

# **Le paquet *luadraw***

## **version 2.6**

### **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.

Patrick Fradin

17 février 2026

# Table des matières

<b>I</b>	<b>Dessin 2d</b>	<b>8</b>			
I	Introduction	8			
1)	Prérequis	8			
2)	Options de l'environnement	9			
3)	La classe cpx (complexes)	9			
4)	Afficher une variable dans le terminal	10			
5)	Création d'un graphe	10			
6)	Peut-on utiliser directement du tikz dans l'environnement <i>luadraw</i> ?	12			
II	Méthodes graphiques	12			
1)	Lignes polygonales	12			
	Dpolyline	12			
	Options de dessin d'une ligne polygonale	12			
2)	Segments et droites	14			
	Dangle	14			
	Dbissec	14			
	Dhline	15			
	Dline	15			
	DlineEq	15			
	Dmarkarc	15			
	Dmarkseg	15			
	Dmed	15			
	Dparallel	15			
	Dperp	15			
	Dseg	16			
	Dtangent	16			
	DtangentC	16			
	DtangentI	16			
	Dtangent_from	16			
	Dnormal	17			
	DnormalC	17			
	DnormalI	17			
3)	Figures géométriques	18			
	Darc	18			
	Dcircle	19			
	Dellipse	19			
	Dellipticarc	19			
	Dparallelogram	19			
	Dpolyreg	19			
	Drectangle	19			
	Dsequence	20			
	Dsquare	21			
	Dwedge	21			
4)	Courbes	21			
	Paramétriques : Dparametric	21			
	Polaires : Dpolar	21			
	Cartésiennes : Dcartesian	22			
	Fonctions périodiques : Dperiodic	22			
	Fonctions en escaliers : Dstepfunction	22			
	Fonctions affines par morceaux :				
	Daffinebypiece	23			
	Équations différentielles : Dodesolve	23			
	Courbes implicites : Dimplicit	25			
	Courbes de niveau : Dcontour	25			
	Paramétrisation d'une ligne polygonale : <i>curvilinear_param</i>	26			
5)	Domaines liés à des courbes cartésiennes	27			
	Ddomain1	27			
	Ddomain2	27			
	Ddomain3	27			
6)	Points (Ddots) et labels (Dlabel)	28			
7)	Chemins : Dpath, Dspline et Dtcurve	30			
	Qu'est ce qu'un chemin	30			
	Dessiner un chemin	31			
8)	Chemins et clipping : Beginclip() et Endclip()	33			
9)	Axes et grilles	34			
	Daxes	34			
	DaxeX et DaxeY	37			
	Dgradline	38			
	Dgrid	39			
	Dgradbox	40			
10)	Dessins d'ensembles (diagrammes de Venn)	41			
	Dessiner un ensemble	41			
	Opérations sur les ensembles	42			
11)	Importer une image	43			
	Placer une image dans le graphique	43			
	Mapper une image sur un parallélogramme	44			
12)	Les couleurs	45			
	Calculs sur les couleurs	45			
III	Constructions géométriques	47			
1)	circumcircle(), incircle()	47			
2)	cvx_hull2d()	47			
3)	delaunay()	47			
4)	voronoi()	48			
5)	line2strip()	49			

6)	parallel_polyline()	50	2)	Matrice associée au graphe	61
7)	sss_triangle()	50		g:Composematrix()	61
8)	sas_triangle()	50		g:Det2d()	61
9)	asa_triangle()	50		g:IDmatrix()	61
IV	Calculs sur les listes	51		g:Mtransform()	61
1)	concat	51		g:MLtransform()	61
2)	cut	51		g:Rotate()	62
3)	cutpolyline	51		g:Scale()	62
4)	getbounds	52		g:Savematrix() et g:Restorematrix()	62
5)	getdot	53		g:Setmatrix()	62
6)	insert	53		g:Shift()	62
7)	interCC	53	3)	Changement de vue. Changement de repère	63
8)	interD	54	VII	Ajouter ses propres méthodes à la classe graph	64
9)	interDC	54	1)	Un exemple	65
10)	interDL	54	2)	Comment importer le fichier	66
11)	interL	54	3)	Modifier une méthode existante	67
12)	interP	54	2	Dessin 3d	69
13)	isobar	54	I	Introduction	69
14)	linspace	54	1)	Prérequis	69
15)	map	55	2)	Quelques rappels	69
16)	merge	55	3)	Création d'un graphe 3d	70
17)	polyline2path	55	4)	Modes de projection affine	71
18)	range	55	5)	Projection centrale	72
19)	read_csv_file	55	II	La classe pt3d	73
20)	Fonctions de clipping	55	1)	Représentation des points et vecteurs	73
21)	Ajout de fonctions mathématiques	55	2)	Opérations sur les points 3d	73
	Évaluation protégée : evalf	56	3)	Méthodes de la classe <i>pt3d</i>	73
	int	56	4)	Fonctions mathématiques	73
	gcd	56	5)	Afficher une variable dans le terminal	74
	lcm	56	III	Méthodes graphiques élémentaires	74
	solve	56	1)	Dessin aux traits	74
V	Transformations	58		Ligne polygonale : Dpolyline3d	74
1)	affin	58		Angle droit : Dangle3d	74
2)	ftransform	58		Segment : Dseg3d	74
3)	hom	58		Droite : Dline3d	75
4)	inv	58		Arc de cercle : Darc3d	75
5)	proj	58		Cercle : Dcircle3d	75
6)	projO	59		Chemin 3d : Dpath3d	75
7)	rotate	59		Plan : Dplane	76
8)	shift	59		Courbe paramétrique : Dparametric3d	77
9)	simil	59		Paramétrisation d'une ligne polygonale : <i>curvilinear_param3d</i>	78
10)	sym	59		Le repère : Dboxaxes3d	78
11)	symG	59	2)	Points et labels	79
12)	symO	59		Points 3d : Ddots3d, Dballdots3d, Dcrossdots3d	79
VI	Calcul matriciel	60		Labels 3d : Dlabel3d	79
1)	Calculs sur les matrices	60	3)	Solides de base (sans facette)	80
	applymatrix et applyLmatrix	60		Cylindre : Dcylinder	80
	composematrix	60			
	invmatrix	60			
	matrixof	60			
	mtransform et mLtransform	61			

	Cône : Dcone . . . . .	80		Ajouter des cloisons séparatrices : g:addWall . . . . .	108
	Tronc de cône : Dfrustum . . . . .	81			
	Sphère : Dsphere . . . . .	82	VI	Constructions géométriques . . . . .	111
IV	Solides à facettes . . . . .	83	1)	Cercle circonscrit, cercle inscrit : cir- cumcircle3d(), incircle3d() . . . . .	111
1)	Définition d'un solide . . . . .	83	2)	Enveloppe convexe : cvx_hull3d() . . . . .	111
2)	Dessin d'un polyèdre : Dpoly . . . . .	83	3)	Plans : plane(), planeEq(), ortho- frame(), plane2ABC() . . . . .	112
3)	Visualiser les numéros des facettes et/ou ceux des sommets d'un polyèdre . . . . .	85	4)	Sphère circonscrite, Sphère inscrite : circumsphere(), insphere() . . . . .	113
4)	Fonctions de construction de poly- èdres . . . . .	85	5)	Tétraèdre à longueurs fixées : te- tra_len() . . . . .	113
5)	Lecture dans un fichier obj . . . . .	89	6)	Triangles : sss_triangle3d(), sas_tri- angle3d(), asa_triangle3d() . . . . .	114
6)	Dessin d'une liste de facettes : Dfa- cet et Dmixfacet . . . . .	90	VII	Transformations calcul matriciel et quelques fonctions mathématiques . . . . .	114
7)	Fonctions de construction de listes de facettes . . . . .	92	1)	Transformations 3d . . . . .	114
	surface() . . . . .	92		Appliquer une fonction de transfor- mation : ftransform3d . . . . .	115
	cartesian3d() . . . . .	92		Projections : proj3d, proj3dO, dproj3d . . . . .	115
	cylindrical_surface() . . . . .	93		Projections sur les axes ou les plans liés aux axes . . . . .	115
	curve2cone() . . . . .	94		Symétries : sym3d, sym3dO, dsym3d, psym3d . . . . .	115
	curve2cylinder() . . . . .	94		Rotation : rotate3d, rotateaxe3d . . . . .	115
	line2tube(); section2tube() . . . . .	95		Homothétie : scale3d . . . . .	115
	rotcurve() . . . . .	96		Inversion : inv3d . . . . .	115
	rotline() . . . . .	97		Stéréographie : projstereo et inv_projstereo . . . . .	115
8)	Arêtes d'un solide . . . . .	98		Translation : shift3d . . . . .	116
9)	Méthodes et fonctions s'appliquant à des facettes ou polyèdres . . . . .	99	2)	Calcul matriciel . . . . .	116
10)	Découper un solide : cutpoly et cut- facet . . . . .	100		applymatrix3d et applyLmatrix3d . . . . .	116
11)	Clipper des facettes avec un poly- èdre convexe : clip3d . . . . .	101		composematrix3d . . . . .	116
12)	Clipper un plan avec un polyèdre convexe : clipplane . . . . .	102		invmatrix3d . . . . .	116
V	La méthode Dscene3d . . . . .	102		matrix3dof . . . . .	116
1)	Le principe, les limites . . . . .	102		mtransform3d et mLtransform3d . . . . .	116
2)	Construction d'une scène 3d . . . . .	103	3)	Matrice associée au graphe 3d . . . . .	116
3)	Méthodes pour ajouter un objet dans la scène 3d . . . . .	104		g:Composematrix3d() . . . . .	117
	Ajouter des facettes : g:addFacet et g:addPoly . . . . .	104		g:Det3d() . . . . .	117
	Ajouter un plan : g:addPlane et g:addPlaneEq . . . . .	104		g:IDmatrix3d() . . . . .	117
	Ajouter une ligne polygonale : g:add- Polyline . . . . .	104		g:Mtransform3d() . . . . .	117
	Ajouter des axes : g:addAxes . . . . .	105		g:MLtransform3d() . . . . .	117
	Ajouter une droite : g:addLine . . . . .	105		g:Rotate3d() . . . . .	117
	Ajouter un angle "droit" : g:addAngle . . . . .	105		g:Scale3d() . . . . .	117
	Ajouter un arc de cercle : g:addArc . . . . .	105		g:Setmatrix3d() . . . . .	117
	Ajouter un cercle : g:addCircle . . . . .	105		g:Shift3d() . . . . .	117
	Ajouter des points : g:addDots . . . . .	107	4)	Fonctions mathématiques supplé- mentaires . . . . .	117
	Ajouter des labels : g:addLabels . . . . .	107		clippolyline3d() . . . . .	117
				clipline3d() . . . . .	118
				cutpolyline3d() . . . . .	118

getbounds3d()	118	Ajouter un label : g:DSlabel	134
interDP()	118	Ajouter des points : g:DSdots et g:DSstars	134
interPP()	118	Stéréographie inverse : g:DSinvstereo_curve et g:DSinvstereo_polyline	134
interDD()	118	Exemples	134
interCS()	118	3) Le module <i>luadraw_palettes</i>	139
interDS()	118	4) Le module <i>luadraw_compile_tex</i>	139
interPS()	119	Première partie : compilation et lecture	139
interSS()	119	Deuxième partie : exploitation du résultat	140
interSSS()	119	5) Le module <i>luadraw_cvx_polyhedra_nets</i>	142
merge3d()	119	La fonction de base	142
split_points_by_visibility()	119	La méthode de dessin	143
VIII Exemples plus poussés	120	Exemples	143
1) La boîte de sucre	120	La fonction <i>unfold_tree()</i>	145
2) Empilement de cubes	122	6) Le module <i>luadraw_fields</i>	146
3) Illustration du théorème de Dandelin	123	7) Le module <i>luadraw_shadedforms</i>	147
4) Volume défini par une intégrale double	125	Dshadedpolyline	147
5) Volume défini sur autre chose qu'un pavé	126	Dcolorbar	148
<b>3 Annexes</b>	<b>128</b>	Dshadedrectangle	149
I Extensions	128	Dshadedregion	150
1) Le module <i>luadraw_polyhedrons</i>	128	8) Le module <i>luadraw_povray</i>	151
2) Le module <i>luadraw_spherical</i>	130	Avant la création des objets	151
Variables et fonctions globales du module	130	Création des objets	151
Définition de la sphère	131	Liste des objets prédéfinis	152
Ajouter un cercle : g:DScircle	131	Sauvegarde, exécution, inclusion	154
Ajouter un grand cercle : g:DSbigcircle	131	Exemples	154
Ajouter un arc de grand cercle : g:DSarc	132	II Historique	157
Ajouter un angle : g:DSangle	132	1) Version 2.6	157
Ajouter une facette sphérique : g:DSfacet	132	2) Version 2.5	157
Ajouter une courbe sphérique : g:DScurve	132	3) Version 2.4	158
Ajouter un segment : g:DSseg	133	4) Version 2.3	158
Ajouter une droite : g:DSline	133	5) Version 2.2	158
Ajouter une ligne polygonale : g:DSpolyline	133	6) Version 2.1	159
Ajouter un plan : g:DSplane	133	7) Version 2.0	159
		8) Version 1.0	159

# Table des figures

1	Un premier exemple : trois sous-figures dans un même graphique	8
2	Champ de vecteurs, courbe intégrale de $y' = 1 - xy^2$	14
3	Tangentes issues d'un point	17
4	Symétrique de l'orthocentre	18
5	Suite $u_{n+1} = \cos(u_n)$	20
6	Un système différentiel de Lokta-Volterra	24
7	Exemple avec Dcontour	26
8	Points répartis sur une ligne polygonale	27
9	Partie entière, fonctions Ddomain1 et Ddomain3	28
10	Path example	30
11	Path et Spline	32
12	Courbe d'interpolation avec vecteurs tangents imposés	33
13	Exemple de clipping	34
14	Exemple avec axes avec grille	36
15	Exemples de droites graduées	39
16	Exemple de repère non orthogonal	40
17	Utilisation de Dgradbox	41
18	Dessiner un ensemble	42
19	Opérations sur les ensembles	43
20	Importer une image	44
21	Mapper une image	45
22	Les cinq palettes par défaut	46
23	Triangulation de Delaunay	48
24	Diagramme de Voronoï	49
25	Exemple avec <i>line2strip</i>	50
26	sss_triangle, sas_triangle et asa_triangle	51
27	Illustrer un exercice de programmation linéaire	52
28	Tangentes à un cercle {O,2} et à une ellipse {O,3,2} issues d'un point	54
29	Fonction $f$ définie par $\int_x^{f(x)} \exp(t^2)dt = 1$ .	58
30	Utilisation de transformations	60
31	Utilisation de la matrice du graphe	62
32	Utilisation de Shift, Rotate et Scale	63
33	Classification des points d'une courbe paramétrée	64
34	Utilisation des nouvelles méthodes	67
35	Modification d'une méthode existante	68
1	Point col en $M(0, 0, 0)$ ( $z = x^2 - y^2$ )	69
2	Angles de vue	71
3	Modes de projection affine	71

4	Projection centrale . . . . .	72
5	Dplane, exemple avec mode = left+bottom . . . . .	77
6	Une courbe et ses projections sur trois plans . . . . .	78
7	Un tétraèdre et les centres de gravité de chaque face . . . . .	79
8	Cylindres, cônes et sphères . . . . .	83
9	Section d'un tétraèdre par un plan . . . . .	85
10	Visualiser les faces et sommets d'un polyèdre . . . . .	85
11	Cône tronqué, pyramide tronquée, cylindre oblique . . . . .	87
12	Hyperbole : intersection cône - plan . . . . .	88
13	Section de cône avec plusieurs vues . . . . .	89
14	Masque de Nefertiti . . . . .	90
15	Exemple de courbes de niveaux sur une surface . . . . .	92
16	Surfaces utilisant l'option <i>addWall</i> . . . . .	93
17	Exemple de cône elliptique . . . . .	94
18	Section d'un cylindre non circulaire . . . . .	95
19	Exemple avec <i>line2tube</i> et <i>section2tube</i> . . . . .	96
20	Exemple avec <i>rotcurve</i> . . . . .	97
21	Exemple avec <i>rotline</i> . . . . .	98
22	Sphère inscrite dans un octaèdre avec projection du centre sur les faces . . . . .	100
23	Cube coupé par un plan ( <i>cutpoly</i> ), avec <i>close=false</i> et avec <i>close=true</i> . . . . .	101
24	Exemple avec <i>clip3d</i> : construction d'un dé à partir d'un cube et d'une sphère . . . . .	102
25	Premier exemple avec <i>Dscene3d</i> : intersection de deux plans . . . . .	103
26	Cylindre plein plongé dans de l'eau . . . . .	106
27	Construction d'un icosaèdre . . . . .	108
28	Exemple avec <i>addWall</i> (les deux facettes transparentes roses sont normalement invisibles) . . . . .	109
29	Tore et lemniscate . . . . .	110
30	Section de sphère sans <i>Dscene3d()</i> . . . . .	111
31	Utilisation de <i>cvx_hull3d()</i> . . . . .	112
32	Faces d'un cube trouées avec un hexagone régulier . . . . .	113
33	Un tétraèdre avec la longueur des arêtes fixée . . . . .	114
34	Une courbe sur un cylindre . . . . .	120
35	Boîte de morceaux de sucre . . . . .	122
36	Empilement de cubes . . . . .	123
37	Illustration du théorème de Dandelin . . . . .	125
38	Volume correspondant à $\int_{x_1}^{x_2} \int_{y_1}^{y_2} f(x,y) dx dy$ . . . . .	126
39	Volume : $0 \leq x \leq 1$ ; $0 \leq y \leq x^2$ ; $0 \leq z \leq y^2$ . . . . .	127
1	Polyèdres du module <i>luadraw_polyhedrons</i> . . . . .	130
2	Cube dans une sphère . . . . .	135
3	Fenêtre de Viviani . . . . .	136
4	Un pavage sphérique . . . . .	137
5	Tangentes à la sphère issues d'un point . . . . .	138
6	Méthodes <i>DSinustereo_curve</i> et <i>DSinustereo_polyline</i> . . . . .	138
7	Exemple avec <i>compile_tex</i> en 2d . . . . .	140
8	Écrire sur un cylindre . . . . .	142
9	Patron d'un parallélépipède . . . . .	144
10	Patron imposé d'un parallélépipède . . . . .	144
11	Parallélépipède tronqué à demi déplié . . . . .	145
12	Dépliage d'un dodécaèdre . . . . .	146
13	Shaded polyline . . . . .	148
14	Color bars . . . . .	149
15	Shaded rectangle . . . . .	150

