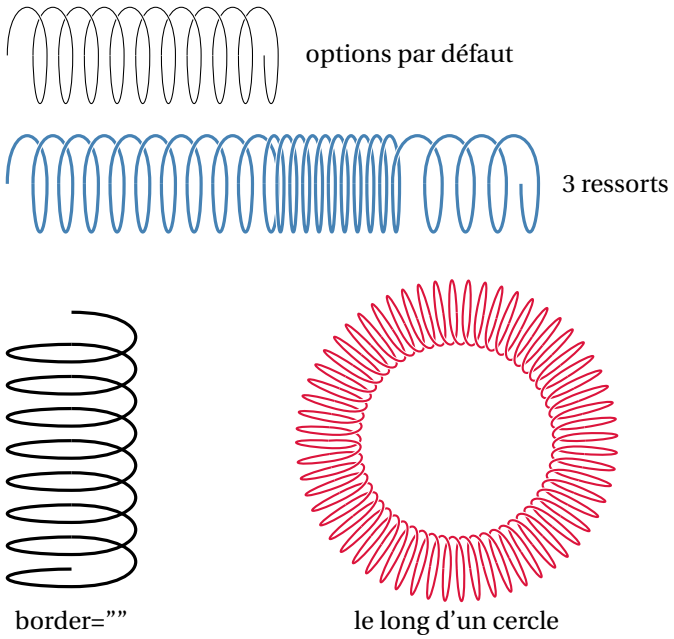
– `border="white"` : cette option est soit une chaîne vide (`""`), soit une chaîne représentant une couleur, dans le second cas, le dessin utilise l'option *double* de TikZ avec cette couleur en bordure permettant ainsi de voir les spires "avant" passer devant les spires "arrière", comme un ressort en 3D.

– `border_width=<épaisseur courante>` : épaisseur de la bordure (utilisée lorsque l'option `border` est pas égale à `nil`).

```
\begin{luadraw}{name=Dcoil2_examples}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{window={-5.6,5,-5,5}, size={10,10}}
require 'luadraw_coils_chains'
local a, b, c, d = Z(-5,4), Z(-1,4), Z(1,4), Z(3,4)
g:Dcoil2({a,10,b}, 0.75, {}); g:Dlabel("options par défaut", b+0.5, {pos="E"})
g:Shift(Z(0,-2))
g:Dcoil2({a,10,b,10,c,4,d}, 0.75, {color="SteelBlue", direction=-1, width=12})
g:Dlabel("3 ressorts", d+0.5, {pos="E"})
g:Shift(Z(0,-2))
g:Dcoil2({Z(-4,4),8, Z(-4,0)}, 1, {border="", width=12}); g:Dlabel('border=""',Z(-4,-0.5),{pos="S"})
local C = ld.circle(Z(2,2),2)
g:Dcoil2( {C,60}, 0.5, {color="Crimson", width=8, border="white"})
g:Dlabel("le long d'un cercle",Z(2,-0.5),{pos="S"})
g:Show()
\end{luadraw}
```

FIGURE 33 : Méthode *g:Dcoil2()*



Les chaînes

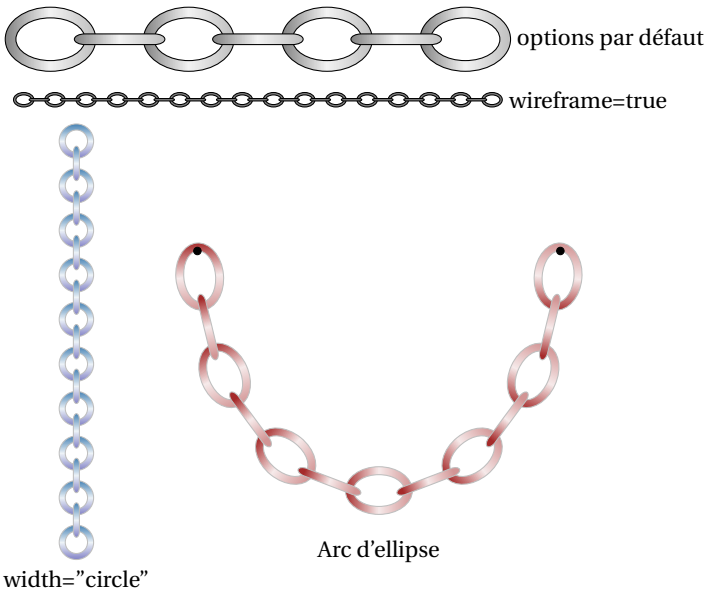
- La méthode **g:Dchain(list, link_length, options)** permet de dessiner une chaîne. L'argument ⟨*list*⟩ est une liste d'au moins deux nombres complexes, ce sont les points par où la chaîne passera. L'argument ⟨*link_length*⟩ est la longueur d'un maillon de la chaîne, mais celui-ci sera ajusté pour avoir un nombre entier de maillons. L'argument ⟨*options*⟩ est une table dont les champs sont les options qui vont modifier l'aspect esthétique de la chaîne. Ces options sont (avec leur valeur par défaut) :
  – `width=<(longueur calculée d'un maillon)/1.618>` : largeur d'un maillon, avec la valeur `width="circle"`, le maillon sera circulaire.
  – `wire_dia=<width/4>` : diamètre du fil.
  – `color="gray"` : couleur de remplissage des maillons.

– `colorB=color` : deuxième couleur de remplissage.
– `border=<couleur courante>` : couleur du contour des maillons.
– `border_width=<épaisseur courante>` : épaisseur du contour des maille.
– `wireframe=false` : avec la valeur `true`, les maillons sont remplies d'une couleur unie, mais avec la valeur `false` (valeur par défaut) elles sont remplies avec un gradient, plus précisément avec la formule suivante :
`left color=<color>!<leftC>, right color=<colorB>!<rightC>, middle color=<color>!<midC>`
avec les trois coefficients qui sont des options : `leftC=100`, `rightC=50`, `midC=10`.

```
\begin{luadraw}{name=Dchain_examples}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{window={-5.25,6,-5,5.25}, size={10,10}}
require 'luadraw_coils_chains'
g:Labelsize("small")
local a, b = Z(-5,4.5), Z(3,4.5)
g:Dchain({a,b}, 1, {}); g:Dlabel("options par défaut", b, {pos="E", dist=0.1})
g:Shift(Z(0,-1))
g:Dchain({a,b}, 0.25, {wireframe=true}); g:Dlabel("wireframe=true", b, {pos="E"})
g:Shift(Z(0,1))
a, b = Z(-4,3), Z(-4,-4)
g:Dchain({a,b}, 0.35, {color="SteelBlue", colorB="DarkBlue", width="circle", wire_dia=0.1, border="lightgray"})
g:Dlabel('width="circle"', b, {pos="S",dist=0.1})
a, b = Z(-2,1), Z(4,1)
local C = ld.ellipticarc(a,Z(1,1),b,3,4,1)
g:Dchain(C, 0.75, {color="Brown", border="lightgray"}); g:Ddots({a,b})
g:Dlabel("Arc d'ellipse",Z(1,-3),{pos="S",dist=0.5})
g:Show()
\end{luadraw}
```

FIGURE 34 : Méthode *g:Dchain()*



- La méthode **g:Dchain2(list, R, options)** permet de dessiner une chaîne sous forme de deux fils ondulés entrecroisés. L'argument ⟨*list*⟩ est une liste d'au moins deux nombres complexes, ce sont les points par où la chaîne passera. L'argument ⟨*R*⟩ est le rayon de la chaîne (hauteur d'une ondulation). L'argument ⟨*options*⟩ est une table dont les champs sont les options qui vont modifier l'aspect esthétique de la chaîne.
  – `direction=1` : sens de rotation des fils (1 pour le sens direct, −1 pour le sens inverse).
  – `color=<couleur courante>` : couleur de la chaîne.
  – `width=<épaisseur courante>` : épaisseur des fils (en dixième de point).
  – `border="white"` : cette option est soit une chaîne vide (`""`), soit une chaîne représentant une couleur, dans le

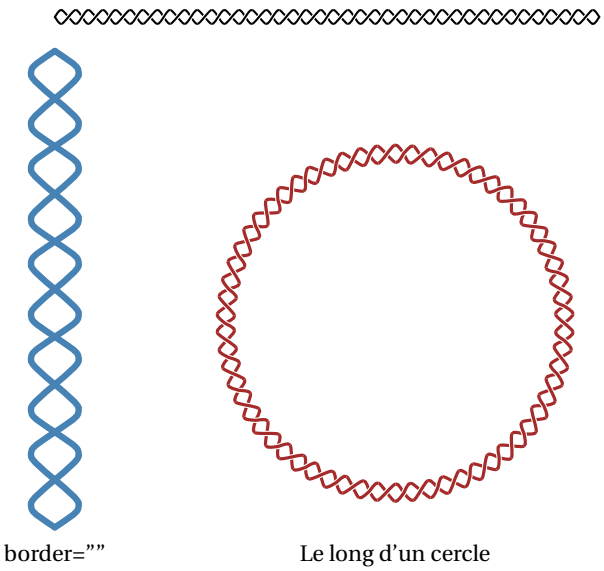
second cas, le dessin utilise l'option *double* de TikZ avec cette couleur en bordure permettant ainsi de distinguer le fil qui passe devant de celui qui passe derrière.

– `border_width=<épaisseur courante>` : épaisseur de la bordure (utilisée lorsque l'option `border` est pas égale à `nil`).

```
\begin{luadraw}{name=Dchain2_examples}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{window={-5,5,-4.5,5}, size={10,10}}
require 'luadraw_coils_chains'
g:Labelsize("small")
local a, b = Z(-4,4.5), Z(4,4.5)
g:Dchain2({a,b}, 0.1, {width=8})
g:Shift(Z(0,1))
a, b = Z(-4,3), Z(-4,-4)
g:Dchain2({a,b}, 0.35, {color="SteelBlue", width=24, border=""})
g:Dlabel('border=""', b, {pos="S",dist=0.1})
a, b = Z(-2,1), Z(4,1)
local C = ld.circle(Z(1,-1), 2.5)
g:Dchain2(C, 0.125, {color="Brown", width=12})
g:Dlabel("Le long d'un cercle",Z(1,-4),{pos="S",dist=0.1})
g:Show()
\end{luadraw}
```

FIGURE 35 : Méthode *g:Dchain2()*



12) Le module *luadraw_pdfliteral*

Ce module ne renvoie rien, il ajoute de nouvelles méthodes graphiques aux classes *ld.graph* et *ld.graph3d*.