16	Shaded region . . . . .	150
17	Intersection de deux surfaces implicites . . . . .	155
18	Cercles de Villarceau . . . . .	156
19	Trous dans une demi-sphère . . . . .	157



# Dessin 2d

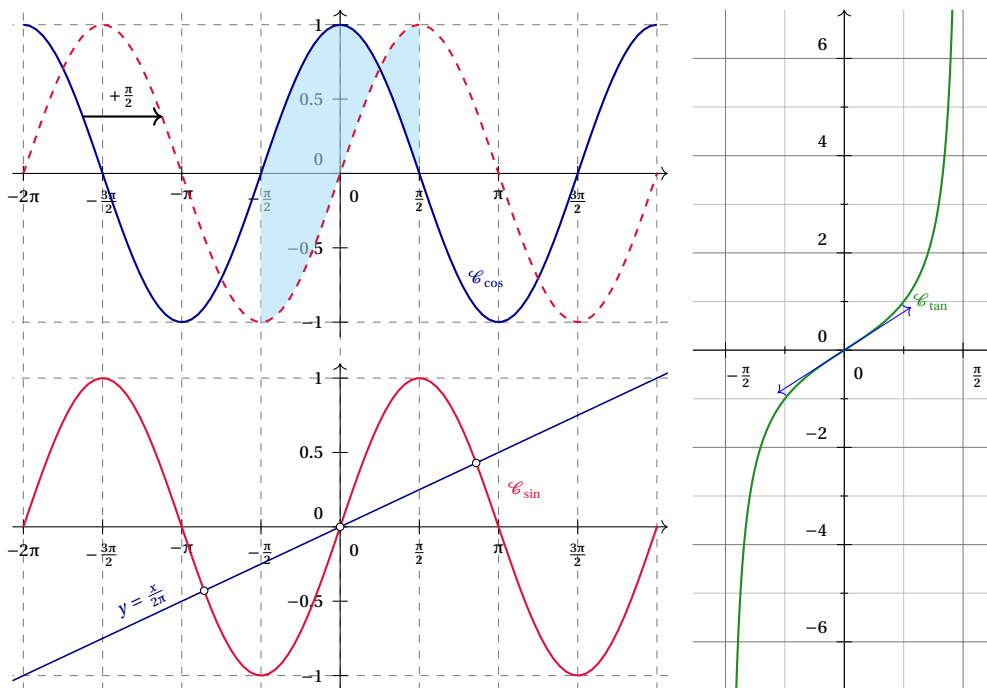


FIGURE 1 : Un premier exemple : trois sous-figures dans un même graphique

## I Introduction

### 1) 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 *graph*, et fournit l'environnement *luadraw* qui permet de faire des graphiques en Lua.

**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 *graph3d* (qui s'appuie sur la classe *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*
- *plotmarks*
- *arrows.meta*
- *decorations.markings*

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 }
-- création d'un nouveau graphique en lui donnant un nom local
local g = graph:new{ 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}
```

**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=*.

## 2) Options de l'environnement

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*.

## 3) La classe cpx (complexes)

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 **Z(a,b)** pour  $a + i \times b$ , ou bien avec la fonction **Zp(r,theta)** pour  $r \times e^{i\theta}$  en coordonnées polaires.

- Exemple : *local z = 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*.
- **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 =. Lorsqu'un calcul échoue le résultat renvoyé en principe doit être égal à *nil*.

- À cela s'ajoutent les fonctions suivantes (il faut utiliser la notation pointée en Lua) :
  - 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'exponentielle complexe : **cpx.exp(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 une variable *epsilon* qui vaut  $1e-16$  par défaut.

La dernière fonction renvoie un booléen, les fonctions *bar*, exponentielle et *round* renvoient un complexe, et les autres renvoient un réel.

On dispose également de la constante *cpx.I* qui représente l'imaginaire pur *i*.

Exemple :

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

Le symbole de multiplication est obligatoire.

#### 4) Afficher une variable dans le terminal

L'instruction **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*. L'argument *msg* est une chaîne optionnelle (vide par défaut) qui est affichée avec le type pour repérer la variable dans le terminal.

#### 5) 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 :

```
1 local g = graph:new{ window={x1,x2,y1,y2,xscale,yscale}, margin={left,right,top,bottom},
2 size={largeur,hauteur,ratio}, bg="color", border=true/false, bbox=true/false, pictureoptions="" }
```

La classe *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).

- Le paramètre (facultatif) *window* définit le pavé de  $\mathbf{R}^2$  correspondant au graphique : c'est  $[x_1, x_2] \times [y_1, y_2]$ . Les paramètres *xscale* et *yscale* sont facultatifs et valent 1 par défaut, ils représentent l'échelle (cm par unité) sur les axes. Par défaut on a *window* =  $\{-5, 5, -5, 5, 1, 1\}$ .
- Le paramètre (facultatif) *margin* définit des marges autour du graphique en cm. Par défaut on a *margin* =  $\{0.5, 0.5, 0.5, 0.5\}$ .
- Le paramètre (facultatif) *size* permet d'imposer une taille (en cm, marges incluses) pour le graphique, l'argument *ratio* 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. L'utilisation de ce paramètre va modifier les valeurs de *xscale* et *yscale* pour avoir les bonnes tailles. Par défaut la taille est de  $11 \times 11$  (en cm) avec les marges ( $10 \times 10$  sans les marges).
- Le paramètre (facultatif) *bg* 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.

- Le paramètre (facultatif) *border* indique si un cadre doit être dessiné ou non autour du graphique (en incluant les marges). Par défaut ce paramètre vaut *false*.
- Le paramètre (facultatif) *bbox* 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.
- Le paramètre (facultatif) *pictureoptions* est une chaîne de caractères destinée à contenir des options qui seront passées à *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  $\text{\TeX}$ , 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  $\text{\TeX}$  tout en les sauvegardant dans un fichier texte<sup>1</sup>. La méthode ***g:Save()*** enregistre le graphique dans le fichier désigné par l'option (de l'environnement) *name* mais sans envoyer les instructions à  $\text{\TeX}$ .
- 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, 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.

### 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 la pavé  $[x1, x2] \times [y1, y2]$ .
  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.

1. Ce fichier contiendra un environnement *tikzpicture*.

## 6) 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 :

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

Ou encore pour changer des styles :

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

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

```
1 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 :

```
1 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* 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. L'argument *clip* doit contenir ou bien *nil* (valeur par défaut) ou bien une table de la forme  $\{x_1, x_2, y_1, y_2\}$ , 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 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 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* est 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.

On peut modifier certaines options individuellement avec les méthodes :

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

```

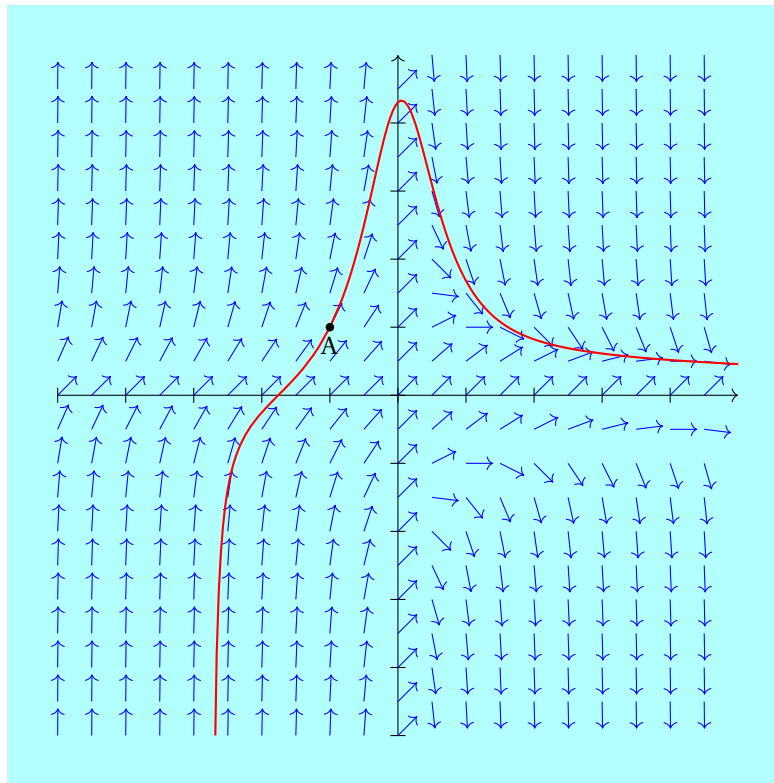
1 \begin{luadraw}{name=champ}
2 local g = graph:new{window={-5,5,-5,5},bg="Cyan!30",size={10,10}}
3 local f = function(x,y) -- éq. diff. y' = 1-x*y^2=f(x,y)
4     return 1-x*y^2
5 end
6 local A = Z(-1,1) -- A = -1+i
7 local deltaX, deltaY, long = 0.5, 0.5, 0.4
8 local champ = function(f)
9     local vecteurs, v = {}
10    for y = g:Yinf(), g:Ysup(), deltaY do
11        for x = g:Xinf(), g:Xsup(), deltaX do
12            v = Z(1,f(x,y)) -- coordonnées 1 et f(x,y)
13            v = v/cpx.abs(v)*long -- normalisation de v

```

```

14     table.insert(vecteurs, {Z(x,y), Z(x,y)+v} )
15     end
16 end
17 return vecteurs -- vecteurs est une ligne polygonale
18 end
19 g:Daxes( {0,1,1}, {labelpos={"none","none"}, arrows="->"} )
20 g:Dpolyline( champ(f), "->,blue")
21 g:Dodesolve(f, A.re, A.im, {t={-2.75,5},draw_options="red,line width=0.8pt"})
22 g:Dlabeldot("$A$", A, {pos="S"})
23 g:Show()
24 \end{luadraw}

```

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

## 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 **angleD(B,A,C,r)** renvoie la liste des points de cet angle.

### Dbissec

- La méthode **g:Dbissec(B,A,C,interior,draw\_options)** 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`.
- La fonction **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)** dessine une demi-droite, l'argument  $d$  doit être une liste de complexes  $\{A,B\}$ , c'est la demi-droite  $[A,B)$  qui est dessinée.

Variante : **g:Dhline(A,B,draw\_options)**. L'argument  $draw\_options$  est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction  $\backslash draw$ .

**Dline**

La méthode **g:Dline(d,draw\_options)** trace la droite  $d$ , celle-ci est une liste du type  $\{A,u\}$  où  $A$  représente un point de la droite (un complexe) et  $u$  un vecteur directeur (un complexe non nul).

Variante : la méthode **g:Dline(A,B,draw\_options)** trace la droite passant par les points  $A$  et  $B$  (deux complexes). L'argument  $draw\_options$  est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction  $\backslash draw$ .

**DlineEq**

- La méthode **g:DlineEq(a,b,c,draw\_options)** 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  $\backslash draw$ .
- La fonction **lineEq(a,b,c)** 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.

**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  $\backslash draw$ .

**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 ( $45^\circ$  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  $\backslash draw$ .

**Dmed**

- La méthode **g:Dmed(A,B,draw\_options)** trace la médiatrice du segment  $[A;B]$ .  
Variante : **g:Dmed(seg,draw\_options)** où  $seg$  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  $\backslash draw$ .
- La fonction **med(A,B)** (ou **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)** 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  $\backslash draw$ .
- La fonction **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)** 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  $\backslash draw$ .



- La fonction **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 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`.

## 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  $t_0$ . Si l'argument *long* vaut *nil* (valeur par défaut) alors c'est la droite entière qui est dessinée, sinon c'est un segment de longueur *long*. L'argument *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction `\draw`.
- La fonction **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  $x_0$ . Si l'argument *long* vaut *nil* (valeur par défaut) alors c'est la droite entière qui est dessinée, sinon c'est un segment de longueur *long*. L'argument *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction `\draw`.
- La fonction **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  $(x_0, y_0)$  supposé sur la courbe. Si l'argument *long* vaut *nil* (valeur par défaut) alors c'est la droite entière qui est dessinée, sinon c'est un segment de longueur *long*. L'argument *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction `\draw`.
- La fonction **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)** dessine la ou les tangentes à la courbe paramétrée par  $p : t \mapsto p(t)$  (à valeurs complexes) issue(s) du point  $A$  (nombre complexe). Les arguments  $t_1$  et  $t_2$  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  $A$ .
- La fonction **tangent\_from(A,p,t1,t2,dp)** renvoie la liste des 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.

```

1 \begin{luadraw}{name=tangent_from}
2 local i, cos, sin, pi = cpx.I, math.cos, math.sin, math.pi
3 local g = graph:new{window={-5,5,-5,5}, size={10,10}}
4 local p1 = function(t) return t+i*(t^2/4-1) end -- parabola
5 local p2 = function(t) return 1/2+i*rotate(2*cos(t)+i*sin(t),15) end -- ellipse
6 local p3 = function(t) return math.sinh(t)+i*math.cosh(t)-i*2 end -- branch of a hyperbola
7 local p4 = function(t) return 2*cos(3*t)*cpx.exp(i*t) end -- other
8 local P = 0.5-1.25*i
9 local draw = function(p,t1,t2)
10     local axis_style = { arrows='-Stealth', legend={'$x$', '$y$'} }
11     local S = {}

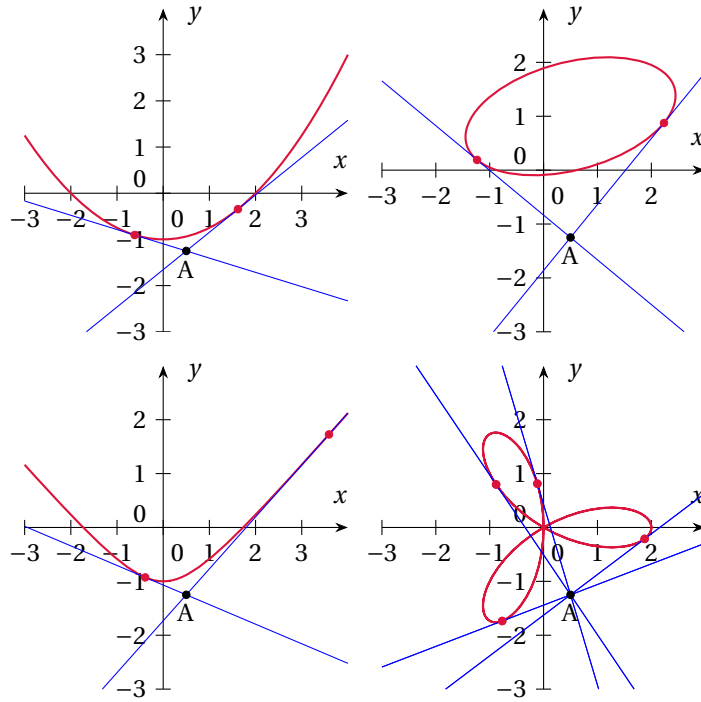
```

```

12 g:Daxes({0, 1, 1}, axis_style)
13 g:Dparametric(p,{t={t1,t2}, draw_options="line width=0.8pt, Crimson"})
14 g:Dtangent_from(P,p,t1,t2,"blue",S)
15 g:Ddots(S,"Crimson"); g:Ddots(P); g:Dlabel("$A$",P,{pos="S"})
16 end
17 g:Saveattr(); g:Viewport(-5,-0.25,0.25,5); g:Coordsystem(-3,4,-3,4); draw(p1,-4,4); g:Restoreattr()
18 g:Saveattr(); g:Viewport(0.25,5,0.25,5); g:Coordsystem(-3,3,-3,3); draw(p2,-pi,pi); g:Restoreattr()
19 g:Saveattr(); g:Viewport(-5,-0.25,-5,-0.25); g:Coordsystem(-3,4,-3,3); draw(p3,-4,4); g:Restoreattr()
20 g:Saveattr(); g:Viewport(0.25,5,-5,-0.25); g:Coordsystem(-3,3,-3,3); draw(p4,-pi,pi); g:Restoreattr()
21 g:Show()
22 \end{luadraw}

```

FIGURE 3 : Tangentes issues d'un point



### Dnormal

- La méthode **g:Dnormal(p,t0,long,draw\_options)** dessine la normale 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, sinon c'est un segment de longueur *long*. L'argument *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction `\draw`.
- La fonction **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  $x0$ . Si l'argument *long* vaut *nil* (valeur par défaut) alors c'est la droite entière qui est dessinée, sinon c'est un segment de longueur *long*. L'argument *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction `\draw`.
- La fonction **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, sinon c'est un segment de longueur *long*. L'argument *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction `\draw`.

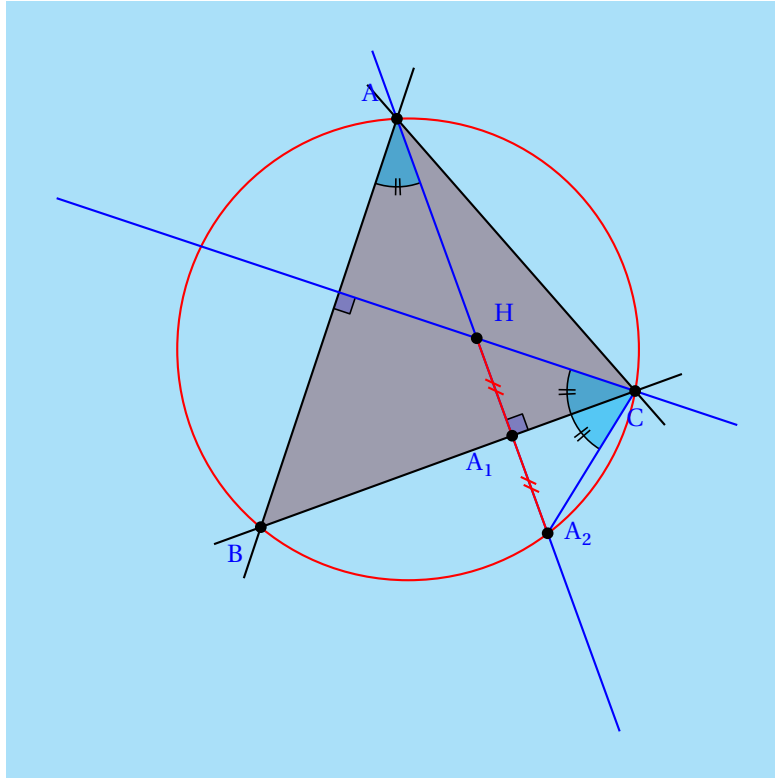
- La fonction **normalI(f,x0,y0,long)** renvoie soit la droite, soit un segment (suivant que *long* vaut *nil* ou pas).