Lorsque le nombre de données que TikZ doit dessiner devient très important, par exemple un grand nombre de facettes d'une surface, un grand nombre de segments ou chemins d'un pavage, ou encore un nuage de points avec un grand nombre de points, les macros utilisées par TikZ deviennent tellement nombreuses que les temps de compilation et d'affichage peuvent être rédhibitoires.

Une façon de contourner cette difficulté en réduisant de manière significative les temps de compilation est de faire ce type de dessins directement en pdf sans passer par les macros TikZ. Cela ne veut pas dire qu'on n'utilise plus TikZ, car celui-ci est très performant dans la gestion des styles par exemple. L'idée est de rester dans un environnement *tikzpicture*, mais lorsqu'il y a un très grand nombre de segments à tracer par exemple, sans changement de style autre que la couleur (en

particulier sans changement d'opacité), avec des remplissages de type *solide* uniquement (ou alors pas de remplissage du tout), on écrit les instructions de dessins, non plus en utilisant la commande *\draw*, mais directement en langage pdf natif.

La méthode g:Dliteralpath()

La méthode **g:Dliteralpath(path1, args1, path2, args2, ...)** permet de dessiner en pdf les chemins 2D ⟨*path1*⟩, ⟨*path2*⟩,.... Chaque chemin doit être suivi d'une table : ⟨*args1*⟩, ⟨*args2*⟩, ... Ces tables représentent les options de chacun des chemins, celles-ci sont :

- `fill=<table {r,g,b}>` : couleur de remplissage, par défaut c'est la couleur courante. La valeur `fill="none"` supprime le remplissage, dans les autres cas la couleur doit être une table au format $\{r,g,b\}$ (avec r, g et b entre 0 et 1) et le remplissage est **forcément solide** (avec l'opacité courante), les autres styles de remplissages sont sans effet.
- `draw=<table {r,g,b}>` : couleur de tracé, par défaut c'est la couleur courante. La valeur `draw="none"` supprime le tracé (mais pas le remplissage), dans les autres cas la couleur doit être une table au format $\{r,g,b\}$ (avec r, g et b entre 0 et 1) et le tracé se fait dans le style courant et l'opacité courante.
- `width=<épaisseur en dixième de points>` : épaisseur du trait, c'est l'épaisseur courante par défaut.

NB : les options choisies pour un chemin s'appliquent également aux chemins suivants si elles ne sont pas modifiées.

La méthode g:Dliteralpolyline()

La méthode **g:Dliteralpolyline(polyline1, args1, polyline2, args2, ...)** permet de dessiner en pdf les lignes polygonales 2D ⟨*polyline1*⟩, ⟨*polyline2*⟩,.... Chaque ligne polygonale doit être suivie d'une table : ⟨*args1*⟩, ⟨*args2*⟩, ... Ces tables représentent les options de chacune des lignes polygonales, celles-ci sont les mêmes que pour la méthode **g:Dliteralpath()**, plus l'option :

- `close=<booléen>` : indiquant si le ligne polygonale doit être refermée (`false` par défaut).

NB : les options choisies pour une ligne polygonale s'appliquent également aux suivantes si elles ne sont pas modifiées.

La méthode g:Dliteraldots()

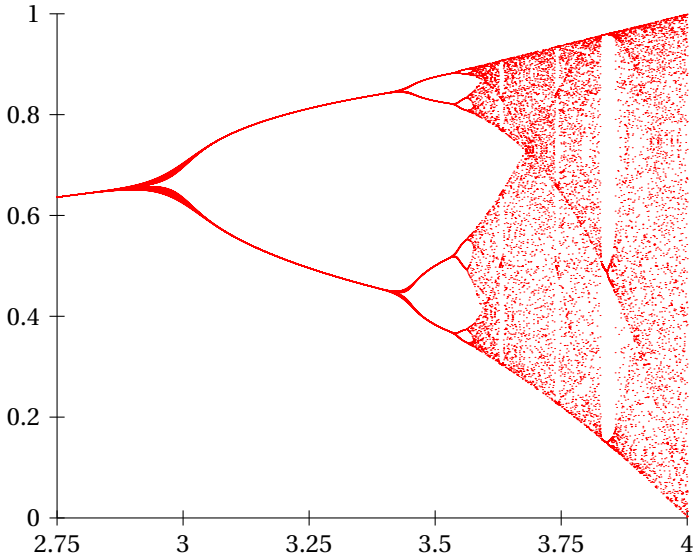
La méthode **g:Dliteraldots(dots1, args1, dots2, args2, ...)** permet de dessiner en pdf les listes de points 2D (nombres complexes) ⟨*dots1*⟩, ⟨*dots2*⟩,.... Chaque liste de points doit être suivie d'une table : ⟨*args1*⟩, ⟨*args2*⟩, ... Ces tables représentent les options, celles-ci sont :

- `color=<table {r,g,b}>` : couleur des points, par défaut c'est la couleur courante du tracé. La couleur doit être une table au format $\{r,g,b\}$ (avec r, g et b entre 0 et 1) et le tracé se fait en trait plein et avec l'opacité courante. Les points sont circulaires.
- `width=<épaisseur en dixième de points>` : épaisseur du trait, c'est l'épaisseur courante par défaut. Cette épaisseur détermine le diamètre des points.

NB : les options choisies pour une ligne polygonale s'appliquent également aux suivantes si elles ne sont pas modifiées.

```
\begin{luadraw}{name=bifurcation}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z
local g = ld.graph:new{ window={2.65,4,-0.15,1}, size={10,10} }
require 'luadraw_pdfliteral'
local dots = {}
for r = 2.75, 4, 0.001 do
    local u = 0.5
    for k = 1, 25 do u = r*u*(1-u) end -- transition
    for k = 1, 25 do u = r*u*(1-u); table.insert(dots, Z(r,u)) end
end
g:Daxes({Z(2.75,0),0.25,0.2}, {limits={{2.75,4},{0,1.001}}, originpos={"center","center"} })
g:Dliteraldots(dots, {width=1, color=ld.Red}) -- 31275 points
g:Show()
\end{luadraw}
```

FIGURE 36 : Diagramme de bifurcation de la suite $u_{u+1} = r\,u_n(1-u_n)$



La méthode g:Dliteralfacet()

La méthode **g:Dliteralfacet(F1, args1, F2, arg2, ...)** fait la même chose que la méthode **g:Dmixfacet()**, sauf que le dessin des facettes est fait en pdf natif. Les arguments ⟨*F1*⟩, ⟨*F2*⟩, ..., sont des listes de facettes (listes de listes de points 3D), et ⟨*args1*⟩, ⟨*arg2*⟩, ..., sont des tables représentant les options, celles-ci sont :

- `mode=ld.mShaded` : définit le mode de représentation. Les valeurs possibles sont :
  - `ld.mWireframe` : mode fil de fer, on dessine les arêtes seulement. Lorsque l'option `usepalette` est différente de `nil`, la couleur de chaque arête et calculée dans la palette (suivant le même mode que les facettes lorsque celles-ci sont peintes).
  - `ld.mFlat` ou `ld.mFlatHidden` : on dessine les faces de couleur unie, ainsi que les arêtes.
  - `ld.mShaded` ou `ld.mShadedHidden` : on dessine les faces de couleur nuancée en fonction de leur inclinaison, ainsi que les arêtes.
  - `ld.mShadedOnly` : on dessine les faces de couleur nuancée en fonction de leur inclinaison, mais pas les arêtes.
- `contrast=1` : ce nombre permet d'accentuer ou diminuer la nuance des couleurs des facettes dans les modes `ld.mShaded`, `ld.mShadedHidden`, `ld.mShadedOnly`.
- `edgecolor=<couleur courante>` : chaîne qui définit la couleur des arêtes.
- `edgewidth=<épaisseur courante>` : épaisseur de trait des arêtes en dixième de point.
- `backcull=false` : avec la valeur `true`, les facettes considérées comme non visibles (vecteur normal non dirigé vers l'observateur) ne sont pas affichées. Cette option est intéressante pour les objets convexes car elle permet de diminuer le nombre de facettes à dessiner.
- `clip=false` : avec la valeur `true`, les facettes sont clippées par la fenêtre 3D.
- `twoside=true` : booléen qui indique si on distingue les deux côtés des facettes (intérieur et extérieur), avec la valeur `true` les deux côtés n'auront pas exactement la même couleur.
- `reverse=false` : avec la valeur `true` l'orientation des facettes est inversée.
- `color=<table {r,g,b}>` : couleur de remplissage des facettes, par défaut c'est la couleur `ld.White`. La couleur doit être une table au format $\{r, g, b\}$ (avec r, g et b entre 0 et 1).
- `usepalette=nil` : cette option permet éventuellement de préciser une palette de couleurs pour peindre les facettes ainsi qu'un mode de calcul, la syntaxe est : `usepalette={palette,mode[,minmax]}`, où ⟨*palette*⟩ désigne une table de couleurs qui sont elles-mêmes des tables de la forme $\{r, g, b\}$ où r, g et b sont des nombres entre 0 et 1. L'argument ⟨*mode*⟩ peut être :
  - soit une des chaînes : `"x"`, `"y"`, `"z"`. Dans le premier cas par exemple, les facettes au centre de gravité d'abscisse minimale ont la première couleur de la palette, les facettes au centre de gravité d'abscisse maximale ont la dernière couleur de la palette, pour les autres, la couleur est calculée en fonction de l'abscisse du centre de gravité par

interpolation linéaire.

– soit une fonction : ⟨*mode*⟩ : $f \mapsto \text{mode}(f) \in \mathbb{R}$, où f désigne une facette (liste de points 3D). Les facettes ayant la valeur minimale ont la première couleur de la palette, celles ayant la valeur maximale ont la dernière couleur de la palette, pour les autres la couleur est calculée par interpolation linéaire.

L'argument facultatif ⟨*minmax*⟩ est une table de deux nombres, ces deux nombres sont utilisés comme valeur minimale et valeur maximale de l'argument ⟨*mode*⟩. En l'absence de ⟨*minmax*⟩, ces deux valeurs sont calculées automatiquement.