```

1 \begin{luadraw}{name=orthocentre}
2 local g = graph:new{window={-5,5,-5,5},bg="cyan!30",size={10,10}}
3 local i = cpx.I
4 local A, B, C = 4*i, -2-2*i, 3.5
5 local h1, h2 = perp({B,C-B},A), perp({A,B-A},C) -- hauteurs
6 local A1, F = proj(A,{B,C-B}), proj(C,{A,B-A}) -- projetés
7 local H = interD(h1,h2) -- orthocentre
8 local A2 = 2*A1-H -- symétrique de H par rapport à BC
9 g:Dpolyline({A,B,C},true, "draw=none,fill=Maroon,fill opacity=0.3") -- fond du triangle
10 g:Linewidth(6); g:Filloptions("full", "blue", 0.2)
11 g:Dangle(C,A1,A,0.25); g:Dangle(B,F,C,0.25) -- angles droits
12 g:Linecolor("black"); g:Filloptions("full","cyan",0.5)
13 g:Darc(H,C,A2,1); g:Darc(B,A,A1,1) -- arcs
14 g:Filloptions("none","black",1) -- on rétablit l'opacité à 1
15 g:Dmarkarc(H,C,A1,1,2); g:Dmarkarc(A1,C,A2,1,2) -- marques
16 g:Dmarkarc(B,A,H,1,2)
17 g:Linewidth(8); g:Linecolor("black")
18 g:Dseg({A,B},1.25); g:Dseg({C,B},1.25); g:Dseg({A,C},1.25) -- côtés
19 g:Linecolor("red"); g:Dcircle(A,B,C) -- cercle
20 g:Linecolor("blue"); g:Dline(h1); g:Dline(h2) -- hauteurs
21 g:Dseg({A2,C}); g:Linecolor("red"); g:Dseg({H,A2}) -- segments
22 g:Dmarkseg(H,A1,2); g:Dmarkseg(A1,A2,2) -- marques
23 g:Labelcolor("blue") -- pour les labels
24 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"}}
25 g:Linecolor("black"); g:Filloptions("full"); g:Ddots({A,B,C,H,A1,A2}) -- dessin des points
26 g:Show(true)
27 \end{luadraw}

```

FIGURE 4 : 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, 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 **arc(B,A,C,r,sens)** renvoie la liste des points de cet arc (ligne polygonale).
- La fonction **arcb(B,A,C,r,sens)** renvoie cet arc sous forme d'un chemin (voir Dpath) utilisant des courbes de Bézier.

### Dcircle

- La méthode **g:Dcircle(c,r,d,draw\_options)** trace un cercle. Lorsque l'argument *d* est 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 **circle(c,r,d)** renvoie la liste des points de ce cercle (ligne polygonale).
- La fonction **circleb(c,r,d)** renvoie ce cercle sous forme d'un chemin (voir Dpath) 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 **ellipse(c,rx,ry,inclin)** renvoie la liste des points de cette ellipse (ligne polygonale).
- La fonction **ellipseb(c,rx,ry,inclin)** renvoie cette ellipse sous forme d'un chemin (voir Dpath) 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 Ox (angle nul par défaut), allant de *B* (complexe) vers *A* (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*.
- La fonction **ellipticarc(B,A,C,rx,ry,sens,inclin)** renvoie la liste des points de cet arc (ligne polygonale).
- La fonction **ellipticarcb(B,A,C,rx,ry,sens,inclin)** renvoie cet arc sous forme d'un chemin (voir Dpath) 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 **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 **polyreg(sommet1,sommet2,nbcotes,sens)** et la fonction **polyreg(centre,sommet,nbcotes)**, renvoient la liste des sommets de ce polygone régulier.

### Drectangle

- La méthode **g:Drectangle(a,b,c,draw\_options)** dessine le rectangle ayant comme sommets consécutif *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 **rectangle(a,b,c)** renvoie la liste des sommets de ce rectangle.

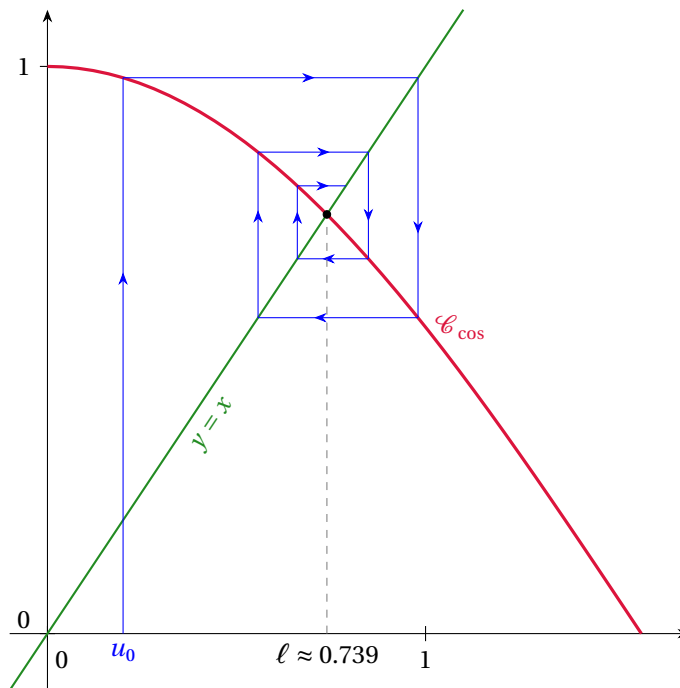
## Dsequence

- La méthode **g:Dsequence(f,u0,n,draw\_options)** fait le dessin des "escaliers" de la suite récurrente définie par son premier terme  $u_0$  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 **sequence(f,u0,n)** renvoie la liste des points constituant ces "escaliers".

```

1 \begin{luadraw}{name=sequence}
2 local g = graph:new{window={-0.1,1.7,-0.1,1.1},size={10,10,0}}
3 local i, pi, cos = cpx.I, math.pi, math.cos
4 local f = function(x) return cos(x)-x end
5 local ell = solve(f,0,pi/2)[1]
6 local L = sequence(cos,0.2,5) -- u_{n+1}=cos(u_n), u_0=0.2
7 local seg, z = {}, L[1]
8 for k = 2, #L do
9     table.insert(seg,{z,L[k]})
10    z = L[k]
11 end -- seg est la liste des segments de l'escalier
12 g:Writeln("\tikzset{->/.style={decoration={markings, mark=at position #1 with {\arrow{Stealth}}},
13   ~ postaction={decorate}}}")
14 g:Daxes({0,1,1}, {arrows="-Stealth"})
15 g:DlineEq(1,-1,0,"line width=0.8pt,ForestGreen")
16 g:Dcartesian(cos, {x={0,pi/2},draw_options="line width=1.2pt,Crimson"})
17 g:Dpolyline(seg,false,"->=0.65,blue")
18 g:Dlabel("$u_0$",0.2,{pos="S",node_options="blue"})
19 g:Dseg({ell, ell*(1+i)},1,"dashed,gray")
20 g:Dlabel("$\ell\approx\text{.round(ell,3)}..\"", ell,{pos="S"})
21 g:Ddots(ell*(1+i)); g:Labelcolor("Crimson")
22 g:Dlabel("$\mathcal{C}_{\cos}$",Z(1,cos(1)),{pos="E"})
23 g:Dlabel("$y=x$",Z(0.4,0.4),{pos="S",dist=0.1})
24 g:Show()
25 \end{luadraw}

```

FIGURE 5 : 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 **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 **parametric(p,t1,t2,nbdots,discont,nbdiv)** fait le calcul des points et renvoie une ligne polygonale (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  $t1 < t2$ , 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,args)** fait le calcul des points et le dessin de la courbe paramétrée par  $p$ . Le paramètre *args* est une table à 6 champs :

```
{ t={t1,t2}, nbdots=40, discont=true/false, nbdiv=5, draw_options="", clip={x1,x2,y1,y2} }
```

- Par défaut, le champ  $t$  est égal à  $\{g:Xinf(),g:Xsup()\}$ ,
- le champ *nbdots* vaut 40,
- le champ *discont* vaut *false*,
- le champ *nbdiv* vaut 5,
- le champ *draw\_options* est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction `\draw`),
- le champ *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.

### Polaires : Dpolar

- La fonction **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,args)** fait le calcul des points et le dessin de la courbe polaire paramétrée par  $rho$ . Le paramètre *args* est une table à 6 champs :

```
{ t={t1,t2}, nbdots=40, discont=true/false, nbdiv=5, draw_options="", clip={x1,x2,y1,y2} }
```

- Par défaut, le champ  $t$  est égal à  $\{-\pi, \pi\}$ ,
- le champ *nbdots* vaut 40,
- le champ *discont* vaut *false*,
- le champ *nbddiv* vaut 5,
- le champ *draw\_options* est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction `\draw`),
- le champ *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.

### Cartésiennes : Dcartesian

- La fonction **cartesian(f,x1,x2,nbdots,discont,nbddiv)** fait le calcul des points 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,args)** fait le calcul des points et le dessin de la courbe de  $f$ . Le paramètre *args* est une table à 6 champs :

```
{ x={x1,x2}, nbdots=40, discont=true/false, nbddiv=5, draw_options="", clip={x1,x2,y1,y2} }
```

- Par défaut, le champ  $x$  est égal à  $\{g:Xinf(),g:Xsup()\}$ ,
- le champ *nbdots* vaut 40,
- le champ *discont* vaut *false*,
- le champ *nbddiv* vaut 5,
- le champ *draw\_options* est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction `\draw`),
- le champ *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.

### Fonctions périodiques : Dperiodic

- La fonction **periodic(f,period,x1,x2,nbdots,discont,nbddiv)** fait le calcul des points 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,args)** fait le calcul des points et le dessin de la courbe de  $f$ . Le paramètre *args* est une table à 6 champs :

```
{ x={x1,x2}, nbdots=40, discont=true/false, nbddiv=5, draw_options="", clip={x1,x2,y1,y2} }
```

- Par défaut, le champ  $x$  est égal à  $\{g:Xinf(),g:Xsup()\}$ ,
- le champ *nbdots* vaut 40,
- le champ *discont* vaut *false*,
- le champ *nbddiv* vaut 5,
- le champ *draw\_options* est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction `\draw`),
- le champ *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.

### Fonctions en escaliers : Dstepfunction

- La fonction **stepfunction(def,discont)** fait le calcul des points 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}, {c1,c2,...} }
```

Le premier élément  $\{x_1, x_2, x_3, \dots, x_n\}$  doit être une subdivision du segment  $[x_1, x_n]$ .

Le deuxième élément  $\{c_1, c_2, \dots\}$  est la liste des constantes avec  $c_1$  pour le morceau  $[x_1, x_2]$ ,  $c_2$  pour le morceau  $[x_2, x_3]$ , etc.

- L'argument *discont* est un booléen qui vaut *true* par défaut.

- La méthode **g:Dstepfunction(def,args)** 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 *args* est une table à 3 champs :

```
{ discont=true/false, draw_options="", clip={x1,x2,y1,y2} }
```

Par défaut, le champ *discont* vaut *true*, et le champ *draw\_options* est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction `\draw`). Le champ *clip* est soit *nil* (valeur par défaut), soit une table  $\{x_1, x_2, y_1, y_2\}$ , 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 **affinebypiece(def,discont)** fait le calcul des points 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  $\{x_1, x_2, x_3, \dots, x_n\}$  doit être une subdivision du segment  $[x_1, x_n]$ .

Le deuxième élément  $\{\{a_1, b_1\}, \{a_2, b_2\}, \dots\}$  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,args)** 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 *args* est une table à 3 champs :

```
{ discont=true/false, draw_options="", clip={x1,x2,y1,y2} }
```

Par défaut, le champ *discont* vaut *true*, et le champ *draw\_options* est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction `\draw`). Le champ *clip* est soit *nil* (valeur par défaut), soit une table  $\{x_1, x_2, y_1, y_2\}$ , 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 **odesolve(f,t0,Y0,tmin,tmax,nbdots,method)** permet une résolution approchée de l'équation différentielle  $Y'(t) = f(t, Y(t))$  dans l'intervalle  $[tmin, tmax]$  qui doit contenir  $t_0$ , avec la condition initiale  $Y(t_0) = Y_0$ .

- L'argument *f* est une fonction  $f : (t, Y) \rightarrow f(t, Y)$  à valeurs dans  $\mathbb{R}^n$  et où  $Y$  est également dans  $\mathbb{R}^n : Y = \{y_1, y_2, \dots, y_n\}$  (lorsque  $n = 1$ ,  $Y$  est un réel).

- Les arguments  $t_0$  et  $Y_0$  donnent les conditions initiales avec  $Y_0 = \{y_1(t_0), \dots, y_n(t_0)\}$  (les  $y_i$  sont réels), ou  $Y_0 = y_1(t_0)$  lorsque  $n = 1$ .

- Les arguments *tmin* et *tmax* définissent l'intervalle de résolution, celui-ci doit contenir  $t_0$ .

- L'argument *nbdots* indique le nombre de points calculés de part et d'autre de  $t_0$ .

- 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.



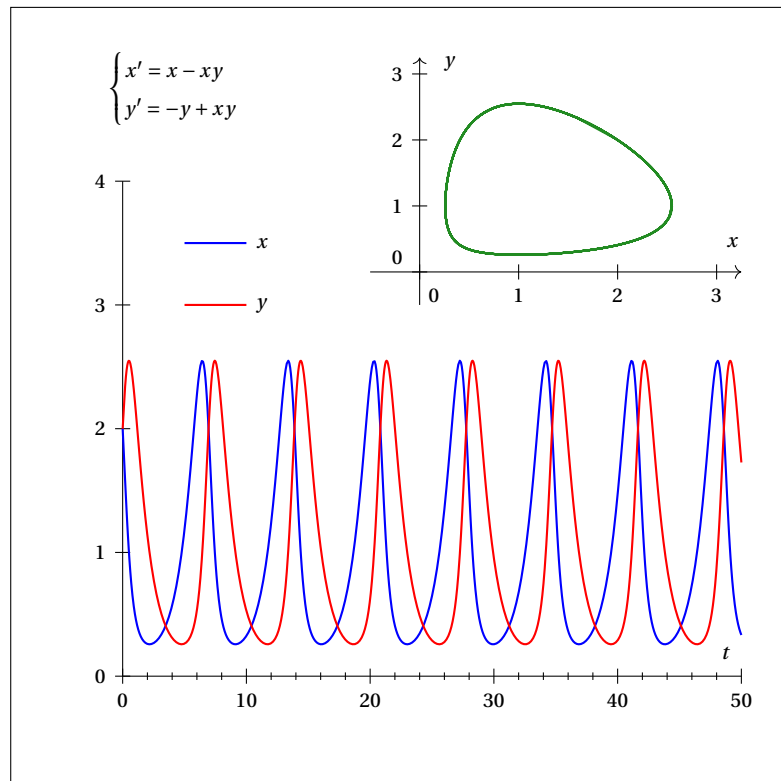
- La méthode **g:DplotXY(X,Y,draw\_options,clip)**, où les arguments  $X$  et  $Y$  sont deux listes de réels de même longueur, dessine la ligne polygonale constituée des points  $(X[k], Y[k])$ . L'argument *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction `\draw`. Le champ *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.

```

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

```

FIGURE 6 : Un système différentiel de Lokta-Volterra



- La méthode **g:Dodesolve(f,t0,Y0,args)** permet le dessin d'une solution à l'équation  $Y'(t) = f(t, Y(t))$ .
  - L'argument obligatoire  $f$  est une fonction  $f : (t, Y) \rightarrow f(t, Y)$  à valeurs dans  $\mathbb{R}^n$  et où  $Y$  est également dans  $\mathbb{R}^n$  :  $Y = \{y_1, y_2, \dots, y_n\}$  (lorsque  $n = 1$ ,  $Y$  est un réel).
  - Les arguments  $t_0$  et  $Y_0$  donnent les conditions initiales avec  $Y_0 = \{y_1(t_0), \dots, y_n(t_0)\}$  (les  $y_i$  sont réels), ou  $Y_0 = y_1(t_0)$  lorsque  $n = 1$ .