Quelques remarques

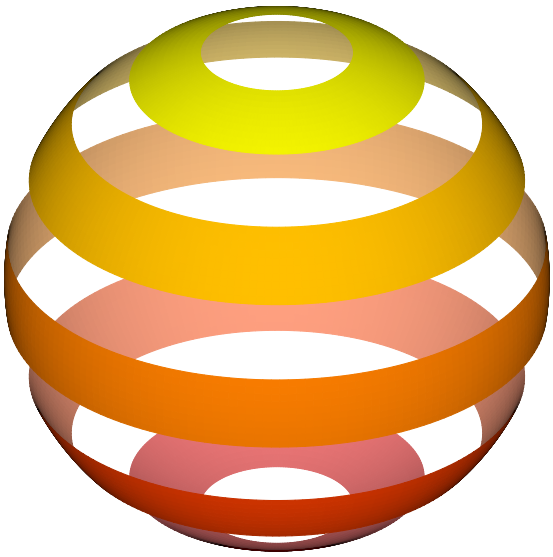
- Les options choisies pour une liste de facettes s'appliquent également aux suivantes si elles ne sont pas modifiées.
- Le dessin est fait avec l'opacité et le style de ligne par défaut. Ils peuvent être modifiés **avant** l'exécution, avec les méthodes **g:Fillopacity()** et **g:Linestyle()**.

```
\begin{luadraw}{name=Dliteralfacet}
local ld = luadraw
local M, Ms = ld.pt3d.M, ld.pt3d.Ms
local g = ld.graph3d:new{ size={10,10} }
require 'luadraw_pdfliteral'
local R, nb = 4, 5
local phi2, phi11, dphi = math.pi, nil, math.pi/(2*nb+1)

local sph_strip = function(p1,p2)
    return ld.surface( function(u,v) return Ms(R,u,v) end, -math.pi, math.pi, p1, p2,{201,11})
end

local colors = ld.getpalette( ld.palAutumn, nb,true)
local S = {}
for k = 1, nb do
    phi1 = phi2 -dphi; phi2 = phi1 - dphi
    table.insert(S, sph_strip(phi1,phi2))
    table.insert(S, {color=colors[k], mode=ld.mShadedOnly})
end
g:Dliteralfacet( table.unpack(S) ) -- 10^4 facets
g:Show()
\end{luadraw}
```

FIGURE 37 : Bandes sphériques



Remarque : il est possible de réaliser la même chose sans dessiner les facettes, mais c'est plus technique.

13) Le module *luadraw_linprog*

Ce module ne renvoie rien, il ajoute de nouvelles fonctions et méthodes graphiques aux classes *ld.graph* et *ld.graph3d*. L'objectif est la résolution et la représentation graphique de problèmes de programmations linéaires dans une fenêtre donnée.

Ce module charge le module *luadraw_decorations*.

En 2D

Il s'agit de représenter des systèmes de contraintes de la forme $ax+by<c$ ou $ax+by>c$ dans une fenêtre donnée, la solution est alors une liste de nombres complexes représentant le polygone convexe solution (dans la région donnée).

À partir de cette solution on peut chercher à optimiser une fonctionnelle du type $(x,y)\mapsto \alpha x+\beta y$ sur ce polygone, c'est à dire déterminer son minimum et son maximum sur le polygone.

Le calcul La fonction **ld.linprogSolve(constraints, x1, x2, y1, y2, objectives)** fait les calculs[4] et renvoie une séquence constituée de deux choses :

1. le polygone solution (liste de nombres complexes),
2. la liste des solutions pour chaque fonctionnelle.

L'argument ⟨*constraints*⟩ est la liste des contraintes, chaque contrainte est une chaîne de caractères de la forme `"a*x+b*y>c"` ou bien `"a*x+b*y<c"`, où a, b et c sont trois valeurs numériques, x et y sont deux lettres (le symbole de multiplication est obligatoire).

Les arguments ⟨*x1*⟩, ⟨*x2*⟩, ⟨*y1*⟩ et ⟨*y2*⟩ définissant la fenêtre de résolution : $[x_1;x_2]\times[y_1;y_2]$.

L'argument ⟨*objectives*⟩ est la liste des fonctionnelles à optimiser (elle peut être vide), chaque fonctionnelle est une chaîne de la forme `"u*x+v*y"` avec u et v deux valeurs numériques. La solution d'une fonctionnelle est une table dont les champs sont :

- `expr` : qui contient l'expression de la fonctionnelle,
- `min` : qui contient la valeur numérique du minimum,
- `mindot` : qui contient le point où le minimum est atteint (nombre complexe),
- `minline` : qui contient la droite passant par le point où le minimum est atteint,
- `max` : qui contient la valeur numérique du maximum,
- `maxdot` : qui contient le point où le maximum est atteint (nombre complexe),
- `maxline` : qui contient la droite passant par le point où le maximum est atteint,

Représentation graphique Suivants les conventions, certaines personnes dessinent et peignent le polygone solution, puis ajoutent les droites solution de l'optimisation pour chaque fonctionnelle (si il y en a) ; d'autres préfèrent peindre pour chaque contrainte, le demi-plan contenant les points qui ne vérifient pas celle-ci, ainsi le polygone solution est la partie non peinte qui reste.

- La méthode **g:DlinprogRegion(constraints [, options, objectives, objectives_options])** résout le système de contraintes contenu dans la liste ⟨*constraints*⟩, dessine le polygone solution, et les optimisations s'il y en a. L'argument ⟨*options*⟩ est la table des options possibles pour cette représentation, celles-ci sont (avec valeur valeur par défaut) :
  - `view=<fenêtre 2D par défaut>`, liste de la forme $\{x_1,x_2,y_1,y_2\}$ représentant la fenêtre de résolution : $[x_1;x_2]\times[y_1;y_2]$.
  - `draw_options=""`, chaîne contenant les options de dessin pour le polygone solution (chaîne transmise à *\draw*).
  - `lines=false`, avec la valeur `true` la droite correspondant à chaque contrainte (d'équation $ax+by=c$) est dessinée avec un label au milieu de la droite et orienté dans le sens de la droite.
  - `color=<couleur courante>`, couleur pour le tracé des droites et de leur label (si `lines=true`).
  - `width=<épaisseur courante>`, épaisseur de trait en dixième de point pour les droites (si `lines=true`).
  - `style=<style courante>`, style de tracé des droites (si `lines=true`).
  - `dots=false`, avec la valeur `true` les sommets du polygone solution sont dessinés.

4. Il s'agit d'une résolution géométrique. La fenêtre de résolution est un polygone qui est découpé avec les différentes droites correspondant aux contraintes.

- `mark_options=""`, chaîne contenant les options de dessin pour sommets lorsque `dots=true` (chaîne transmise à *\draw*).
- `outside=false`, avec la valeur `true`, c'est l'extérieur du polygone solution qui sera peint et non l'intérieur.
- `out=nil`, si on affecte à cette option `out` une variable de type liste (table), alors après l'exécution de la méthode, cette table contiendra : la liste des sommets du polygone solution, suivi de la liste des solutions de chaque fonctionnelle.
- À ces options s'ajoutent celles du module *luadraw_decorations* pour les lignes polygonales.

L'argument ⟨*objectives*⟩ est la liste des fonctionnelles à optimiser et à dessiner (droites réalisant le minimal et/ou le maximum).

L'argument ⟨*objectives_options*⟩ est la table des options possibles pour cette représentation, celle-ci sont (avec valeur valeur par défaut) :

- `color=<couleur courante>`, couleur pour le tracé de la droite et de son label.
- `width=<épaisseur courante>`, épaisseur de trait en dixième de point.
- `style=<style courante>`, style de tracé de la droite.
- `sense="minmax"`, autre valeurs possibles : "min", ou "max". Cette option permet de sélectionner la droite solution que l'on souhaite dessiner (les deux par défaut).
- `dots=false`, avec la valeur `true` le ou les points solutions sont dessinés (là où l'optimisation est réalisée).
- `mark_options=""`, chaîne contenant les options de dessin pour le ou les points solutions lorsque `dots=true` (chaîne transmise à *\draw*).
- Les options suivantes proviennent du module *luadraw_decorations* pour les droites :
  - `label=<auto>`, un label par défaut est affiché avec la droite (ou les droites), c'est l'expression de la fonctionnelle avec la valeur min ou max suivant les cas.
  - `node_options=<auto>`, chaîne contenant les options pour le label, par défaut il est de la couleur de la droite, sur fond blanc transparent.
  - `anchor=<auto>`, point d'ancrage du label, par défaut c'est le point où est réalisé l'extremum (on peut utiliser aussi `anchor1d`).
  - `dir=<auto>`, sens d'écriture du label, par défaut c'est la direction de la droite.
  - `pos=<auto>`, position du label par rapport à son point d'ancrage, par défaut le label est placé de l'autre côté du polygone solution.

- La méthode **g:DlinprogHalfPlanes(constraint1, options1, constraint2, options2, ...)**, permet de dessiner pour chaque contrainte le demi-plan non solution, chaque contrainte est une chaîne de la forme `"a*x+b*y>c"` ou bien `"a*x+b*y<c"`, et est suivie de sa liste d'options, celles-ci sont :
  - `color=<couleur courante>`, couleur pour le tracé de la droite, son label et le remplissage du demi-plan.
  - `width=<épaisseur courante>`, épaisseur de trait en dixième de point pour la droite.
  - `style=<style courante>`, style de tracé de la droite.
  - `pattern="none"`, motif de remplissage du demi-plan (aucun par défaut). Pour un remplissage solide on utilise `pattern="fill"`, sinon, tout autre motif de remplissage connu de TikZ.