- L'argument *args* (facultatif) permet de définir les paramètres pour la courbe, c'est une table à 6 champs :

```
{ t={tmin,tmax}, out={i1,i2}, nbdots=50, method="rkf45"/"rk4", draw_options="", clip={x1,x2,y1,y2} }
```

- \* Le champ *t* détermine l'intervalle pour la variable *t*, par défaut il vaut  $\{g:Xinf(), g:Xsup()\}$ .
- \* Le champ *out* est une table de deux entiers  $\{i1, i2\}$ , si *M* désigne la matrice renvoyée par la fonction *odesolve*, les points dessinés auront pour abscisses les  $M[i1]$  et pour ordonnées les  $M[i2]$ . Par défaut on a  $i1=1$  et  $i2=2$ , ce qui correspond à la fonction *y1* en fonction de *t*.
- \* Le champ *nbdots* détermine le nombre de points à calculer pour la fonction (50 par défaut).
- \* Le champ *method* détermine la méthode à utiliser, les valeurs possibles sont "rkf45" (valeur par défaut), ou "rk4".
- \* Le champ *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- \* le champ *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.

### Courbes implicites : Dimplicit

- La fonction **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 du paramètre *grid*.
  - L'argument obligatoire *f* est une fonction  $f : (x, y) \rightarrow f(x, y)$  à valeurs dans R.
  - Les arguments *x1*, *x2*, *y1*, *y2* définissent la fenêtre du tracé, qui sera le pavé  $[x_1, x_2] \times [y_1, y_2]$ , on doit avoir  $x1 < x2$  et  $y1 < y2$ .
  - L'argument *grid* est une table contenant deux entiers positifs :  $\{n1, n2\}$ , le premier entier indique le nombre de subdivisions suivant *x*, et le second le nombre de subdivisions suivant *y*.
- La méthode **g:Dimplicit(f,args)** fait le dessin de la courbe implicite d'équations  $f(x, y) = 0$ .
  - L'argument obligatoire *f* est une fonction  $f : (x, y) \rightarrow f(x, y)$  à valeurs dans R.
  - L'argument *args* permet de définir les paramètres du tracé, c'est une table à 3 champs :

```
{ view={x1,x2,y1,y2}, grid={n1,n2}, draw_options="" }
```

- \* Le champ *view* détermine la zone de dessin  $[x_1, x_2] \times [y_1, y_2]$ . Par défaut on a  $view=\{g:Xinf(), g:Xsup(), g:Yinf(), g:Ysup()\}$ ,
- \* le champ *grid* détermine la grille, ce champ vaut par défaut  $\{50, 50\}$ ,
- \* le champ *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

### Courbes de niveau : Dcontour

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

- L'argument *z* (obligatoire) est la liste des différents niveaux à tracer.
- L'argument *args* (facultatif) permet de définir les paramètres du tracé, c'est une table à 4 champs :

```
{ view={x1,x2,y1,y2}, grid={n1,n2}, colors={"color1","color2",...}, draw_options="" }
```

- Le champ *view* détermine la zone de dessin  $[x1,x2] \times [y1,y2]$ , par défaut on a  $view=\{g:Xinf(), g:Xsup(), g:Yinf(), g:Ysup()\}$ .
- Le champ *grid* détermine la grille, par défaut on a  $grid=\{50, 50\}$ .
- Le champ *colors* est la liste des couleurs par niveau, par défaut cette liste est vide et c'est la couleur courante de tracé qui est utilisée.
- Le champ *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

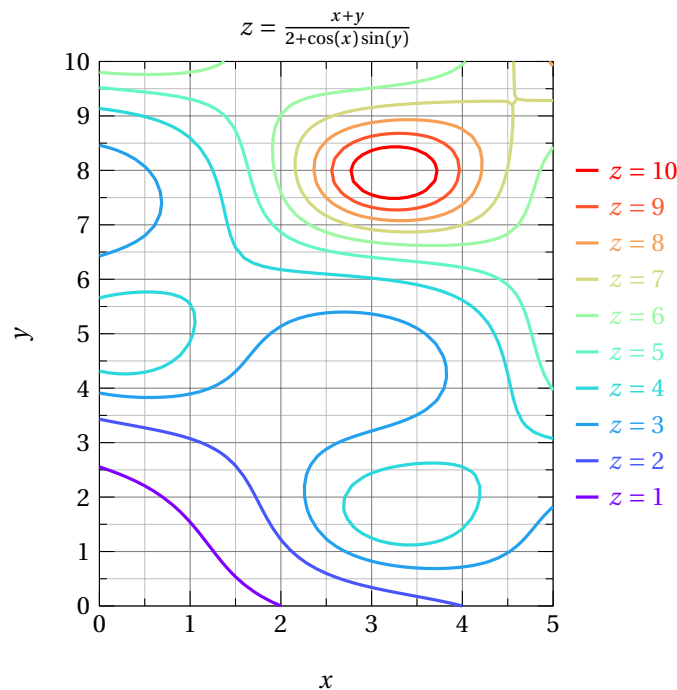
```
1 \begin{luadraw}{name=Dcontour}
2 local g = graph:new{window={-1,6.5,-1.5,11},size={10,10,0}}
3 local i, sin, cos = cpx.I, math.sin, math.cos
4 local f = function(x,y) return (x+y)/(2+cos(x)*sin(y)) end
5 local Lz = range(1,10) -- niveaux à tracer
6 local Colors = getpalette(palRainbow,10)
7 g:Dgradbox({0,5+10*i,1,1},{legend={"$x$","$y$"}, grid=true, title="$z=\frac{x+y}{2+\cos(x)\sin(y)}$"})
8 g:Linewidth(12); g:Dcontour(f,Lz,{view={0,5,0,10}, colors=Colors})
9 for k = 1, 10 do
```

```

10 local y = (2*k+4)/3*i
11 g:Dseg({5.25+y,5.5+y},1,"color"..Colors[k])
12 g:Labelcolor(Colors[k])
13 g:Dlabel("$z="..k.."$",5.5+y,{pos="E"})
14 end
15 g:Show()
16 \end{luadraw}

```

FIGURE 7 : 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  $L$ ).

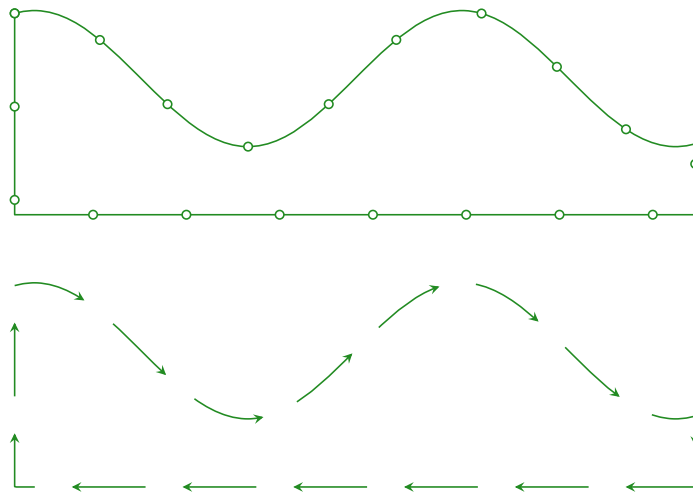
La fonction **curvilinear\_param**( $L, \text{close}$ ) renvoie une fonction d'une variable  $t \in [0; 1]$  et à valeurs sur la ligne  $L$  (nombres complexes), la valeur en  $t = 0$  est le premier point de  $L$ , et la valeur en  $t = 1$  est le dernier point; cette fonction est suivie d'un nombre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

```

1 \begin{luadraw}{name=curvilinear_param}
2 local g = graph:new{bbox=false,size={10,10}}
3 local i = cpx.I; g:Linewidth(8)
4 local L = cartesian(math.sin,-5,5)[1]
5 insert(L, {5-2*i, -5-2*i})
6 local f = curvilinear_param(L, true)
7 local I = map(f,linspace(0,1,20)) -- 20 points répartis sur L
8 g:Shift(4*i)
9 g:Lineoptions(nil,"ForestGreen",6); g:Dpolyline(L,true)
10 g:Filloptions("full","white"); g:Ddots(I) -- le premier et le dernier point sont confondus car L est fermée
11
12 -- autre exemple d'utilisation:
13 local nb = 16 --nb arrows
14 local t = linspace(0,1,3*nb+1)
15 g:Shift(-4*i)
16 for k = 0,nb-1 do
17     g:Dparametric(f,{t={t[3*k+1],t[3*k+3]}},nbdots=10,nbdiv=2,draw_options="-stealth")
18 end
19 g:Show()
20 \end{luadraw}

```

FIGURE 8 : Points répartis sur une ligne polygonale



## 5) Domaines liés à des courbes cartésiennes

### Ddomain1

- La fonction **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  $f$  sur un intervalle  $[a; b]$ , l'axe  $Ox$ , et les droites  $x = a$ ,  $x = b$ .
- La méthode **g:Ddomain1(f,args)** dessine ce contour. L'argument *args* (facultatif) permet de définir les paramètres pour la courbe, c'est une table à 5 champs :

```
{ x={a,b}, nbdots=50, discont=false, nbdiv=5, draw_options="" }
```

- Le champ  $x$  détermine l'intervalle d'étude, par défaut il vaut  $\{g:Xinf(), g:Xsup()\}$ .
- Le champ *nbdots* détermine le nombre de points à calculer pour la fonction (50 par défaut).
- Le champ *discont* indique s'il y a ou non des discontinuité pour la fonction (*false* par défaut).
- Le champ *nbdiv* est utilisé dans la méthode de calcul des points de la courbe (5 par défaut).
- Le champ *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction `\draw`.

### Ddomain2

- La fonction **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  $f$ , la courbe de la fonction  $g$ , et les droites  $x = a$ ,  $x = b$ .
- La méthode **g:Ddomain2(f,g,args)** dessine ce contour. L'argument *args* (facultatif) permet de définir les paramètres pour les courbes, c'est une table à 6 champs :

```
{ x={a,b}, nbdots=50, discont=false, nbdiv=5, draw_options="" }
```

- Le champ  $x$  détermine l'intervalle d'étude, par défaut il vaut  $\{g:Xinf(), g:Xsup()\}$ .
- Le champ *nbdots* détermine le nombre de points à calculer pour la fonction (50 par défaut).
- Le champ *discont* indique s'il y a ou non des discontinuité pour la fonction (*false* par défaut).
- Le champ *nbdiv* est utilisé dans la méthode de calcul des points de la courbe (5 par défaut).
- Le champ *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction `\draw`.

### Ddomain3

- La fonction **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  $f$  et celle de la fonction  $g$  (avec recherche de points d'intersection dans l'intervalle  $[a; b]$ ).
- La méthode **g:Ddomain3(f,g,args)** dessine ce contour. L'argument *args* (facultatif) permet de définir les paramètres pour la courbe, c'est une table à 5 champs :

```
{ x={a,b}, nbdots=50, discont=false, nbdiv=5, draw_options="" }
```

- Le champ  $x$  détermine l'intervalle d'étude, par défaut il vaut  $\{g:Xinf(), g:Xsup()\}$ .
- Le champ *nbdots* détermine le nombre de points à calculer pour la fonction (50 par défaut).

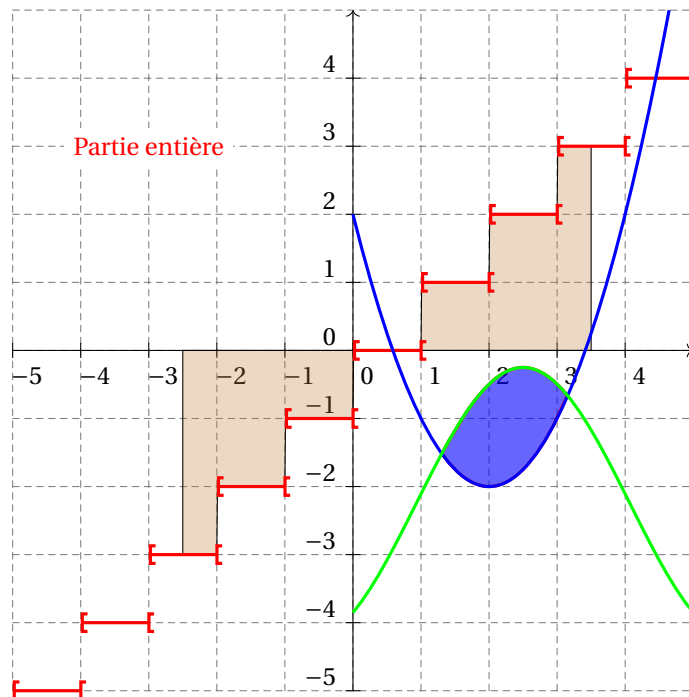
- Le champ *discont* indique s'il y a ou non des discontinuité pour la fonction (false par défaut).
- Le champ *nbdiv* est utilisé dans la méthode de calcul des points de la courbe (5 par défaut).
- Le champ *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction `\draw`.

```

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

```

FIGURE 9 : Partie entière, fonctions Ddomain1 et Ddomain3



## 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 liste de complexes. Les points sont dessinés dans la couleur courante du tracé de lignes.
  - L'argument *mark\_options* est une chaîne de caractères facultative qui sera passée telle quelle à l'instruction `\draw` (modifications locales), exemple :

```
"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, args1, text2, anchor2, args2, ...).**

- 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 *args1*, *args2*,... permettent de définir localement les paramètres des labels, ce sont des tables à 4 champs :

```
{ pos=nil, dist=0, dir={dirX,dirY,dep}, node_options="" }
```

- \* Le champ *pos* indique la position du label 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.
- \* Le champ *dist* est une distance en cm qui vaut 0 par défaut, c'est la distance entre le label et son point d'ancrage lorsque *pos* n'est pas égal à *center*.
- \* *dir={dirX,dirY,dep}* est 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.
- \* L'argument *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'argument *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 *dir={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={}*, cela représente le sens usuel de l'écriture.
- La méthode **g:Dlabeldot(texte,anchor,args)** 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 *args* (facultatif) permet de définir les paramètres du label et du point, c'est une table à 4 champs :

```
{ pos=nil, dist=0, node_options="", mark_options="" }
```

On retrouve les champs identiques à ceux de la méthode *Dlabel*, plus le champ *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 :

```
1 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" }
```

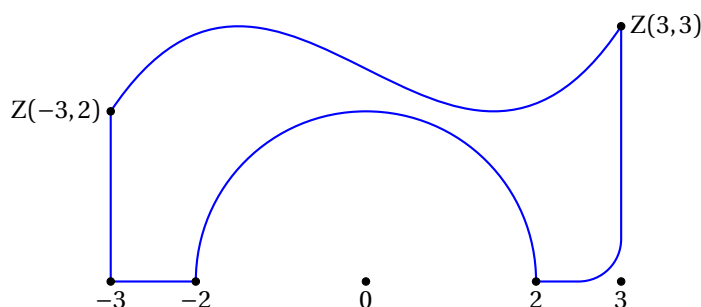
Le chemin  $L$  est composé de cinq morceaux, qui sont :

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 :

```
1 \begin{luadraw}{name=path_example}
2 local g = graph:new{window={-4,4,-0.5,3}, size={10,10}}
3 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" }
4 g:Dpath(L, "line width=0.8pt, blue")
5 g:Ddots({Z(-3,2), -3, -2, 0, 2, 3, Z(3,3)})
6 g:Dlabel("$Z(-3,2)$", Z(-3,2), {pos="W"}, "$-3$", -3, {pos="S"}, "$-2$", -2, {}, "$0$", 0, {}, "$2$", 2, {}, "$3$", 3, {}},
7   "$Z(3,3)$", Z(3,3), {pos="E"})
8 g:Show()
9 \end{luadraw}
```

FIGURE 10 : 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  $z1$ .
- $z1, \dots, zn, "l"$  (lineto), dessine la ligne polygonale  $\{last, z1, \dots, zn\}$ .
- $c1, c2, z2, "b"$  (bézier) dessine la courbe de Bézier  $\{last, c1, c2, z2\}$ , où  $c1$  et  $c2$  sont les deux points de contrôle.
- $z1, \dots, zn, "s"$  (spline), dessine la spline cubique naturelle passant par les points  $\{last, z1, \dots, zn\}$ .
- $z1, "c"$  (cercle), dessine le cercle de centre  $z1$  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*,  $z1$  et  $z2$ .
- $z1, z2, r, sens, "ca"$  (arc de cercle), dessine un arc de cercle de centre  $z1$ , de rayon  $r$ , allant de *last* vers  $z2$ , 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  $z1$  allant de *last* vers  $z2$ ,  $rx$  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  $z1$ , 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.
- $z1, \dots, zn, r, "la"$  (line arc), dessine la ligne polygonale  $\{last, z1, \dots, zn\}$  en remplaçant chaque "angle" par un arc de cercle de rayon  $r$ .
- $z1, \dots, zn, r, "cla"$  (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).

## Dessiner un chemin

- La fonction **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, sa valeur par défaut est la variable globale *bezier\_nbdots* qui est initialisée à 8.
- La méthode **g:Dpath(chemin, draw\_options)** fait le dessin du *chemin* (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*.
  - L'argument *chemin* a été décrit ci-dessus.
  - L'argument *draw\_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **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*, 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*.

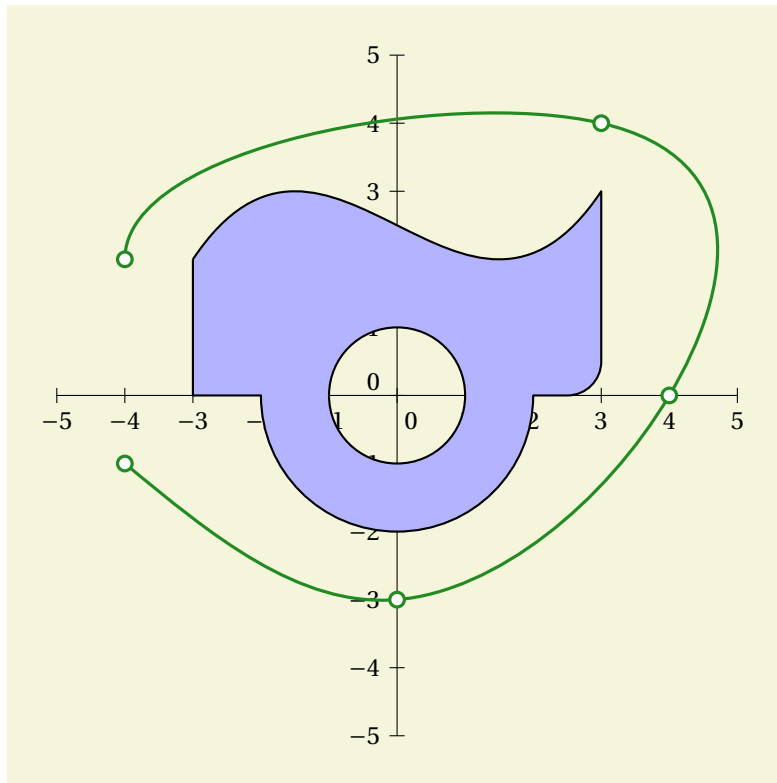
```