    La valeur de ces quatre options pour une contrainte, s'applique également aux suivantes si elle n'est pas modifiée. Par contre les options suivantes sont propres à chaque contrainte, ce sont les options qui proviennent du module *luadraw_decorations* pour les droites :
    - `label=<auto>`, un label par défaut est affiché avec la droite.
    - `node_options=<auto>`, chaîne contenant les options pour le label, par défaut il est de la couleur de la droite, sur fond blanc transparent.
    - `anchor1d=0.5`, point d'ancrage du label, par défaut c'est le milieu de la droite (on peut utiliser aussi `anchor`).
    - `dir=<auto>`, sens d'écriture du label, par défaut c'est la direction de la droite.
    - `pos=<auto>`, position du label par rapport à son point d'ancrage, par défaut le label est placé de l'autre côté du polygone solution.
- La méthode **g:DlinprogObjectiveLine(objective_sol,objective_options)** permet de dessiner une optimisation. L'argument ⟨*objective*⟩ est la solution de optimisation d'une fonctionnelle, c'est à dire une table dont les champs sont :
  - `expr` : qui contient l'expression de la fonctionnelle,

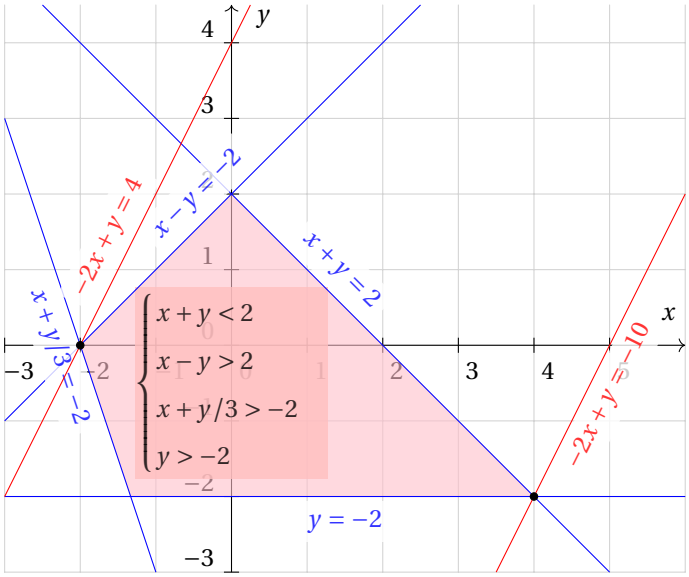
- `min` : qui contient la valeur numérique du minimum,
- `mindot` : qui contient le point où le minimum est atteint (nombre complexe),
- `minline` : qui contient la droite passant par le point où le minimum est atteint,
- `max` : qui contient la valeur numérique du maximum,
- `maxdot` : qui contient le point où le maximum est atteint (nombre complexe),
- `maxline` : qui contient la droite passant par le point où le maximum est atteint,

L'argument ⟨*objective_options*⟩ est la table des options possibles pour cette représentation, celles-ci ont déjà été décrites un peu plus haut avec la méthode **g:DlinprogRegion()**

Exemples

```
\begin{luadraw}{name=linear_prog1}
local ld = luadraw
local cpx = ld.cpx
local Z = ld.cpx.Z
local g = ld.graph:new{window={-3,6,-3,4.5}, size={10,10}}
require 'luadraw_linprog'
local system = \luastringO{$\begin{cases}x+y<2\\ x-y>2\\ x+y/3>-2\\ y>-2\end{cases}$}
g:Daxes({0,1,1},{grid=true,gridcolor="LightGray",arrows="->",legend={"$x$","$y$"}})
g:DlinprogRegion(
    {'x+y<2','x-y>-2','x+y/3>-2','y>-2'}, -- contraintes
    {draw_options="draw=none,fill=Pink, fill opacity=0.6",lines=true, color="blue", label=system, anchor=Z(0,-0.5),
    node_options="inner sep=0,draw=none,fill=pink, fill opacity=0.8" }, -- options
    {"-2*x+y"}, {color="red",dots=true, anchor1d=0.5} -- une fonctionnelle et ses options
    )
g:Show()
\end{luadraw}
```

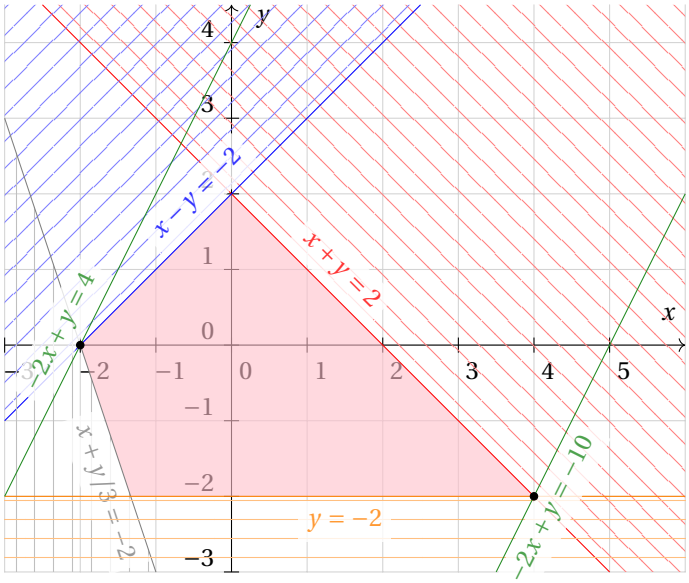
FIGURE 38 : La méthode *g:DlinprogRegion()*




```
\begin{luadraw}{name=linear_prog2}
local ld = luadraw
local cpx = ld.cpx
local Z = ld.cpx.Z
local g = ld.graph:new{window={-3,6,-3,4.5}, size={10,10}}
require 'luadraw_linprog'
local C = {'x+y<2','x-y>-2','x+y/3>-2','y>-2'} -- contraintes
local function mypattern(angle, distance)
    angle = angle or 0
    distance = distance or 0.25 -- distance in centimeter
    return "{Lines[angle="..angle..",distance="..distance.."cm]}"
end
```

```
local polygon, optimization = ld.linprogSolve(C,-5,5,-5,5,{"-2*x+y"})
g:Daxes({0,1,1},{grid=true,gridcolor="LightGray",arrows="->",legend={"$x$","$y$"}})
g:Dpolyline(polygon, true, "draw=none,fill=Pink, fill opacity=0.6")
g:DlinprogHalfPlanes(
    C[1], {color="red", pattern=mypattern(-45)},
    C[2], {color="blue", pattern=mypattern(45)},
    C[3], {color="gray", pattern=mypattern(90), anchor1d=0.2},
    C[4], {color="orange", pattern=mypattern(0)}
    )
for _, obj in ipairs(optimization) do
    g:DlinprogObjectiveLine(obj, {color="ForestGreen", dots=true})
end
g:Show()
\end{luadraw}
```

FIGURE 39 : La méthode *g:DlinprogHalfPlanes()*



En 3D

Il s'agit de représenter des systèmes de contraintes de la forme $ax + by + cz < d$ ou $ax + by + cz > d$ dans une fenêtre 3D donnée, la solution est un polyèdre convexe.

À partir de cette solution on peut chercher à optimiser une fonctionnelle du type $(x, y, z) \mapsto \alpha x + \beta y + \gamma z$ sur ce polyèdre, c'est à dire déterminer son minimum et son maximum sur le polyèdre.

La fonction **ld.linprogSolve3d(constraints, x1, x2, y1, y2, z1, z2, objectives)** fait les calculs[5] et renvoie une séquence constituée de deux choses :

1. le polyèdre solution (table avec deux champs : *vertices* et *facets*),
2. la liste des solutions pour chaque fonctionnelle.

L'argument ⟨*constraints*⟩ est la liste des contraintes, chaque contrainte est une chaîne de caractères de la forme `"a*x+b*y+c*z>d"` ou bien `"a*x+b*y+c*z<d"`, où a, b, c et d sont des valeurs numériques, x, y et z sont des lettres (le symbole de multiplication est obligatoire).

Les arguments ⟨*x1*⟩, ⟨*x2*⟩, ⟨*y1*⟩, ⟨*y2*⟩, ⟨*z1*⟩ et ⟨*z2*⟩ définissant la fenêtre de résolution : $[x_1; x_2] \times [y_1; y_2] \times [z_1; z_2]$.

L'argument ⟨*objectives*⟩ est la liste des fonctionnelles à optimiser (elle peut être vide), chaque fonctionnelle est une chaîne de la forme `"u*x+v*y+w*z"` avec u, v et w trois valeurs numériques. La solution d'une fonctionnelle est une table dont les champs sont :

- `expr` : qui contient l'expression de la fonctionnelle,
- `min` : qui contient la valeur numérique du minimum,
- `mindot` : qui contient le point où le minimum est atteint (nombre complexe),

5. Il s'agit d'une résolution géométrique. La fenêtre choisie est un polyèdre qui est découpé avec les différents plans correspondant aux contraintes.

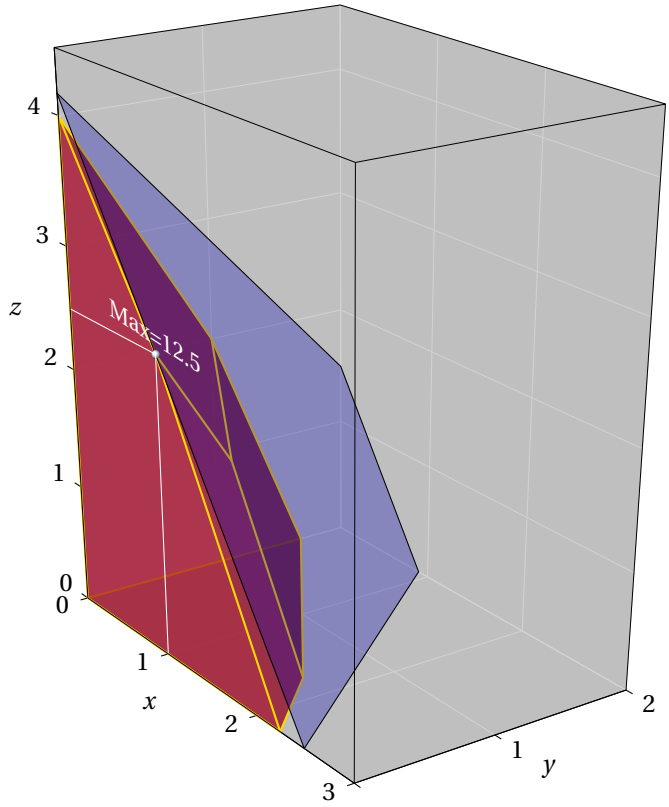
- `minplane` : qui contient le plan passant par le point où le minimum est atteint,
- `max` : qui contient la valeur numérique du maximum,
- `maxdot` : qui contient le point où le maximum est atteint (nombre complexe),
- `maxplane` : qui contient le plan passant par le point où le maximum est atteint,

Le module ne propose pas de méthode graphique spécifique à la 3D. Voici sur un exemple comment utiliser la fonction **ld.linprogSolve3d()**.