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

```



FIGURE 11 : Path et Spline



- La fonction **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** 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.

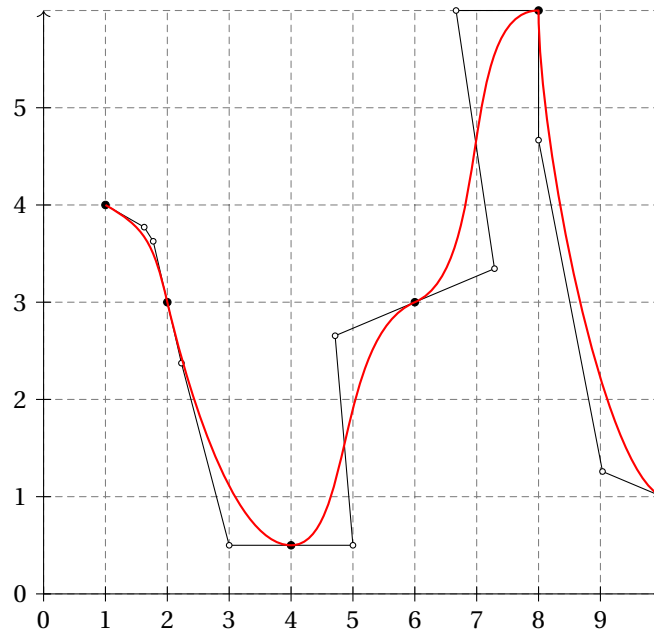
- La méthode **g:Dtcurve(L,options)** fait le dessin du chemin obtenu par *tcurve* décrit ci-dessus. L'argument *options* est une table à deux champs :
  - *showdots=true/false* (false par défaut), cette option permet de dessiner les points d'interpolation donnés ainsi que les points de contrôles calculés, ce qui permet une visualisation des contraintes.
  - *draw\_options=""*, c'est une chaîne de caractères qui sera passée directement à l'instruction `\draw`.

```

1 \begin{luadraw}{name=tcurve}
2 local g = graph:new{window={-0.5,10.5,-0.5,6.5},size={10,10,0}}
3 local i = cpx.I
4 local L = {
5     1+4*i,{2,-20},
6     2+3*i,{2,-70},
7     4+i/2,{3,0},
8     6+3*i,{4,15},
9     8+6*i,{4,0,4,-90}, -- point anguleux
10    10+i,{3,-15}}
11 g:Dgrid({0,10+6*i},{gridstyle="dashed"})
12 g:Daxes(nil,{limits={{0,10},{0,6}},originpos={"center","center"},arrows="->"})
13 g:Dtcurve(L,{showdots=true,draw_options="line width=0.8pt,red"})
14 g:Show()
15 \end{luadraw}

```

FIGURE 12 : Courbe d'interpolation avec vecteurs tangents imposés



## 8) Chemins et clipping : `Beginclip()` et `Endclip()`

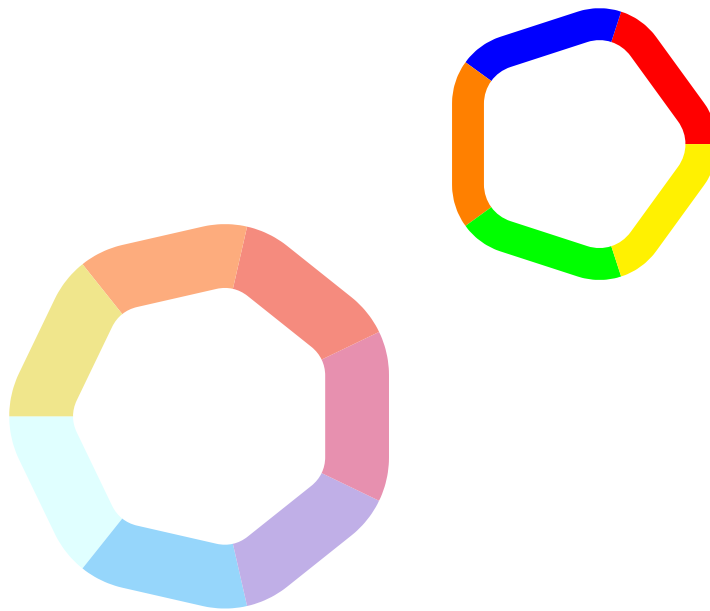
Un chemin peut être utilisé pour faire du clipping grâce à deux fonctions : `g:Beginclip(chemin,inverse)` et `g:Endclip()`. La première ouvre un groupe *scope* et passe le *chemin* comme argument à la fonction `\clip` de tikz. La seconde referme le groupe *scope*, elle est indispensable (sinon il y aura une erreur de compilation). 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.

```

1 \begin{luadraw}{name=polygon_with_different_line_color_and_rounded_corners}
2 local g = graph:new{window={-5,5,-5,5},size={10,10}}
3 local i = cpx.I
4 local Dcolored_polyreg = function(c,a,nb,r,wd,colors)
5   -- c=center, a=vertice, nb=number of sides, r=radius, wd=width in point, colors=list of colors
6   local L = polyreg(c,a,nb)
7   insert(L,{r,"cla"}) --polygon width rounded corners (radius=r)
8   local angle = 360/nb
9   local b = a
10  for k = 1, nb do
11    a = b; b = rotate(a,angle,c)
12    g:Beginclip({2*a-c,c,2*b-c,"1"}) -- définition d'un secteur angulaire pour clipper
13    g:Dpath(L,"line width"..wd.."pt,color"..colors[k])
14    g:Endclip()
15  end
16 end
17 Dcolored_polyreg(3+2*i,5+2*i,5,0.8,12,{"red","blue","orange","green","yellow"}) -- pentagon
18 Dcolored_polyreg(-2.5-2*i,-5-2*i,7,1,24,getpalette(palGasFlame,7)) -- heptagon
19 g:Show()
20 \end{luadraw}

```

FIGURE 13 : Exemple de clipping



## 9) Axes et grilles

Variables globales utilisées pour les axes et les grilles :

- *maxGrad* = 100 : nombre max de graduations sur un axe.
- *defaultlabelshift* = 0.125 : lorsqu'une grille est dessinée avec les axes (option *grid=true*) les labels sont automatiquement décalés le long de l'axe avec cette variable.
- *defaultxylabsep* = 0 : définit la distance par défaut entre les labels et les graduations.
- *defaultlegendsep* = 0.2 : définit la distance par défaut entre la légende et l'axe.
- *digits* = 4 : nombre de décimales par défaut dans les conversions en chaînes de caractères, les 0 terminaux sont supprimés.
- *dollar* = true : pour ajouter des dollars autour des labels des graduations.
- *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 **num(x)**, celle-ci transforme le nombre *x* en une chaîne *str* avec le nombre de décimales fixées par la variable globale *digits*, lorsque la variable *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 :

```
1 function num(x) -- x is a real, returns a string
2   local rep = strReal(x) -- conversion to string with digits decimals max
3   if siunitx then rep = "\num{..rep..}" end --needs \usepackage{siunitx}
4   return rep
5 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 `"->"` 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.
  - `xylabsep={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 de la graduation au croisement des axes (graduation numéro 0).

La formule qui définit le label à la graduation numéro  $n$  est :  $(\text{originnum} + \text{unit} \cdot n) \text{labeltext} / \text{labelden}$ .

- `originloc=A`. Cette option précise le point de croisement des axes.
- `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={0.2,0.2}`. 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, 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"`.
- `labelden={1,1}`. Cette option précise le dénominateur des labels (entier) pour l'axe. La formule qui définit le label à la graduation numéro  $n$  est :  $(\text{originnum} + \text{unit} \cdot n) \text{labeltext} / \text{labelden}$ .
- `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"`.
- `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.

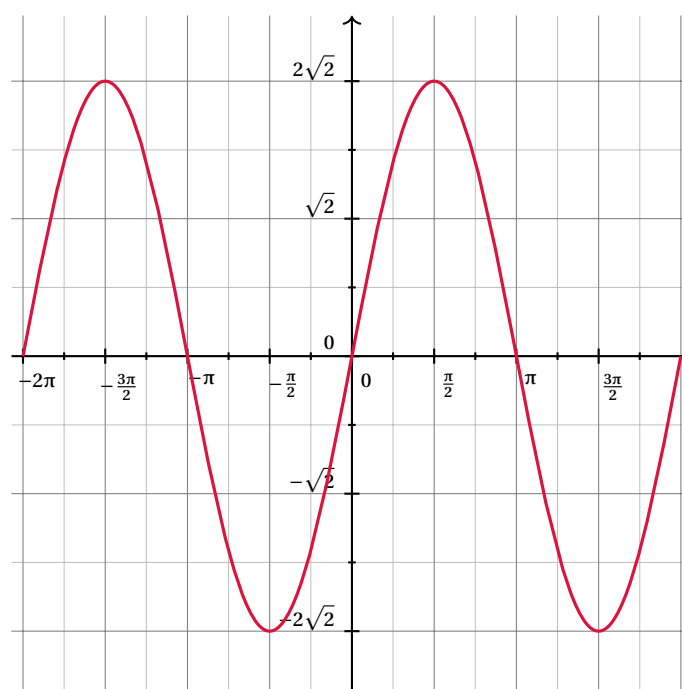
- `labelshift={0,0}`. Cette option permet de définir un décalage systématique pour les labels sur l'axe (décalage de long de l'axe).
- `nbdeci={2,2}`. Cette option précise le nombre de décimales pour les valeurs numériques sur l'axe.
- `numericFormat={0,0}`. Cette option précise le type d'affiche numérique (non encore implémenté).
- `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  $A+pos1 \cdot xpas$ .
- `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  $A+pos1 \cdot ypas \cdot i$ .
- `grid=false`. Cette option permet d'ajouter ou non une grille.
- `drawbox=false`. Cette option de dessiner les axes sous la forme d'une boîte, 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).

```

1 \begin{luadraw}{name=axes_grid}
2 local g = graph:new{window={-6.5,6.5,-3.5,3.5}, size={10,10,0}}
3 local i, pi, a = cpx.I, math.pi, math.sqrt(2)
4 local f = function(x) return 2*a*math.sin(x) end
5 g:Labelsize("footnotesize"); g:Linewidth(8)
6 g:Daxes({0,pi/2,a},{labeltext={"\\pi", "\\sqrt{2}"}, labelden={2,1}, nbsubdiv={1,1}, grid=true, arrows="->"})
7 g:Lineoptions("solid", "Crimson", 12); g:Dcartesian(f, {x={-2*pi, 2*pi}})
8 g:Show()
9 \end{luadraw}

```

FIGURE 14 : 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  $A$ ) et le pas des graduations sur l'axe. Par défaut le point  $A$  est l'origine  $Z(0, 0)$ , et le pas est égal à 1.
- L'argument *options* 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=0.2**. 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, 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"** 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"**.
- **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.

- `numericFormat=0`. Cette option précise le type d’affiche numérique (non encore implémenté).
- `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’affiche 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  $M = A + xu$ . 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=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, *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, 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"`. Cette option précise la position des labels par rapport à l’axe, les valeurs possibles sont : `"none"`, `"bottom"` ou `"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"`.
- `labelangle=0`. Cette option définit l’angle des labels en degrés par rapport à l’horizontale.



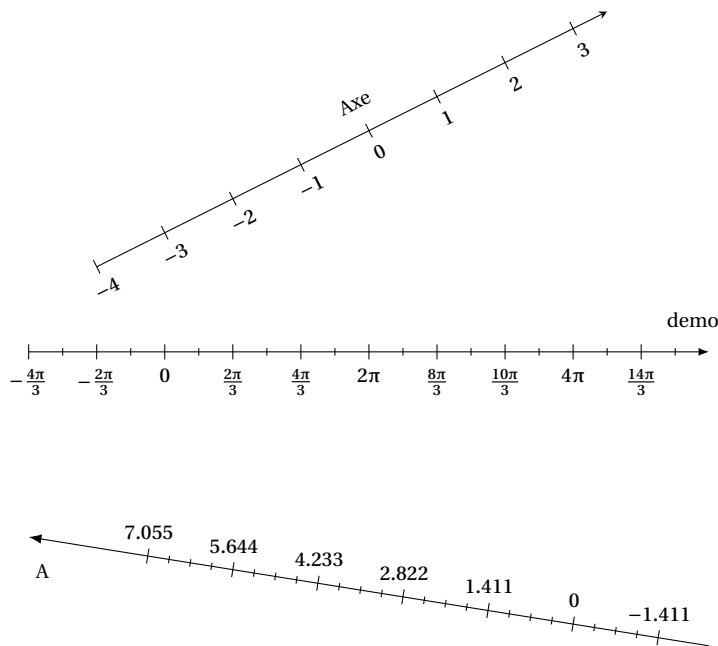
- `labelcolor=""`. Cette option permet de choisir une couleur pour les labels. La chaîne vide représente la couleur courante du texte.
- `labelshift=0`. Cette option permet de définir un décalage systématique pour les labels sur l'axe (décalage de long de l'axe).
- `nbdeci=2`. Cette option précise le nombre de décimales pour les labels numériques.
- `numericFormat=0`. Cette option précise le type d'affiche numérique (non encore implémenté).
- `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 :  $\{x_1, \text{"text1"}, x_2, \text{"text2"}, \dots\}$ . Les nombres  $x_1, x_2, \dots$  représentent des abscisses dans le repère  $(A, u)$ .

```

1 \begin{luadraw}{name=gradline}
2 local g = graph:new{window={-5,5,-5,5},size={10,10}}
3 g:Labelsize("footnotesize")
4 local i = cpx.I
5 g:Dgradline({3.25*i,1+i/2}, {limits={-4,4}, legend="Axe", legendpos=0.5, arrows="-stealth"})
6 g:Dgradline({-3,1}, {legend="demo", labeltext="\pi", labelden=3, unit=2, nbsubdiv=1, arrows="-latex"})
7 g:Dgradline({3-4*i,-1.25+i/5}, {legend="A", labelstyle="N", gradlimits={-1,5}, nbsubdiv=3, unit=1.411, nbdeci=3,
8   arrows="-Latex"})
9 g:Show()
10 \end{luadraw}

```

FIGURE 15 : 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.
  - `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.