Exemple : Maximiser $5x+4y+3z$ avec : $\begin{cases} 2x+3y+z<5 \\ 4x+4y+2z<9 \\ 3x+4y+2z<8 \\ x>0,\ y>0,\ z>0 \end{cases}$

```
\begin{luadraw}{name=linear_prog3}
local ld = luadraw
local cpx = ld.cpx
local pt3d = ld.pt3d
local Z, M = cpx.Z, pt3d.M
local O, I, J, K = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK
local g = ld.graph3d:new{ window={-2,2,-3,3}, size={10,12,0},window3d={0,3,0,2,0,4.5},
    viewdir={"central",-35,65,15,M(1.5,1,2.25)}}
require 'luadraw_linprog'
local C = {'2*x+3*y+z<5', '4*x+4*y+2*z<9', '3*x+4*y+2*z<8'}
local poly, sol = ld.linprogSolve3d(C,0,5,0,5,0,5,{"5*x+4*y+3*z"})
local P = sol[1].maxplane
local A = sol[1].maxdot
local max = sol[1].max
g:Dboxaxes3d({grid=true,gridcolor="LightGray",fillcolor="lightgray", drawbox=true})
g:Dscene3d(
    g:addPlane(P, {color="blue", edge=true, opacity=0.3, scale=1}),
    g:addPoly(poly, {color="Crimson",opacity=0.7, edge=true, edgecolor="Gold", edgewidth=8}),
    g:addDots(A, {scale=0.75, color="white"})
    )
g:Dlabel3d("Max=$"..max.."$", A, {pos="N", dir={I,K}, node_options="text=white"})
g:Dpolyline3d({ld.pz(A), A, ld.px(A)}, "white")
g:Show()
\end{luadraw}
```

FIGURE 40 : La fonction *ld.linprogSolve3d()*



II Historique

1) Version 3.7

- Ajout de la fonction **ld.poly2graph()** qui permet de convertir un polyèdre en graphe.
- Ajout de la méthode **g:Dpolyline2()**, celle-ci fait la même chose que la méthode **g:Dpolyline()** sauf que la liste de points, en plus des nombres complexes, peut contenir des nœuds TikZ sous forme de chaînes, comme par exemple : `"(A.south)"`.
- Pour les méthodes **g:Cylinder_outline()**, **g:Cone_outline()** et **g:Frustum_outline()** : le résultat renvoyé contient deux champs supplémentaires : le champ *hiddenside* et le champ *hiddensection*.
- Dans le module *luadraw_decorations* : ajout de la méthode **g:Dangle3d()** qui dessine un angle droit dans l'espace avec des options pour peindre le parallélogramme et pour ajouter un label.
- Ajout de la méthode **g:Dmarkseg3d()** qui permet de marquer un segment de l'espace avec des petits segments.
- Ajout des nouveaux modes de projection en 3D : `"yx"`, `"zx"`, `"zy"`, `"osi"`, `"yOx"`, `"zOx"`, et `"zOy"`, qui sont les "symétriques" des modes `"xy"`, `"xz"`, `"yz"`, `"iso"`, `"xOy"`, `"xOz"`, et `"yOz"`.
- Ajout de la fonction **ld.cutfacet2()** qui permet de découper une liste de facettes (ou un polyèdre) non pas avec un plan (c'est le rôle de la fonction **ld.cutfacet()**), mais avec une surface implicite d'équation $f(x,y,z,) = 0$, sous certaines conditions sur f.
- Dans le module *luadraw_spherical*, ajout de trois méthodes : **g:Dhalf_sphere()**, **g:Dquater_sphere()**, et **g:Dheighth_sphere()**.
- Pour la méthode **g:Dcut_sphere**, ajout des options : `section=""`, qui permet de dessiner ou non la section circulaire, et `outline=false` qui permet de dessiner ou non le contour de la partie sphérique.
- Correction de bug...

2) Version 3.6

- Dans le module *luadraw_shadedforms* : ajout de l'option `bilinear=true/false` pour la méthode **g:Dshadedrectangle()** qui permet d'utiliser ou non un ombrage bilinéaire sur chaque maille (`true` par défaut).

- Ajout de la fonction **ld.pg_gonreg()** qui renvoie une liste de listes de nombres complexes représentant un polygone régulier étoilé.
- Dans le module *luadraw_spherical* :
  - La plupart des fonctions globales du module peuvent maintenant être appliquées à une autre sphère que la sphère courante.
  - Ajout de la méthode **g:Split_facet()** qui calcule et renvoie sous forme de chemins 3D le contour de la partie visible de la facette sphérique et le contour de la partie cachée.
  - Ajout de la méthode **g:Split_region()** qui calcule et renvoie sous forme de chemins 3D le contour de la partie visible de la région sphérique et le contour de la partie cachée.
  - Pour les méthodes **g:DSfacet()** et **g:DSregion()**, deux options sont ajoutées : `front` et `back`, celles-ci permettent de récupérer en sortie, le chemin correspondant au contour de la partie visible pour la première option, et le chemin correspondant au contour de la partie cachée pour la deuxième option.
  - La plupart des éléments graphiques de ce module ont maintenant des options pour ajouter un label.
- Pour la fonction **ld.section2tube()**, ajout de l'option `opensection=true/false`, avec la valeur `true` la section ne sera pas refermée, c'est à dire qu'il n'y aura pas de facettes entre le dernier et le premier point de la section, le tube aura une ouverture latérale.
- Pour la fonction **ld.line2tube()**, ajout de l'option `opensection=true/false` (comme ci-dessus), et de l'option `angle={angle1,angle2}`, avec cette option on peut modifier l'angle de départ et d'arrivée (en degrés) pour que la section ne soit qu'une partie d'un cercle.
- Ajout de trois options pour la méthode **g:BeginOnPlane()** : l'option `border=true/false` et l'option `border_options`, qui permettent de dessiner un cadre autour de la fenêtre 2D choisie dans le plan, ainsi que l'option `tikzmatrix=true/false` qui permet d'utiliser ou non la matrice de transformation de TikZ pour cette méthode.
- Ajout de la fonction **ld.parabola(A, B, C)** qui renvoie une courbe de Bézier représentant un arc de parabole allant du point $\langle A\rangle$ au point $\langle C\rangle$, en passant par le point $\langle B\rangle$ (trois nombres complexes).
- Ajout de la fonction **ld.param_path(P)** qui renvoie une fonction représentant une paramétrisation du chemin 2D $\langle P\rangle$.
- Ajout de la fonction **ld.splitpath()** qui permet de couper un chemin en un point donné, en deux chemins.
- Ajout de la fonction **ld.common_tangent()** et de la méthode de dessin **g:Dcommon_tangent()** correspondante, celle-ci permet de dessiner les tangentes communes à deux courbes paramétrées.
- Ajout de la fonction **ld.parallel_tangent()** et de la méthode de dessin **g:Dparallel_tangent()** correspondante, celle-ci permet de dessiner les tangentes à une courbe paramétrée, qui sont parallèles à la tangente en un point de paramètre donné, ou parallèles à une direction donnée.
- L'argument optionnel de la fonction **ld.solve()** peut désormais prendre une valeur supplémentaire : `"bisection"`, auquel cas la méthode utilisée sera une dichotomie (cela peut être utile en cas d'échec de la méthode Newton).
- Ajout de la fonction **ld.getmatrix(T1, T2)** qui calcule et renvoie la matrice transformant le triangle $\langle T1\rangle$ (liste de trois points 2D non alignés) en le triangle $\langle T2\rangle$ (liste de trois points 2D).
- Ajout de la fonction **ld.getmatrix3d(T1, T2)** qui calcule et renvoie la matrice transformant le tétraèdre $\langle T1\rangle$ (liste de quatre points 3D non coplanaires) en le tétraèdre $\langle T2\rangle$ (liste de quatre points 3D).
- Correction de bug...

3) Version 3.5

- Dans le module *luadraw_spherical* :
  - La méthode **g:Define_sphere()** a trois options supplémentaires : `back=true/false`, `inside=true/false`, et `front=true/false`. Elles permettent d'afficher ou non ce qui est derrière la sphère, ce qui est à l'intérieur de la sphère, et ce qui devant la sphère.
  - Ajout de la méthode **g:DSaxes()** qui permet de dessiner les trois axes.
  - Pour la plupart des méthodes graphiques, il y a des options supplémentaires pour ajouter un label.
  - Ajout des fonctions **ld.smidpoint()** et **ld.sbarycenter()**.
- Dans le module *luadraw_povray* :
  - ajout de l'option `clip=<booléen>` pour la méthode **g:Pov_polyline()** pour clipper ou non la ligne polygonale avec la fenêtre 3D courante.

  - ajout de l'option `csg=true/false` pour la méthode **g:Pov_facet()**, avec la valeur `true` l'objet défini peut être utilisé dans des opérations de CSG (sous certaines réserves).
- Pour les chemins 3D : ajout des instructions *"e"* pour ellipse, et *"ea"* pour arc d'ellipse.
- Nouvelles fonctions 3D pour les ellipses et arc d'ellipses : **ld.ellipse3db()**, **ld.ellipse3d()**, **ld.ellipticarc3db()**, **ld.ellipticarc3d()**, ainsi que les méthodes de dessin associées : **g:Dellipse3d()** et **g:Dellipticarc3d()**.
- Ajout de la méthode **g:Dcut_sphere()** qui permet de dessiner une sphère qui a été coupée par un plan.
- Ajout de l'option `apex=true/false` pour la méthode **g:Dcone()**, avec la valeur `true` le sommet est matérialisé par un point lorsque le cône est vu de dessus.
- La méthode **g:Saveattr()** accepte maintenant un argument : **g:Saveattr(scope_options)**, celui-ci est une chaîne (vide par défaut) qui sera transmise à l'instruction *scope* : `\begin{scope}[<scope_options>]`.
- Ajout de l'option `lowlevel=true/false` pour la méthode **g:BeginOnPlane()**, avec la valeur `true`, la matrice de transformation est appliquée au plus bas niveau, ainsi tous les éléments graphiques sont dessinés dans le plan voulu : flèches, texte, etc.
- Ajout du module *luadraw_linprog*, celui-ci permet la résolution et la représentation graphique de la solution de problèmes de programmation linéaire en 2D ou en 3D.
- Ajout de l'option `gradient=true/false` pour les méthodes **g:Dcylinder()**, **g:Dcone()** et **g:Dfrustum()**. Celle-ci indique si le remplissage doit être fait avec un gradient ou pas (`true` par défaut).
- Correction de bug...