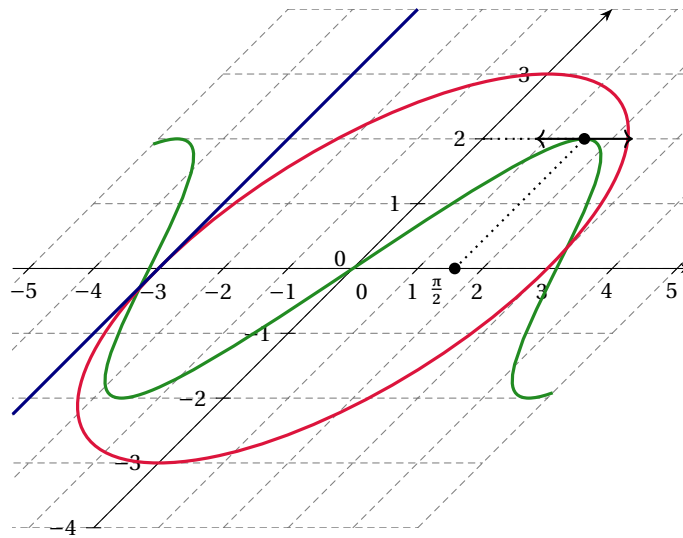
**Exemple :** il est possible de travailler dans un repère non orthogonal. Voici un exemple où l'axe  $Ox$  est conservé, mais la première bissectrice devient le nouvel axe  $Oy$ , on modifie pour cela la matrice de transformation du graphe. À partir de cette modification les affixes représentent les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

```

1 \begin{luadraw}{name=axes_non_ortho}
2 local g = graph:new{window={-5.25,5.25,-4,4},size={10,10}}
3 local i, pi = cpx.I, math.pi
4 local f = function(x) return 2*math.sin(x) end
5 g:Setmatrix({0,1,1+i}); g:Labelsize("small")
6 g:Dgrid({-5-4*i,5+4*i},{gridstyle="dashed"})
7 g:Daxes({0,1,1}, {arrows="-Stealth"})
8 g:Lineoptions("solid","ForestGreen",12); g:Dcartesian(f,{x={-5,5}})
9 g:Dcircle(0,3,"Crimson")
10 g:DlineEq(1,0,3,"Navy") -- droite d'équation x=-3
11 g:Lineoptions("solid","black",8); g:DtangentC(f,pi/2,1.5,"<->")
12 g:Dpolyline({pi/2,pi/2+2*i,2*i},"dotted")
13 g:Ddots(Z(pi/2,2))
14 g:Dlabeldot("$\\frac{\\pi}{2}$",pi/2,{pos="SW"})
15 g:Show()
16 \end{luadraw}

```

FIGURE 16 : 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 boîte, attention cependant à laisser suffisamment de place pour cela.

```

1 \begin{luadraw}{name=gradbox}
2 local g = graph:new{window={-5,4,-5.5,5},size={10,10}}
3 local i, pi = cpx.I, math.pi
4 local h = function(x) return x^2/2-2 end
5 local f = function(x) return math.sin(3*x)+h(x) end
6 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={"Legend $x$", "Legend $y$"}})

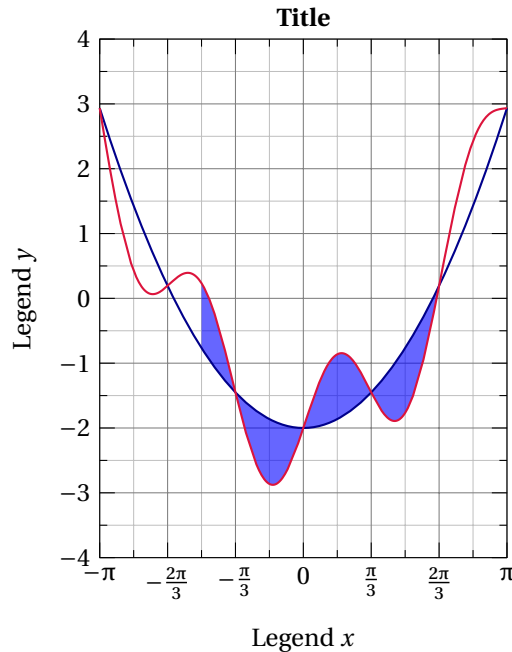
```

```

7 g:Saveattr(); g:Viewport(-pi,pi,-4,4) -- on limite la vue (clip)
8 g:Filloptions("full","blue",0.6); g:Linestyle("noline"); g:Ddomain2(f,h,{x={-pi/2,2*pi/3}})
9 g:Filloptions("none",nil,1); g:Lineoptions("solid",nil,8); g:Dcartesian(h,{x={-pi,pi}, draw_options="DarkBlue"})
10 g:Dcartesian(f,{x={-pi,pi},draw_options="Crimson"})
11 g:Restoreattr()
12 g:Show()
13 \end{luadraw}

```

FIGURE 17 : Utilisation de Dgradbox



## 10) Dessins d'ensembles (diagrammes de Venn)

### Dessiner un ensemble

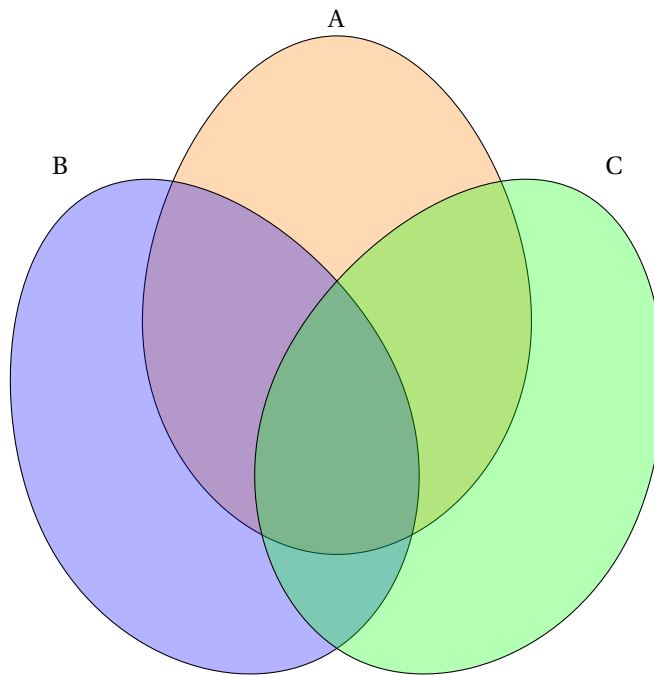
La fonction **set(center,angle,scale)** renvoie un chemin représentant un ensemble (en forme d'oeuf), donc le centre est *center* (complexe), l'argumen *angle* représente l'inclinaison (en degrés) de l'axe vertical de l'ensemble (0 par défaut), et l'argument *scale* est un facteur d'échelle permettant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

```

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

```

FIGURE 18 : 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 **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 **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 **setminus(C1,C2)** renvoie une liste de complexes représentant le contour de la différence des ensembles correspondant à  $C_1$  et  $C_2$  ( $C_1 \setminus C_2$ ).

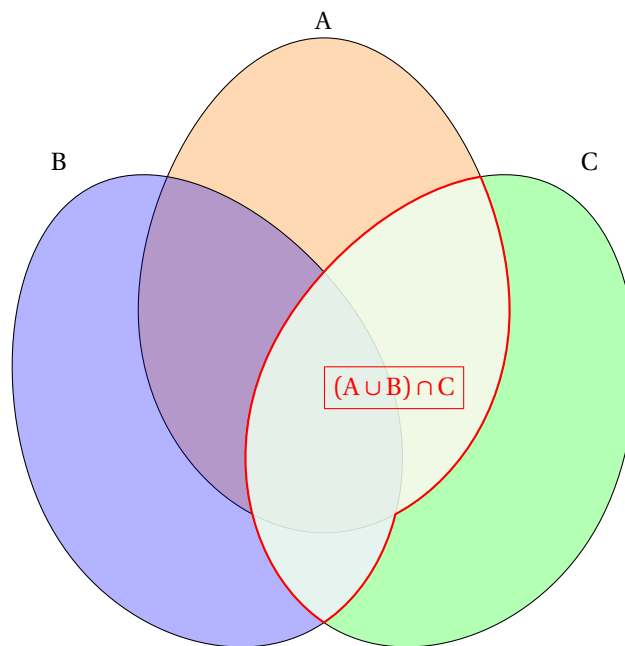
Le résultat de ces opérations, étant une liste de complexes, peut être dessiné avec la méthode **g:Dpolyline()**.

```

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

```

FIGURE 19 : 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.

## 11) 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),**

où :

- *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`,
  - *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.

```

1 \begin{luadraw}{name=Dimage}
2 local g = graph:new{margin={0,0,0,0}, size={10,10},bg="green!30"}
3 local filename = cachedir.."flower.jpg"
4 local g_options = "width=4.5cm,height=4.5cm"
5 g:Dimage(filename,Z(-5,5),{pos="SE", graphics_options=g_options})
6 g:Dimage(filename,Z(5,5),{pos="SE", matrix={0,-1,cpx.I}, graphics_options=g_options})
7 g:Rotate(180)
8 g:Dimage(filename,Z(-5,5),{pos="SE", graphics_options=g_options})
9 g:IDmatrix()
10 g:Dimage(filename,Z(-5,-5),{pos="SE", matrix={0,1,-cpx.I}, graphics_options=g_options})
11 g:Dlabel( "Original", Z(-5,5),{pos="SE"},
12         "Vertical symetry", Z(5,5), {pos="SW"},
13         "Rotation (global)", Z(5,-5), {pos="NW"},

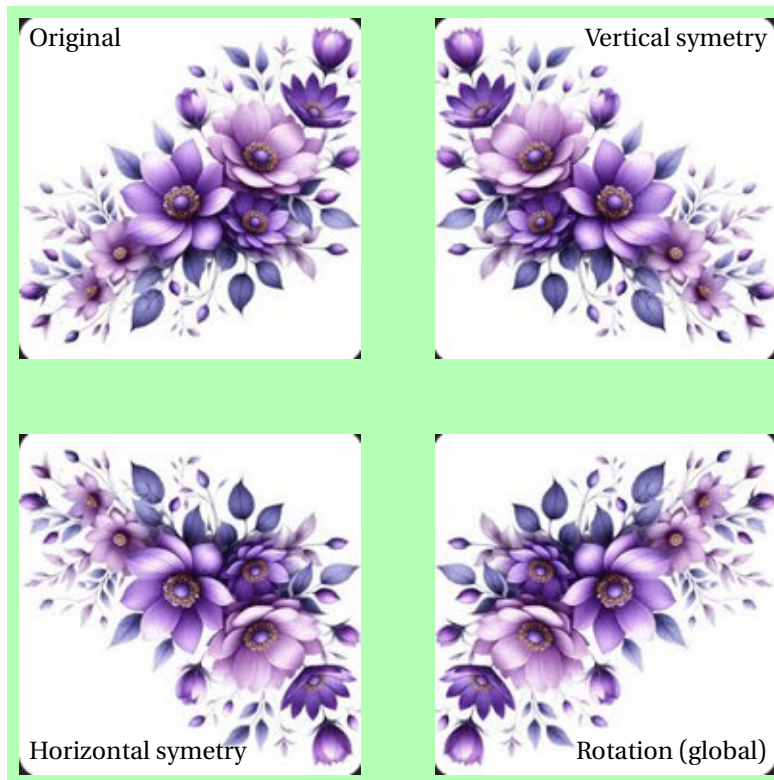
```

```

14 "Horizontal symetry", Z(-5,-5), {pos="NE"} )
15 g:Show()
16 \end{luadraw}

```

FIGURE 20 : Importer une image



### Mapper une image sur un parallélogramm

Ceci est possible avec la méthode : `g:Dmapimage( filename, parallelo, options )`, 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.

```

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

```

FIGURE 21 : Mapper une image



## 12) 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 :

```
1 AliceBlue = {0.9412, 0.9725, 1}
2 AntiqueWhite = {0.9804, 0.9216, 0.8431}
3 Aqua = {0.0, 1.0, 1.0}
4 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 **rgb()** de *luadraw*, par exemple : `g:Linecolor(rgb(AliceBlue))`. Par contre, avec cette fonction *rgb()*, pour changer localement de couleur il faut faire comme ceci (exemple) : `g:Dpolyline(L, "color=" .. rgb(AliceBlue))`, ou `g:Dpolyline(L, "fill=" .. 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 **rgb(r,g,b)** ou **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 **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 entier 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 **mixcolor(color1,proportion1 color2,proportion1,...,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 **palette(colors,pos,table)** : l'argument *colors* est une liste (table) de couleurs au format {r , b , g}, 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 **getpalette(colors,nb,table)** : l'argument *colors* est une liste (table) de couleurs au format {r , b , g}, 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.

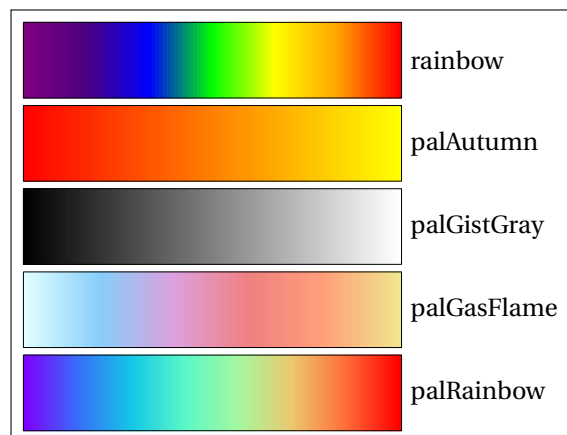
Par défaut, il y a cinq palettes de couleurs<sup>2</sup>.

```

1 \begin{luadraw}{name=palettes}
2 local g = graph:new{window={-5,5,-5,5},bbox=false, border=true}
3 g:Linewidth(1)
4 local Dpalette = function(pal,A,h,L,N,name)
5     local dl = L/N
6     for k = 1, N do
7         local color = palette(pal,(k-1)/(N-1))
8         g:Drectangle(A,A+h,A+h+dl,"color"..color.."",fill="..color")
9         A = A+dl
10    end
11    g:Drectangle(A,A+h,A+h-L); g:Dlabel(name,A+h/2,{pos="E"})
12 end
13 local A, h, dh, L, N = Z(-5,4), Z(0,-1), Z(0,-1.1), 5, 100
14 Dpalette(rainbow,A,h,L,N,"rainbow"); A = A+dh
15 Dpalette(palAutumn,A,h,L,N,"palAutumn"); A = A+dh
16 Dpalette(palGistGray,A,h,L,N,"palGistGray"); A = A+dh
17 Dpalette(palGasFlame,A,h,L,N,"palGasFlame"); A = A+dh
18 Dpalette(palRainbow,A,h,L,N,"palRainbow")
19 g:Show()
20 \end{luadraw}

```

FIGURE 22 : Les cinq palettes par défaut



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.

### III Constructions géométriques

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

#### 1) `circumcircle()`, `incircle()`

- La fonction `circumcircle(a,b,c)` (ou `circumcircle({a,b,c})`), où  $a$ ,  $b$  et  $c$  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 `incircle3d(a,b,c)` (ou `incircle3d({a,b,c})`), où  $a$ ,  $b$  et  $c$  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 `cvx_hull2d(L)` où  $L$  est une liste de complexes, calcule et renvoie une liste de complexes représentant l'enveloppe convexe de  $L$ .

#### 3) `delaunay()`

La fonction `delaunay(L)` où  $L$  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  $L$  (le cercle circonscrit de chacun des triangles ne contient aucun des autres points).

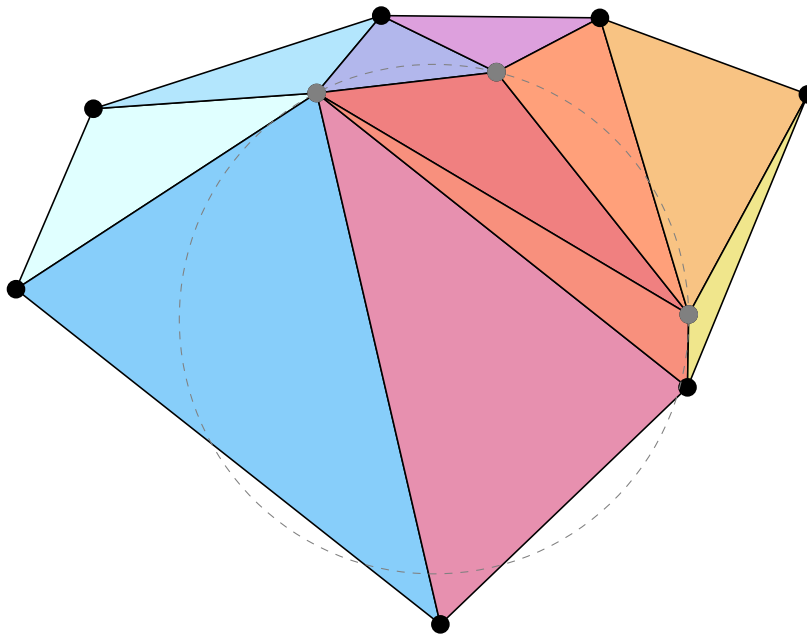
```

1  \begin{luadraw}{name=delaunay}
2  local g = graph3d:new{bbox=false, pictureoptions="scale=2"}
3  local i = cpx.I; g:Linewidth(6)
4  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,
5  ↵ -0.086-2.191*i,-0.477+1.834*i,-0.904+1.322*i,-2.892+0.025*i}
6  local T = delaunay(L) -- list of triangles
7  local n = #T
8  local num = 7 -- we choose a triangle
9  local colors = getpalette(palGasFlame,n)
10 for k = 1, n do
11     g:Dpolyline(T[k],true,'fill'..'colors[k])
12 end
13 g:Ddots(L)
14 g:Dcircle( {circumcircle(T[num])}, "line width=0.4pt,gray,dashed" )
15 g:Ddots(T[num],"gray")
16 g:Show()
17 \end{luadraw}

```



FIGURE 23 : Triangulation de Delaunay



#### 4) voronoi()

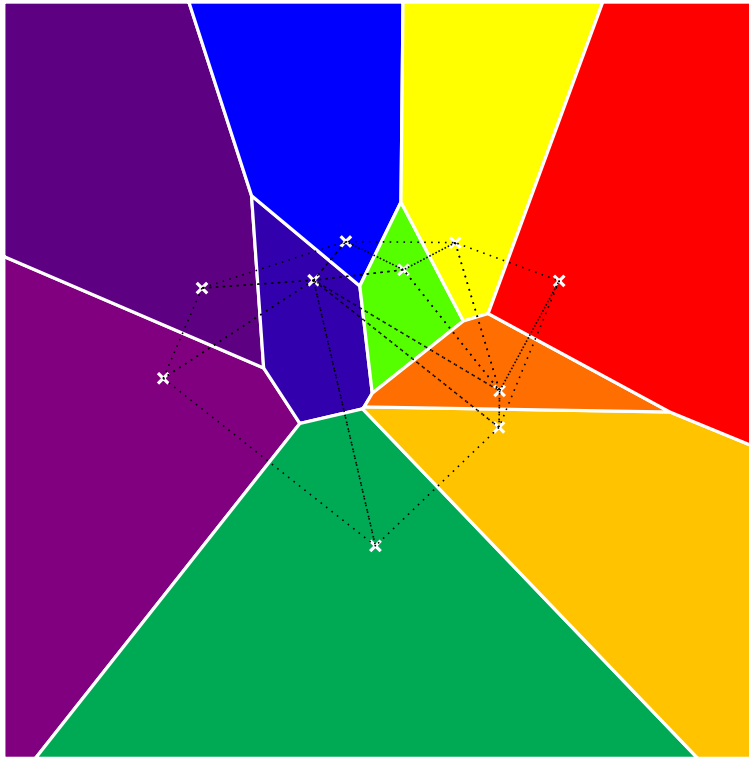
La fonction **voronoi(L,window)** où  $L$  est une liste de nombres complexes **distincts**, détermine le diagramme de Voronoï des points de la liste  $L$ . Cette fonction renvoie une liste d'éléments de la forme  $\{A, \text{polygone}\}$  où  $A$  est un point de la liste  $L$ , et  $\text{polygone}$  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 centres des cercles circonscrits aux triangles de Delaunay (attention : cela ne change pas la fenêtre 2d du graphique en cours).