4) Version 3.4

- Pour la méthode **g:Dfrustum()**, l'ancienne syntaxe **g:Dfrustum(C,R,r,V,options)** a été modifiée en **g:Dfrustum(C,R,r,A,options)**, la différence est que le centre de la deuxième base circulaire est le point ⟨*A*⟩, alors que dans l'ancienne syntaxe c'était le point C + V. De même pour la fonction **ld.frustum()**, l'ancienne syntaxe **ld.frustum(C,R,r,V,nb,open)** devient **ld.frustum(C,R,r,A,nb,open)** avec la même différence. Ceci afin d'avoir une syntaxe homogène pour les cylindres, les cônes et les cônes tronqués.
- Pour la méthode **g:Beginclip(chemin, inverse)**, le ⟨*chemin*⟩ peut être un chemin 2D ou 3D.
- Dans le module *luadraw_spherical* : ajout de la méthode **g:DSregion()** qui permet de peindre une région de la sphère délimitée par une courbe sphérique fermée simple.
- Ajout d'arguments optionnels à la méthode **g:Endclip([draw_path, draw_options])**, ⟨*draw_path*⟩ est un booléen (`false` par défaut), lorsqu'il a la valeur `true`, le chemin qui a servi à clipper est dessiné après la fermeture du groupe, et dans ce cas, l'argument optionnel ⟨*draw_options*⟩, qui est une chaîne de caractères vide par défaut, est transmise à l'instruction *draw*.
- Ajout de fonctions spécifiques aux chemins 2D ou 3D : **ld.path()**, **ld.path3d()**, **ld.convpath()**, **ld.polyline2path()**, **ld.polyline2path3d()**, et les méthodes **g:Convpath3d()**, **g:Path3d2path2d()**.
- Ajout des méthodes **g:Cylinder_outline()**, **g:Cone_outline()**, **g:Frustum_outline()**, et **g:Sphere_outline()** qui renvoient une table contenant les éléments caractéristiques (contour, arêtes visibles, arêtes cachées, ...).
- Ajout des méthodes **g:Cylinder_tangency()**, **g:Cone_tangency()**, **g:Frustum_tangency()** et **g:Sphere_tangency()** qui calculent et renvoient la liste des points de tangence pour les cylindres, les cônes, les cônes tronqués et les cercles tracés sur une sphère.
- Pour la fonction **ld.read_table3d()**, ajout de l'option `mode` permettant de choisir un type de table 3D (types `"xyz"`, ou `"y/x"` ou `"x/y"`.
- L'option `usepalette` (pour le dessin de facettes) accepte désormais un troisième argument (facultatif) :

  `usepalette={palette, mode, minmax}`,

  où ⟨*minmax*⟩ est une table de deux nombres, ces deux nombres sont utilisés comme valeur minimale et valeur maximale de l'argument ⟨*mode*⟩. En l'absence de ⟨*minmax*⟩, ces deux valeurs sont calculées automatiquement.
- Correction de bug...

5) Version 3.3

- Ajout de la fonction **ld.read_table3d()** permettant de lire et exploiter un tableau (liste de listes) contenant les valeurs d'une (ou plusieurs) fonction(s) de deux variables sur un pavé $[x_1;x_2]\times[y_1;y_2]$.

- Ajout de l'option `reverse=true/false` dans les méthodes de dessins de facettes (**g:Dfacet()**, **g:Dmixfacet()**, **g:Dpoly()**, **g:addFacet()**, …), avec la valeur `true`, l'orientation des facettes est inversée.
- Ajout du module *luadraw_pdfliteral*, celui-ci permet de dessiner directement dans le flux PDF : des chemins 2D, des lignes polygonales 2D, des ensembles de points 2D (circulaires) ou des facettes 3D, sans utiliser la commande *\draw*, ce qui réduit notablement le temps de compilation lorsque les données sont en grand nombre, ceci sous certaines réserves : pas de changement d'opacité, et seulement deux styles de remplissage : aucun ou solide.
- Dans le module *luadraw_spherical*, ajout de trois nouvelles méthodes : **g:DSaddback()**, **g:DSaddinside()** et **g:DSaddfront()** qui permettent d'ajouter des éléments graphiques (chemins ou lignes polygonales), sur la face arrière de la sphère, à l'intérieur, ou sur la face avant.
- Dans les méthodes **g:Dfacet()** et **g:Dmixfacet()**, en mode *ld.mWireframe* (dessin des arêtes uniquement), si l'option `usepalette` est différente de `nil`, alors la couleur de chaque arête et calculée dans la palette suivant le même mode que les facettes lorsque celles-ci sont peintes.
- Ajout de l'option `out=<variable>` à la méthode **g:BeginOnPlane()**, elle permet de récupérer une matrice 2D correspondant aux nouveaux axes.
- Ajout de l'option `graphics_options` pour la méthode **g:Dmapimage()**. Ajout de l'option `node_options` pour les méthodes **g:Dmapimage()** et **g:Dimage()**.
- Pour la méthode **g:addPolyline()**, il y a une nouvelle option (expérimentale) qui s'appelle `double={border color, border width}`, celle-ci permet d'ajouter une bordure de chaque côté de la ligne polygonale. Avec la valeur par défaut (`nil`) il n'y a pas de bordure.
- La syntaxe des fonctions **ld.lineEq()** et **ld.planeEq()** a été étendue pour prendre en compte une éventuelle inégalité.
- Correction de bug…

6) Version 3.2

- Ajout des méthodes **g:BeginOnplane()** et **g:EndOnPlane()**, celles-ci permettent de dessiner sur un plan de l'espace en utilisant les méthodes graphiques 2D.
- Dans le module *luadraw_fields* ont été ajoutés les champs de vecteurs tangents à une surface[6]. La fonction **ld.surfacefield()** calcule et renvoie le champ de vecteurs, la méthode **g:Dsurfacefield()** permet le dessin du champ de vecteurs avec (ou sans) la surface.
- La fonction **ld.linspace()** a une deuxième syntaxe possible : **ld.linspace(a1, b1, n1, b2, n2, …, bp, np)** qui renvoie une liste de $\langle n1\rangle$ nombres équirépartis de $\langle a1\rangle$ jusqu'à $\langle b1\rangle$, suivis de $\langle n2\rangle$ nombres équirépartis de $\langle b1\rangle$ jusqu'à $\langle b2\rangle$ (mais $\langle b1\rangle$ n'est pas répété), etc.
- Dans le module *luadraw_spherical*, ajout des fonctions :
  - **ld.interGreatC(C1,C2)** qui renvoie sous forme d'une séquence, les deux points d'intersection des deux grands cercles $\langle C1\rangle$ et $\langle C2\rangle$ appartenant à la sphère.
  - **ld.interSphericalC(P1, P2)** qui renvoie sous forme d'une séquence, les points d'intersection (s'ils existent) entre deux cercles appartenant à la sphère (pas nécessairement des grands cercles).
  - **ld.projstereo_Scircle(P, N, h)** qui renvoie sous forme de chemin la projection stéréographique d'un cercle tracé sur la sphère.
  - **ld.projstereo_Sfacet(L, N, h [, close])** qui renvoie sous forme de chemin la projection stéréographique d'une facette sphérique.
- Ajout des trois options dans la méthode **g:Dboxaxes3d()** : `xlabels={x1,…,xn}`, `ylabels={y1,…,yn}`, `zlabels={z1,…,zn}`. Ces options permettent d'imposer les labels sur les axes. Par défaut ces options ont la valeur `nil`, dans cas ce sont les labels par défaut (un par graduation) qui sont affichés.
- Dans le module *luadraw_povray*, il y a une nouvelle option dans les réglages par défaut : `arrowscale={1,1}`, qui est une table de deux nombres, le premier est un facteur d'échelle pour le rayon de la base des flèches (celles-ci sont des cônes), le second est un facteur d'échelle pour la hauteur des flèches.

  Dans les méthodes **g:Pov_polyline()** et **g:addPolyline()**, l'option du même nom peut désormais être soit une table de deux nombres, soit un seul nombre (dans ce cas les deux nombres sont considérés comme égaux).
- Pour la 3D, il y a une nouvelle variable globale `ld.Hiddenlinescale` qui vaut `2/3` par défaut, cela signifie que l'épaisseur des lignes cachées sera égale à celle des lignes visibles, multipliée par ce nombre, lorsqu'on utilise la

6. Sur une initiative d'Explorer.

méthode **g:Dscene3d()**.

- Correction de bug...

7) Version 3.1

- Ajout de la fonction **ld.implicit_inequality()** qui renvoie un **chemin** représentant le contour de la partie du plan située dans un certain pavé et vérifiant une condition du type $f(x,y) >= 0$ ou $f(x,y) <= 0$.
- Ajout de la méthode **g:Dimplicit_inequalities()** qui peut peindre l'ensemble des points vérifiant un système de contraintes du type $f_i(x,y) > 0$ ou $f_i(x,y) < 0$.
- Ajout de la fonction **plane2rectangle(P, V, L1, L2)** qui renvoie un rectangle (liste de points 3D) qui permet de représenter ce plan, c'est ce même rectangle que dessine la méthode **g:Dplane(P, V, L1, L2)**. Il peut être dessiné avec la méthode **g:Dpolyline3d()** ou bien être dessiné en tant que facette.
- Pour les méthodes **g:addPlane()** et **g:addPlaneEq()** (qui s'utilisent dans **g:Dscene3d()**), ajout de l'option `rectangle`. Lorsque `rectangle=nil` (valeur par défaut) le plan est coupé par la fenêtre 3D et la facette qui en résulte est dessinée. Lorsque `rectangle={V,L1,L2}`, le plan est dessiné sous la forme d'une facette rectangulaire, c'est le même rectangle que dessine la méthode **g:Dplane(P, V, L1, L2)** (où P désigne le plan).
- Lors de la création d'un graphe 3D, l'option `window3d` prend maintenant en compte les échelles sur les trois axes :

  `window3d={x1,x2,y1,y2,z1,z2 [,xscale,yscale,zscale]}`.