```

1 \begin{luadraw}{name=voronoi}
2 local g = graph:new{ bbox=true, margin={0,0,0,0}, size={10,10}}
3 local i = cpx.I
4 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,
5           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}
6 local V = voronoi(S)
7 local colors = getpalette(rainbow,#V)
8 for k,T in ipairs(V) do
9     local A, polygon = table.unpack(T)
10    g:Dpolyline(polygon,true,"color=white, line width=1.2pt,fill="..colors[k])
11    g:Ddots(A,"mark=x,white,scale=2,line width=1.2pt")-- A is one of the points of S
12 end
13 g:Dpolyline(delaunay(S),true,"dotted,line width=0.6pt") -- Delaunay triangles
14 g:Show()
15 \end{luadraw}

```

FIGURE 24 : Diagramme de Voronoï



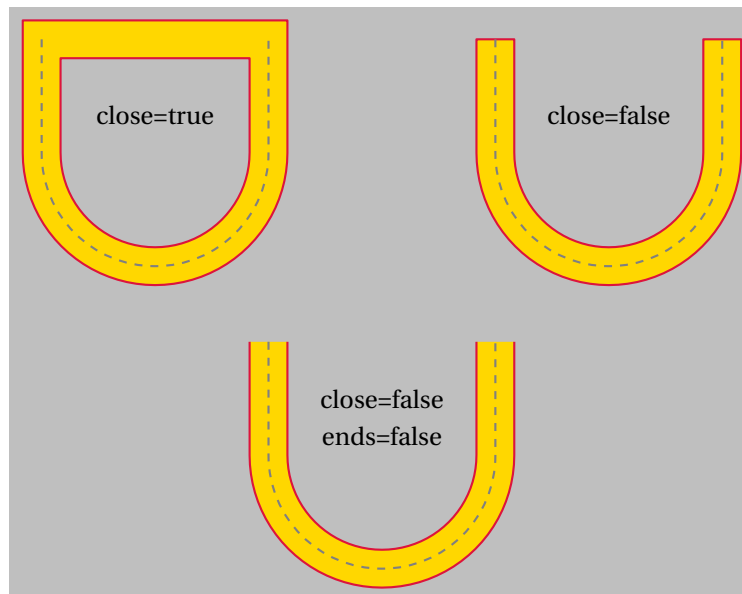
## 5) line2strip()

La fonction **line2strip(L,width,close,ends)** où  $L$  est une liste de nombres complexes, ou une liste de listes de nombres complexes, renvoie un chemin représentant une "bande" centrée 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  $ends$  est un booléen qui indique si les deux extrémités de la bande doivent être dessinées (*true* par défaut, sauf quand l'argument  $close$  vaut *true*).

```

1 \begin{luadraw}{name=line2strip}
2 local g = graph:new{bbox=false, bg="lightgray"}
3 local i = cpx.I; g:Linewidth(8)
4 local p = {-3+3*i,-3,"1",0,3,3,1,"ca", 3+3*i,"1"}
5 g:Setmatrix({-3+3*i,0.5,0.5*i})
6 local L = line2strip(path(p),1,true) -- p is first converted to polyline
7 g:Dpath(L,"Crimson,fill=Gold"); g:Dpath(p,"gray,dashed")
8 g:Dlabel("close=true",i,{})
9 g:Setmatrix({3+3*i,0.5,0.5*i})
10 local L = line2strip(path(p),1,false) -- p is first converted to polyline
11 g:Dpath(L,"Crimson,fill=Gold"); g:Dpath(p,"gray,dashed")
12 g:Dlabel("close=false",i,{})
13 g:Setmatrix({-i,0.5,0.5*i})
14 local L = line2strip(path(p),1,false,false) -- p is first converted to polyline
15 g:Dpath(L,"Crimson,fill=Gold"); g:Dpath(p,"gray,dashed")
16 g:Dlabel("close=false",1.5*i,{}); g:Dlabel("ends=false",0.5*i,{})
17 g:Show()
18 \end{luadraw}

```

FIGURE 25 : Exemple avec *line2strip*

## 6) `parallel_polyline()`

La fonction `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 côté 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 `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 `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 `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`.

```

1 \begin{luadraw}{name=sss_triangles_and_co}
2 local g = graph:new{window={-5,5,-3,5},size={10,10}}
3 g:Labelsize("footnotesize"); g:Linewidth(8)
4 local i = cpx.I
5 local T1 = shift( sss_triangle(4,5,3), 2*i-2)
6 local T2 = shift( sas_triangle(4,60,2), -4-2*i)
7 local T3 = shift( asa_triangle(30,4,50), 0.5-i)
8 g:Dpolyline({T1,T2,T3}, true)
9 g:Linewidth(4)
10 g:Darc(T2[2],T2[1],T2[3],0.5,1,"->")

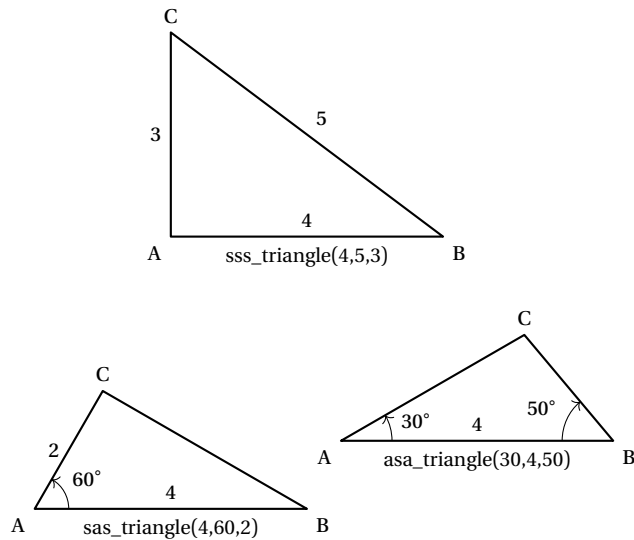
```

```

11 g:Darc(T3[2],T3[1],T3[3],0.75,1,"->")
12 g:Darc(T3[1],T3[2],T3[3],0.75,-1,"->")
13 g:Dlabel(
14     "$4$", (T1[1]+T1[2])/2, {pos="N"}, "$5$", (T1[2]+T1[3])/2, {pos="NE"}, "$3$", (T1[1]+T1[3])/2, {pos="W"},
15     "$4$", (T2[1]+T2[2])/2, {pos="N"},
16     ↪ "$60^\circ", T2[1]+Zp(0.9,30*deg), {pos="center"}, "$2$", (T2[1]+T2[3])/2, {pos="W"},
17     "$4$", (T3[1]+T3[2])/2, {pos="N"}, "$30^\circ", T3[1]+Zp(1.15,15*deg), {pos="center"},
18     "$50^\circ", T3[2]+Zp(1.15,155*deg), {pos="center"},
19     "sss\_triangle(4,5,3)", (T1[1]+T1[2])/2, {pos="S"}, "sas\_triangle(4,60,2)", (T2[1]+T2[2])/2, {},
20     ↪ "asa\_triangle(30,4,50)", (T3[1]+T3[2])/2, {})
19 for _,T in ipairs({T1,T2,T3}) do
20     g:Dlabel("$A$",T[1],{pos="SW"}, "$B$",T[2],{pos="SE"}, "$C$",T[3],{pos="N"})
21 end
22 g:Show()
23 \end{luadraw}

```

FIGURE 26 : sss\_triangle, sas\_triangle et asa\_triangle



## IV Calculs sur les listes

### 1) concat

La fonction `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 `concat( 1, 2, 3, {4, 5, 6}, 7 )` renvoie la table  $\{1, 2, 3, 4, 5, 6, 7\}$ .

### 2) cut

La fonction `cut(L,A,before)` permet de couper  $L$  au point  $A$  qui est supposé être situé sur la ligne  $L$  ( $L$  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 `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 :

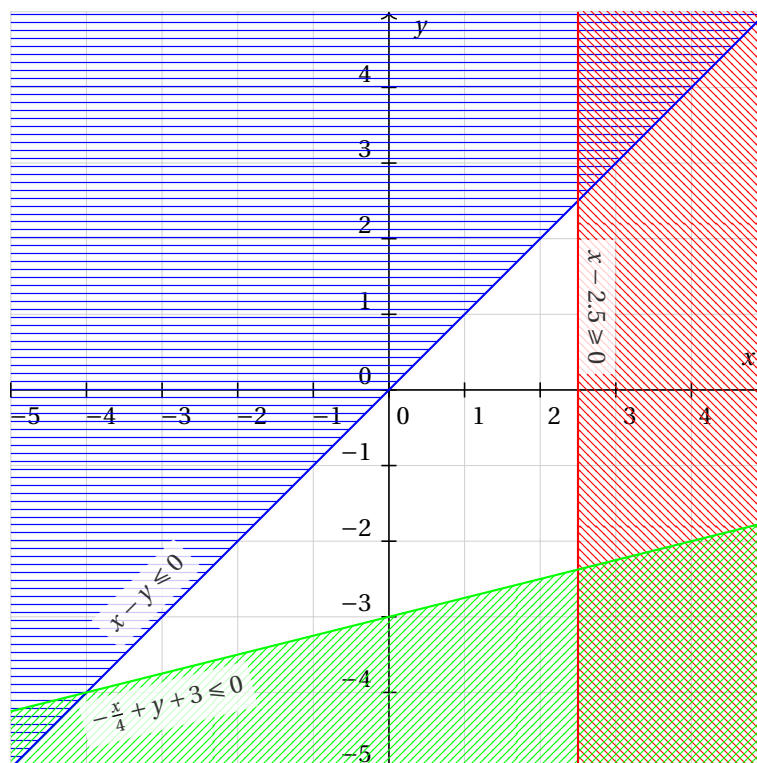
- La partie de  $L$  qui est dans le demi-plan défini par la droite à "gauche" de  $u$  (c'est à dire contenant le point  $A + iu$ ) (c'est une ligne polygonale),
- suivi de la partie de  $L$  qui est dans l'autre demi-plan (ligne polygonale),
- suivi de la liste des points d'intersection entre  $L$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

```

1 \begin{luadraw}{name=cutpolyline}
2 local g = graph:new{window={-5,5,-5,5}, size={10,10},margin={0,0,0,0}}
3 g:Linewidth(6)
4 local i = cpx.I
5 local P = g:Box2d() -- polygon representing the 2d window
6 local D1, D2, D3 = {0,1+i}, {2.5,-i}, {-3*i,-1-i/4} -- three lines
7 local P1 = cutpolyline(P,D1,true)
8 local P2 = cutpolyline(P,D2,true)
9 local P3 = cutpolyline(P,D3,true)
10 g:Daxes({0,1,1},{grid=true,gridcolor="LightGray",arrows="->",legend={"$x$","$y$"}})
11 g:Filloptions("horizontal","blue"); g:Dpolyline(P1,true,"draw=none")
12 g:Filloptions("fdiag","red"); g:Dpolyline(P2,true,"draw=none")
13 g:Filloptions("bdiag","green"); g:Dpolyline(P3,true,"draw=none")
14 g:Filloptions("none","black",1)
15 g:Linewidth(8)
16 g:Dline(D1,"blue"); g:Dline(D2,"red"); g:Dline(D3,"green")
17 g:Dlabel(
18     "$x-y\\leqslant 0$", -3-3*i, {pos="N", dir={1+i,-1+i}, dist=0.1, node_options="fill=white,fill opacity=0.8"},
19     "$x-2.5\\geqslant 0$", 2.5+i, {dir={-i,1}},
20     "$-\frac{x}{4}+y+3\\leqslant 0$", -3-15/4*i, {pos="S", dir={1+i/4,i-1/4}}
21 )
22 g:Show()
23 \end{luadraw}

```

FIGURE 27 : Illustrer un exercice de programmation linéaire



#### 4) getbounds

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

## 5) getdot

La fonction **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  $L$ .

## 6) insert

La fonction **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 **insert(L, {3,4,5},3)**, la variable  $L$  sera égale à  $\{1,2,3,4,5,6\}$ .

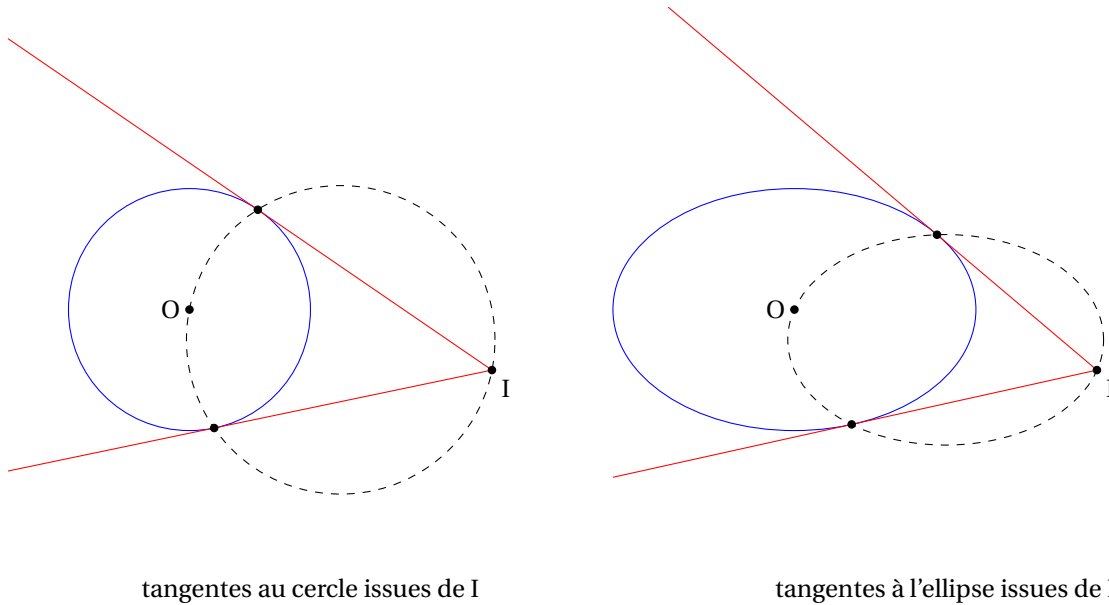
## 7) interCC

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

```

1 \begin{luadraw}{name=interCC}
2 local g = graph:new{window={-10,10,-5,5}, margin={0,0,0,0},size={16,8}}
3 local i = cpx.I
4 -- pour le cercle {0,2}
5 g:Saveattr(); g:Viewport(-10,0,-5,5); g:Coordsystem(-4,6,-5,5)
6 local O = -1
7 local C1, I = {0, 2}, 4-i
8 local C2 = {(O+I)/2,cpx.abs(I-O)/2}
9 local rep = interCC(C1,C2) -- points de tangence
10 g:Dcircle(C1,"blue"); g:Dcircle(C2,"dashed")
11 g:Dhline(I,rep[1],"red"); g:Dhline(I,rep[2],"red") --demi- tangentes
12 g:Ddots(rep); g:Ddots({0,I}); g:Dlabel("$I$",I,{pos="SE"},"$O$",0,{pos="W"},
13     "tangentes au cercle issues de $I$",1-5*i,{pos="N"})
14 g:Restoreattr()
15
16 -- pour l'ellipse (E) : {0,3,2}
17 g:Saveattr(); g:Viewport(0,10,-5,5); g:Coordsystem(-4,6,-5,5)
18 local mat = {0,1.5,i} -- cette matrice transforme un cercle {01,2} en l'ellipse (E)
19 local inv_mat = invmatrix(mat) -- matrice inverse
20 local O1, I1 = table.unpack( mtransform({0,I},inv_mat) ) -- antécédents de O et de I
21 C1 = {O1, 2}
22 C2 = {(O1+I1)/2,cpx.abs(I1-O1)/2}
23 rep = interCC(C1,C2) -- points de tangence (tangentes issues de I1)
24 g:Composematrix(mat) -- on applique la matrice pour retrouver l'ellipse, la tangence est conservée
25 g:Dcircle(C1,"blue"); g:Dcircle(C2,"dashed")
26 g:Dhline(I1,rep[1],'red'); g:Dhline(I1,rep[2],"red")
27 g:Ddots(rep); g:Ddots({O1,I1}); g:Dlabel("$I$",I1,{pos="SE"},"$O$",O1,{pos="W"},
28     "tangentes à l'ellipse issues de $I$",1-5*i,{pos="N"})
29 g:Restoreattr()
30 g:Show()
31 \end{luadraw}

```

FIGURE 28 : Tangentes à un cercle  $\{O,2\}$  et à une ellipse  $\{O,3,2\}$  issues d'un point**8) interD**

La fonction **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.

**9) interDC**

La fonction **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.

**10) interDL**

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

**11) interL**

La fonction **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).

**12) interP**

La fonction **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*).

**13) isobar**

La fonction **isobar(L)** où  $L$  est une liste de complexes, renvoie l'isobarycentre de ces nombres. Si  $L$  contient des éléments qui ne sont pas des nombres réels ou complexes, ceux-ci sont ignorés.

**14) linspace**

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

## 15) map

La fonction **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 **merge(L)** recolle si c'est possible, les composantes connexes de  $L$  qui doit être une liste de listes de complexes, la fonction renvoie le résultat.

## 17) polyline2path

La fonction **polyline2path(L)** où  $L$  est une liste de nombres complexes ou une liste de listes de nombres complexes, renvoie  $L$  sous la forme d'un chemin (que l'on peut dessiner avec la méthode *g:Dpath()*).

## 18) range

La fonction **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.

## 19) read\_csv\_file

La fonction **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 paramètres de la fonction, ceux-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* (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 de chaîne de caractères).

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*.

## 20) Fonctions de clipping

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

## 21) Ajout de fonctions mathématiques

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



**Évaluation protégée : evalf**

La fonction **evalf(f,...)** permet d'évaluer  $f(\dots)$  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 :