  Ces trois échelles sont facultatives et valent 1 par défaut, elles déterminent la matrice 3D initiale du graphe (qui n'est donc plus l'identité si une des échelles est différente de 1).
- Lors de la création d'un graphe, l'option `margin` peut maintenant être réduite à un seul nombre lorsque les quatre marges sont égales, par exemple `margin=0`.
- Ajout de l'option `useclip=<booléen>` pour la méthode **g:Dinequalities()** qui permet de choisir la technique à utiliser, soit le calcul du contour (avec la valeur `false`, valeur par défaut), soit avec une série de clips (avec la valeur `true`).
- La fonction **rectangle()** a désormais une deuxième syntaxe possible : **rectangle(a, b)**, dans ce cas le rectangle a ses côtés parallèles aux axes et admet comme sommets opposés $\langle a \rangle$ et $\langle b \rangle$. Il en va de même pour la méthode graphique correspondante **g:Drectangle()**.
- Pour la fonction **line2strip()**, l'argument $\langle ends \rangle$ qui était jusque là un booléen, peut prendre maintenant les valeurs `"none"` (équivalent à `false`), `"butt"` (équivalent à `true`), ou bien `"round"`. Les valeurs booléennes restent néanmoins acceptées.
- La fonction **line2strip()** a maintenant argument en plus appelé `mode`, celui-ci peut valoir `"center"` (valeur par défaut), ou bien `"left"`, ou bien `"right"`, dans le premier cas la bande est centrée sur la ligne polygonale, dans le second cas elle est sur la gauche de la ligne, et dans le troisième cas elle est sur la droite de la ligne.
- Dans la fonction **read_csv_file()**, ajout de l'option `comment=<char>`, celle-ci indique les caractères qui commencent une ligne de commentaires.
- Toutes les méthodes de dessin de droites ou demi-droites (**g:Dline()**, **g:Dperp()**,...) ont maintenant une option `scale` comme la méthode **g:Dseg()**. Cette option (qui veut 1 par défaut) peut être un nombre (pourcentage), soit une table de deux nombres (pourcentages), le second cas permet de faire varier séparément les deux extrémités.
- Correction de bug...

8) Version 3.0

Cette version amène un changement majeur : toutes les données relatives au paquet *luadraw* sont désormais dans l'espace de noms (table) nommé *luadraw*, ce qui oblige à utiliser la notation pointée, par exemple `luadraw.graph` à la place de `graph`, mais il est possible de créer des raccourcis, par exemple l'instruction `local ld = luadraw` vous permettra d'utiliser *ld* à la place de *luadraw* dans la notation pointée. Reportez vous au tout début de ce document pour avoir plus de détails. Ce changement a pour conséquence une incompatibilité avec les versions antérieures, cependant les changements à effectuer pour adapter les anciens codes sont minimes, d'autant plus qu'il n'y a aucun changement en ce qui concerne les méthodes graphiques (elles étaient déjà encapsulées dans deux classes). Ce changement amène également quelques modifications (mineures) pour les extensions, reportez-vous à la documentation pour plus de détails.

- Ajout d'une deuxième syntaxe possible pour les fonctions **ld.surface()** et **ld.obj_surface()** : **ld.surface(f, mesh)** et

ld.obj_surface(f, mesh) où $\langle mesh\rangle = \{\{u_1,\dots,u_n\},\{v_1,\dots,v_m\}\}$ (liste croissante des valeurs du paramètre u et liste croissante des valeurs du paramètre v).

- Dans le module *luadraw_decorations* : les méthodes graphiques de dessin de lignes 2D et 3D ont été surchargées de manière à ce que l'argument $\langle draw_options\rangle$, qui est normalement une chaîne de caractères transmise à l'instruction *\draw*, puisse être remplacé par une table dont les champs représentent des options possibles (comme par exemple ajouter un label). Les noms des méthodes sont inchangés et l'ancienne syntaxe reste toujours valable.
- Dans les fonctions **ld.curve2cone()**, **ld.curve2cylinder()**, **ld.line2tube()**, **ld.section2tube()**, **ld.rotcurve()** et **ld.rotline()**, ajout de l'option `obj=true/false`; avec la valeur par `false` (par défaut) les fonctions renvoient une liste de facettes, avec la valeur `true` elles renvoient une table `{vertices={3D points}, facets={{index1,…},…}, normals={3D vectors}}`. Si la méthode **g:Dpoly()** ne prend pas en compte le champ *normals*, la méthode **g:Pov_facet()** du module *luadraw_povray* par contre, utilise ce champ lorsqu'il est présent.
- Ajout de la fonction **ld.cutpolyline2(P,f,sg,close)** où $\langle P\rangle$ est un polygone (liste de nombres complexes), $\langle f\rangle$ une fonction ($x\mapsto f(x)\in\mathbf{R}$) et $\langle sg\rangle$ une chaîne qui vaut `">"` ou `"<"`. Cette fonction renvoie le contour de la partie du polygone vérifiant la contrainte $y>f(x)$ ou bien $y<f(x)$ en fonction de la valeur de $\langle sg\rangle$.
- Ajout de la méthode **g:Dinequalities(constraints, options)** qui dessine l'ensemble des points vérifiant un système de contraintes du type $y>f_i(x)$ ou $y<f_i(x)$.
- Dans le module *luadraw_povray* : ajout de l'option `usepalette={palette, mode, minmax}`.
- Correction de bug...

9) Version 2.8

Liste non exhaustive :

- Ajout des fonctions : **nth_root(n,x)** (racine énième d'un réel x, définie sur $\mathbb{R}$ lorsque n est impair); **cpx.cosh()**, **cpx.sinh()** (cosinus et sinus hyperboliques complexes) et **cpx.pow(z,a)** (pour le calcul de z^a avec z nombre complexe et a un réel).
- Ajout de l'extension *luadraw_coils_chains* qui permet de dessiner des ressorts et des chaînes.
- Ajout de l'extension *luadraw_log_axes* qui permet de créer et de dessiner sur grille logarithmique en x, ou en y ou en x et y.
- Ajout de l'option `use_siunitx` pour les méthodes **g:Daxes()**, **g:DaxeX()**, **g:DaxeY()**, **g:Dgradbox()**, **g:Dgradline()**. Elle permet localement d'utiliser ou non le formatage des valeurs numériques par le paquet *siunitx*.
- Ajout de l'option `showlines` pour la méthode **g:Dgrid()** qui permet d'afficher ou non les traits horizontaux et/ou les traits verticaux.
- Ajout de l'extension *luadraw_decorations* qui permet d'enrichir certaines méthodes de dessin en leur ajoutant des options. Pour le moment il y a **g:Ddecoratedarc()** pour les arcs 2D, et **g:Ddecoratedarc3d()** pour les arcs 3D.
- Pour les méthodes **g:Dpoly()**, **g:Dfacet()**, **g:Dmixfacet()** et **g:addFacet()**, dans l'option `usepalette={palette, mode}` le deuxième argument peut maintenant être une fonction $\langle mode\rangle$: $f\mapsto \text{mode}(f)\in\mathbb{R}$, où f désigne une facette (liste de points 3D). Les facettes ayant la valeur minimale ont la première couleur de la palette, celles ayant la valeur minimale ont la dernière couleur de la palette, pur les autres la couleur est calculée par interpolation linéaire. Jusque là l'argument $\langle mode\rangle$ ne pouvait valoir que `"x"`, `"y"` ou `"z"`.
- Ajout de la fonction **obj_surface(f,u1,u2,v1,v2 [, grid])** qui renvoie la surface paramétrée par $\langle f\rangle$: $(u,v)\mapsto f(u,v)\in\mathbf{R}^3$ au format *obj*, c'est à dire une table avec trois champs {vertices={…}, facets={{…},…}, normals={…}}, les deux premiers champs sont identiques au cas des polyèdres, le troisième champ contient les vecteurs unitaires normaux à la surface à chaque sommet. Les facettes sont triangulaires.
- Dans le module *luadraw_povray* : ajout d'une deuxième syntaxe pour dessiner des surfaces paramétriques lisses :

  g:Pov_surface(f,u1,u2,v1,v2,options)

  où $\langle f\rangle$: $(u,v)\mapsto f(u,v)\in\mathbf{R}^3$ est le paramétrage. Cette méthode est plus rapide que la précédente.
- Dans le module *luadraw_spherical* : ajout de la variable globale `Hiddendelayed` = `false`. Avec la valeur `false` les lignes cachées sont dessinées à la fin de l'instruction **g:Dspherical()**, avec la valeur `true` elles sont dessinées à la toute fin du graphique en cours ce qui peut être utile si vous avez ajouté après la sphère des éléments qui cachent une partie de celle-ci.
- Pour la méthode **g:Daxes()** : l'option `originloc` est toujours le point servant d'origine pour les graduations, mais il n'est plus automatiquement le point d'intersection des deux axes.

- Pour la méthode **g:Daxes()** : ajout des options `xnode_options = ""`, `xnode_options = xynode_options` et `ynode_options = xynode_options` qui permettent de passer des options l'instruction `\node{}` pour tous les labels (sauf les légendes).
- Pour la méthode **g:addAxes()**, ajout de l'option `labels={"$x$","$y$","$z$"}` pour gérer les labels affichés à l'extrémité de chaque axe.
- Correction de bug...

10) Version 2.7

Liste non exhaustive :

- Les méthodes de dessin des solides de base : **g:Dcylinder()**, **g:Dcone()** et **g:Dfrustum()** ont désormais deux options supplémentaires : `edgestyle` et `edgewidth` (comme pour la méthode **g:Dsphere()**).
- Dans le module *luadraw_compile_tex*, pour les méthodes :
  g:Dcompiled_tex(L, anchor, options) et **g:Compiled_tex2path3d(L, options)**,
  ajout de l'option `pos` identique aux labels.
- Dans le module *luadraw_compile_tex*, ajout de trois variables globales pour gérer l'accès aux programmes *pdflatex*, *pdf2ps* et emphpstoedit. Ces variables sont `pdflatexcmd`, `pdf2pscmd` et `pstoeditcmd`, elles permettent d'ajouter éventuellement un chemin d'accès au programme, par exemple : `pstoeditcmd = "/usr/bin/pstoedit"`.
- Nouvelle syntaxe pour la fonction **circle(data, nbdots)**, où ⟨*data*⟩ est une liste (centre et rayon, ou bien trois points du cercle) et ⟨*nbdots*⟩ le nombre de points souhaité. Les anciennes syntaxes restent valables.
- Nouvelle syntaxe pour la fonction **ellipse(data, nbdots)**, où ⟨*data*⟩ est une liste : {centre, rx, ry, inclinaison}, et ⟨*nbdots*⟩ le nombre de points souhaité. Les anciennes syntaxes restent valables.
- Ajout de la fonction **mixpalette(pal, percent, color)** qui renvoie une nouvelle palette après avoir mixer chaque couleur de la palette ⟨*pal*⟩ avec ⟨*color*⟩.
- Correction de nombreuses coquilles dans la documentation.
- Correction de bug...

11) Version 2.6

Liste non exhaustive :

- Ajout de l'extension *luadraw_povray* qui permet de créer une image avec Pov-Ray et de l'inclure dans le graphique en cours pour dessiner par dessus.
- Ajout de l'extension *luadraw_fields* qui permet le dessin de champs de vecteurs ou de champs de gradient.
- Ajout de l'extension *luadraw_shadedforms* qui permet de dessiner des lignes polygonales 2D ou de remplir une forme, avec un dégradé de couleurs en fonction de la méthode de calcul et de la palette choisies.
- La méthode **g:Dshadedpolyline()** fait désormais partie de l'extension *luadraw_shadedforms*.
- Ajout des méthodes **g:Dimage()** et **g:Dmapimage()**, la première permet l'inclusion d'une image dans le graphique, la deuxième permet de mapper une image sur un parallélogramme.
- Ajout de la fonction **parallelogram()** qui renvoie les sommets d'un parallélogramme construit à partir d'un sommet et de deux vecteurs. Ajout également de la méthode de dessin **g:Dparallelogram()** correspondante.
- Ajout des options `gradside` et `gradsection` qui permettent de modifier les paramètres du gradient dans le dessin des cylindres, des cônes et des cônes tronqués (méthodes **g:Dcylinder()**, **g:Dcone()** et **g:Dfrustum()**.
- Extension de la méthode **g:Newcolor(name, color)**, le second argument peut maintenant être soit une table de trois composantes RGB, soit une chaîne de caractères représentant une couleur.
- Correction de bug...

12) Version 2.5

Liste non exhaustive :

- Ajout de la fonction **read_csv_file()**[7] qui permet de lire un fichier *csv* avec différentes options.
- L'extension *luadraw_palettes* est passée à la version 1.3.0 du projet `@prism` de Christophe BAL.

7. Sur une idée de Christophe BAL.

- Ajout de la méthode **g:Dshadedpolyline()** qui permet de dessiner une ligne polygonale 2D avec un dégradé de couleurs en fonction de la méthode de calcul et de la palette choisies.
- Ajout de la méthode **g:Dpolynames()** qui permet d'afficher un polyèdre avec le numéro des faces et/ou ceux des sommets.
- Ajout de l'extension *luadraw_cvx_polyhedra_nets* qui permet de déterminer un patron des polyèdres convexes.
- Correction de bug...

13) Version 2.4

Liste non exhaustive :

- Ajout de la projection centrale.
- Ajout de l'option `legendstyle` pour les axes, pour imposer un style de label ("auto", "N", "E", ...) pour les légendes lorsqu'il y en a (jusque là, le style était forcément "auto").
- Ajout de la méthode **g:Labeldir()** qui permet de gérer globalement le sens de l'écriture.
- Ajout des fonctions **interCS()** (intersection entre un cercle dans l'espace et une sphère), et de la fonction **interSSS()** (intersection entre 3 sphères).
- Ajout de la fonction **voronoi()** en complément de la triangulation de Delaunay, elle permet de faire des diagrammes de Voronoï.
- Ajout de la fonction **parallel_polyline()** qui renvoie une ligne polygonale parallèle.
- Ajout de la fonction **tangent_from()** et de la méthode **g:Dtangent_from()** qui permet de tracer les tangentes à une courbe donnée issues d'un point donné.
- Correction de bug...

14) Version 2.3

Liste non exhaustive :

- Ajout des projections en perspective cavalière : sur yz, sur xz ou sur xy, ainsi que de la projection isométrique.
- Ajout de la fonction **section2tube()**.
- Ajout du module *luadraw_compile_tex*.
- Ajout de la méthode **g:Proj3dV()** pour le calcul de la projection des vecteurs de l'espace sur le plan de l'écran.
- Ajout des fonctions **circumcircle()** et **incircle()** en 2D, elles renvoient une séquence : centre et rayon.
- Ajout de la fonction **line2strip()** qui renvoie un chemin représentant une "bande" centrée sur une ligne polygonale donnée.
- Ajout de la fonction **delaunay()** qui fait une triangulation de Delaunay sur une liste de points et renvoie la liste des triangles obtenus.
- Ajout de la fonction **cpx.normalize(z)** qui renvoie le complexe z divisé par son module (ou `nil` s'il est nul).
- Ajout de l'instruction **whatis(variable, msg)** qui affiche dans le terminal le statut d'une ⟨*variable*⟩ (accompagné du message ⟨*msg*⟩) et son contenu.
- Correction de bug...

15) Version 2.2

Liste non exhaustive :

- Ajout de l'option `clip` pour les méthodes : **g:Dfacet()**, **g:Dmixfacet()**, **g:addFacet()**, **g:addPoly()** et **g:addPolyline()**, ainsi que pour les méthodes de dessin de nuages de points, et les méthodes de dessin "au trait" comme **g:Dpolyline3d()**, **g:Dparametric3d()**, **g:Dpath3d()**, etc.
- Ajout de l'option `xyzstep` pour la méthode **g:Dboxaxes3d()**, cette option définit un pas commun aux trois axes (1 par défaut).
- Ajout des méthodes **g:DSdots()**, **g:DSstars()**, **g:DSinvstereo_curve()** et **g:DSinvstereo_polyline()** dans le module *luadraw_spherical*.
- Ajout du module *luadraw_palettes*.
- Ajout de la fonction **interDC()** (intersection entre une droite et un cercle en 2D) et de la fonction **interCC()** (intersection entre 2 cercles en 2D).

- Ajout des fonctions **curvilinear_param()** et **curvilinear_param3d()** qui permettent d'obtenir une paramétrisation d'une liste de points (2D pour l'une, et 3D pour l'autre) avec une fonction d'une variable t entre 0 et 1.
- Ajout de la fonction **cvx_hull2d()** qui renvoie l'enveloppe convexe (ligne polygonale) d'une liste de points en 2D, et de la fonction **cvx_hull3d()** qui renvoie l'enveloppe convexe (liste de facettes) d'une liste de points en 3D.
- Ajout des méthodes **g:Beginclip(chemin)** et **g:Endclip()** qui facilitent la mise en place d'un clipping par TikZ.
- Ajout des fonctions **normal()**, **normalC()**, **normalI()** qui renvoient la normale à une courbe 2D en un point donné. Les méthodes graphiques correspondantes ont également été ajoutées.
- Ajout de la fonction **isobar()** qui renvoie l'isobarycentre d'une liste de complexes.
- Ajout de l'option `usepalette={palette, mode}` pour les méthodes **g:Dpoly()**, **g:Dfacet()**, **g:Dmixfacet()**, **g:addFacet()**.
- Ajout de la fonction **clipplane()** qui permet de clipper un plan avec un polyèdre convexe, la fonction renvoie la section, si elle existe, sous forme d'une facette.
- Ajout des fonctions **cartesian3d()** et **cylindrical_surface()** qui calculent et renvoient des surfaces avec la possibilité d'ajouter ou non des cloisons séparatrices pour la méthode **g:Dscene3d()**.
- Ajout de la fonction **evalf(f, ...)** qui permet une évaluation protégée de $f(\ldots)$, elle renvoie le résultat de l'évaluation s'il n'y a pas d'erreur d'exécution de la part de Lua, sinon, elle renvoie `nil` mais sans provoquer la fin de l'exécution du script.
- Ajout de la fonction **split_points_by_visibility()** (3D) pour séparer une courbe en deux parties : partie visible, partie cachée.
- Dans les méthodes **g:Dfacet()**, **g:Dmixfacet()**, **g:Dpoly()**, **g:Dedges()**, **g:addFacet()**, **g:addPolyline()**, **g:addPoly()**, les valeurs par défaut des options de tracé de lignes (épaisseur, couleur et style), sont les valeurs courantes en cours.
- Correction de bug...

16) Version 2.1

Liste non exhaustive :

- Par défaut, les fichiers TikZ sont sauvegardés dans un sous-dossier appelé *_luadraw*. La nouvelle option de package `cachedir` permet d'en changer.
- L'option `line join=round` est automatiquement ajoutée à l'environnement *tikzpicture*.
- Deux options supplémentaires pour l'environnement *luadraw* : `bbox` et `pictureoptions`.
- Un certain nombre de fonctions de constructions géométriques supplémentaires en 2D et 3D.
- Les axes gradués (2D, 3D) utilisent le package *siunitx* pour formater les labels lorsque la variable globale `siunitx` a la valeur `true`.
- Ajout des cônes tronqués droits ou penchés (**frustum()** et **g:Dfrustum()**).
- Ajout des pyramides régulières (**regular_pyramid()** et pyramides tronquées **truncated_pyramid()**).
- Les cylindres et les cônes ne sont plus forcément droits, ils peuvent désormais être penchés.
- Ajout de la fonction **cutpolyline(L, D, close)**.
- Dessin (élémentaire) d'ensembles (fonction **set()**) et opérations sur les ensembles (**cap()**, **cup()**, **setminus()**).
- Modification de l'option `mode` de la méthode **g:Dplane()**.
- Ajout de l'option `close` pour la méthode **g:addPolyline()**.
- Correction de bug...

17) Version 2.0

- Introduction du module *luadraw_graph3d.lua* pour les dessins en 3D.
- Introduction de l'option `dir` pour la méthode **g:Dlabel()**.
- Menus changements dans la gestion des couleurs.

18) Version 1.0

Première version.