```
1 local f = function(a,b)
2   return 2*Z(a,1/b)
3 end
4 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 :

```
1 local f = function(a,b)
2   return 2*Z(a,1/b)
3 end
4 print(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 **int(f,a,b)** renvoie une valeur approchée de l'intégrale de la fonction  $f$  sur l'intervalle  $[a; b]$ . La fonction  $f$  est à variable réelle et à valeurs réelles ou complexes. La méthode utilisée est la méthode de Simpson accélérée deux fois avec la méthode Romberg.

**Exemple :**

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

**Résultat :**  $\int_0^1 e^{t^2} dt \approx 1.4626517459589$ .

**gcd**

La fonction **gcd(a,b)** renvoie le plus grand diviseur commun entre  $a$  et  $b$ .

**lcm**

La fonction **lcm(a,b)** renvoie le plus petit diviseur commun strictement positif entre  $a$  et  $b$ .

**solve**

La fonction **solve(f,a,b,n)** fait une résolution numérique de l'équation  $f(x) = 0$  dans l'intervalle  $[a; b]$ , celui-ci est subdivisé en  $n$  morceaux ( $n$  vaut 25 par défaut). La fonction renvoie une liste de résultats ou bien *nil*. La méthode utilisée est une variante de Newton.

**Exemple 1 :**

```
\begin{luacode}
resol = function(f,a,b)
  local y = 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  $\text{\solve{f1}{0}{math.pi/2}}\text{\par}$ 
La résolution de l'équation  $\cos(x)=x$  dans  $[\frac{\pi}{2};\pi]$  donne  $\text{\solve{f1}{math.pi/2}{math.pi}}\text{\par}$ 
La résolution de l'équation  $x^3-2x^2+\frac{1}{2}=0$  dans  $[-1;2]$  donne :  $\text{\solve{f2}{-1}{2}}\text{\par}$ .
```

**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 :

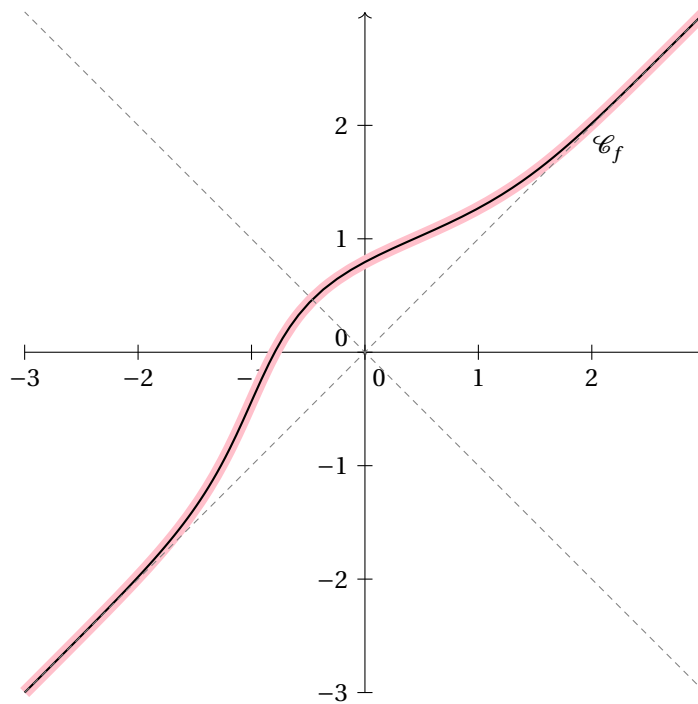
$$\forall x \in \mathbf{R}, \int_x^{f(x)} \exp(t^2) dt = 1.$$

On a deux méthodes possibles :

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

Dessignons les deux :

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

FIGURE 29 : Fonction  $f$  définie par  $\int_x^{f(x)} \exp(t^2) dt = 1$ .

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 **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 **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 **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 **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 **proj**( $L, d$ ) renvoie l'image de  $L$  par la projection orthogonale sur la droite  $d$ .

## 6) projO

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

## 7) rotate

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

## 9) simil

La fonction **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 0.

## 10) sym

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

## 11) symG

La fonction **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

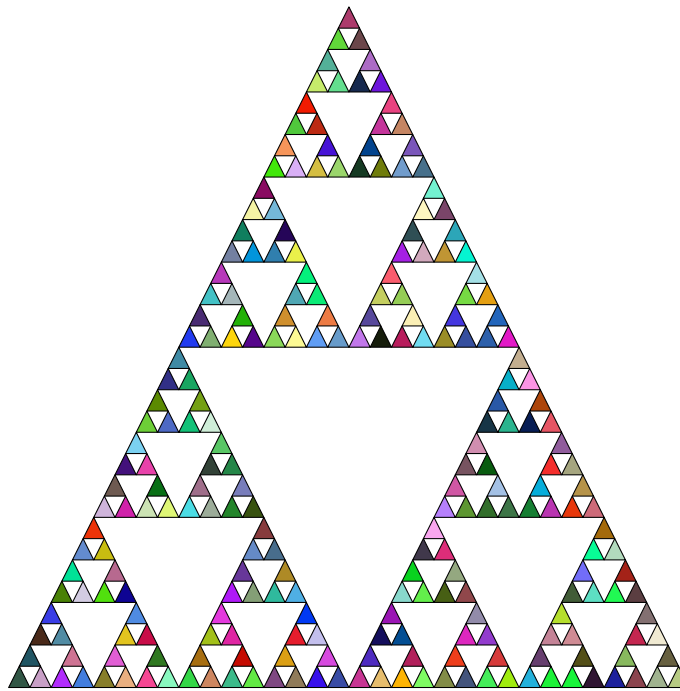
La fonction **symO(L,d)** renvoie l'image de  $L$  par la symétrie par rapport à la droite  $d$  et parallèlement au vecteur  $v$  (symétrie oblique).

```

1  \begin{luadraw}{name=Sierpinski}
2  local g = graph:new{window={-5,5,-5,5},size={10,10}}
3  local i = cpx.I
4  local rand = math.random
5  local A, B, C = 5*i, -5-5*i, 5-5*i -- triangle initial
6  local T, niv = {{A,B,C}}, 5
7  for k = 1, niv do
8      T = concat( hom(T,0.5,A), hom(T,0.5,B), hom(T,0.5,C) )
9  end
10 for _,cp in ipairs(T) do
11     g:Filloptions("full", rgb(rand(),rand(),rand()))
12     g:Dpolyline(cp,true)
13 end
14 g:Show()
15 \end{luadraw}

```

FIGURE 30 : Utilisation de transformations



## VI Calcul matriciel

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

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

où  $Lf$  désigne la partie linéaire de  $f$  (on a  $Lf(1) = f(1) - f(0)$  et  $Lf(i) = f(i) - f(0)$ ). La matrice identité est 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 **applymatrix(z,M)** applique la matrice  $M$  au complexe  $z$  et renvoie le résultat (ce qui revient à calculer  $f(z)$  si  $M$  est la matrice de  $f$ ). Lorsque  $z$  est le complexe *cpx.Jump* alors le résultat est *cpx.Jump*. Lorsque  $z$  est une chaîne de caractères alors la fonction renvoie  $z$ .
- La fonction **applyLmatrix(z,M)** applique la partie linéaire la matrice  $M$  au complexe  $z$  et renvoie le résultat (ce qui revient à calculer  $Lf(z)$  si  $M$  est la matrice de  $f$ ). Lorsque  $z$  est le complexe *cpx.Jump* alors le résultat est *cpx.Jump*.

#### composematrix

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

#### invmatrix

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

#### matrixof

- La fonction **matrixof(f)** calcule et renvoie la matrice de  $f$  (qui doit être une application affine du plan complexe).
- Exemple : `matrixof( function(z) return 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 **mtransform(L,M)** applique la matrice  $M$  à la liste  $L$  et renvoie le résultat.  $L$  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 inchangé (donc renvoyé tel quel).
- La fonction **mLtransform(L,M)** applique la partie linéaire la matrice  $M$  à la liste  $L$  et renvoie le résultat.  $L$  doit être une liste de complexes, si l'un d'eux est le complexe *cpx.Jump* alors il est inchangé.

**2) Matrice associée au graphe**

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

```
1 local g = 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  $M$  (avec  $M$  à droite) et le résultat est affecté à la matrice du graphe. L'argument  $M$  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  $L$  et renvoie le résultat, l'argument  $L$  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  $L$  et renvoie le résultat, l'argument  $L$  doit être une liste de complexes, ou une liste de listes de complexes.

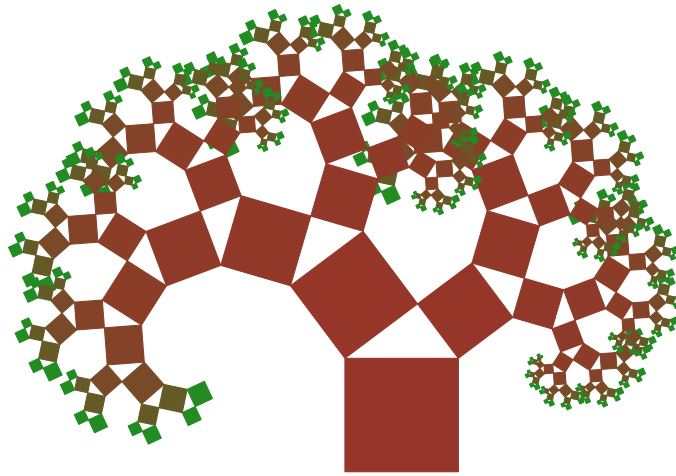
```
1 \begin{luadraw}{name=Pythagore}
2 local g = graph:new{window={-15,15,0,22},size={10,10}}
3 local a, b, c = 3, 4, 5 -- un triplet de Pythagore
4 local i, arccos, exp = cpx.I, math.acos, cpx.exp
5 local f1 = function(z)
6     return (z-c)*a/c*exp(-i*arccos(a/c))+c+i*c end
7 local M1 = matrixof(f1)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

```

20 end
21 arbre(8)
22 g:Show()
23 \end{luadraw}

```

FIGURE 31 : 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 *angle* (en degrés) et de centre *center*. L'argument *center* 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.

```

1 \begin{luadraw}{name=free_art}
2 local du = math.sqrt(2)/2
3 local g = graph:new{window={1-du,4+du,1-du,4+du},
4   margin={0,0,0,0},size={7,7}}
5 local i = cpx.I
6 g:Linestyle("noline")
7 g:Filloptions("full","Navy",0.1)
8 for X = 1, 4 do
9   for Y = 1, 4 do
10    g:Savematrix()
11    g:Shift(X+i*Y); g:Rotate(45)

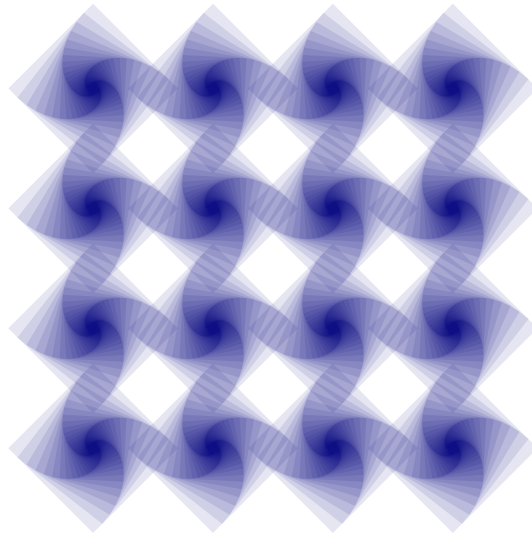
```

```

12   for k = 1, 25 do
13       g:Dsquare((1-i)/2, (1+i)/2, 1)
14       g:Rotate(7); g:Scale(0.9)
15   end
16   g:Restorematrix()
17 end
18 end
19 g:Show()
20 \end{luadraw}

```

FIGURE 32 : 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 :

```

1 local g = 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  $[xmin, xmax] \times [ymin, ymax]$  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 `x1`, `x2`, `y1`, `y2` 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 :

```

1 g:Saveattr()
2 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 `Saveattr()` doit correspondre une instruction `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:Coord-system(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  $[x1, x2] \times [y1, y2]$ , l'argument booléen facultatif `ortho` 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.

```

1 \begin{luadraw}{name=viewport_changewin}
2 local g = graph:new{window={-5,5,-5,5},size={10,10}}
3 local i = cpx.I

```

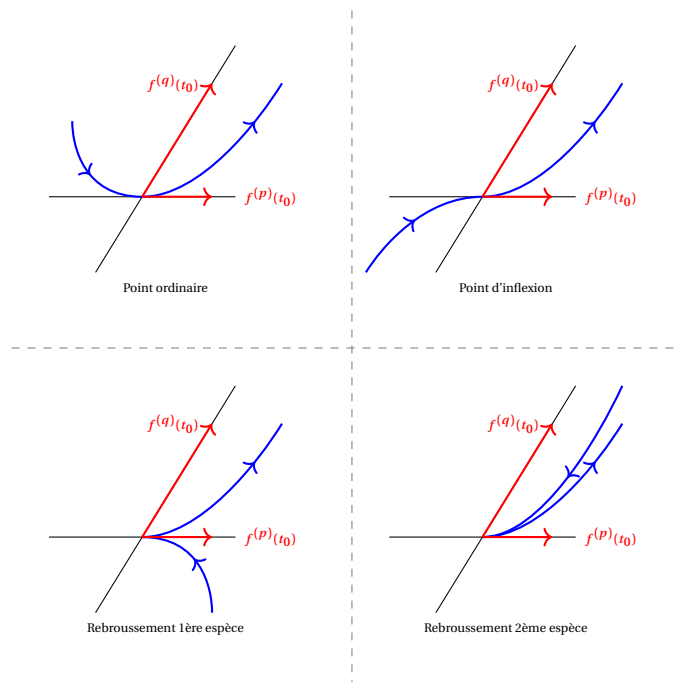


```

4 g:Labelsize("tiny")
5 g:Writeln("\tikzset{>/.style={decoration={markings, mark=at position #1 with {\arrow{>}}},
  ↳ postaction={decorate}}}")
6 g:Dline({0,1},"dashed,gray"); g:Dline({0,i},"dashed,gray")
7 local legende = {"Point ordinaire", "Point d'inflexion", "Rebroussement 1ère espèce", "Rebroussement 2ème espèce"}
8 local A, B, C = (1+i)*0.75, 0.75, 0
9 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}
10 local u = {Z(-5,0), Z(0,0), -5-5*i, -5*i}
11 for k = 1, 4 do
12   g:Saveatrr(); g:Viewport(u[k].re,u[k].re+5,u[k].im,u[k].im+5)
13   g:Coordsystem(-1.4,2.25,-1,1.25)
14   g:Composematrix({0,1,1+i}) -- pour pencher l'axe Oy
15   g:Dpolyline({{-1,1},{-i*0.5,i}}) -- axes
16   g:Lineoptions(nil,"blue",8)
17   g:Dpath({A2[k],(B2[k]+2*A2[k])/3,(C+5*B2[k])/6, C,"b"},">-=0.5")
18   g:Dpath({C,(C+5*B)/6,(B+2*A)/3,A,"b"},">-=0.75")
19   g:Dpolyline({{0,0.75},{0,0.75*i}},false,"->,red")
20   g:Dlabel(
21     legende[k],0.75-0.5*i, {pos="S"},
22     "$f^{(p)}(t_0)$",1,{pos="E",node_options="red"},
23     "$f^{(q)}(t_0)$",0.75*i,{pos="W",dist=0.05})
24   g:Restoreatrr()
25 end
26 g:Show()
27 \end{luadraw}

```

FIGURE 33 : Classification des points d'une courbe paramétrée



## VII Ajouter ses propres méthodes à la classe graph

Sans avoir à modifier les fichiers sources Lua associés au paquet *luadraw*, on peut ajouter ses propres méthodes à la classe *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 14, 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 (en enlevant le mot clé *local*), 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 *graph* deux nouvelles méthodes : une pour dessiner un champ de vecteurs d'une fonction  $f: (x,y) \rightarrow (x,y) \in \mathbf{R}^2$ , on la nommera *graph:Dvectorfield*, et une autre pour dessiner un champ de gradient d'une fonction  $f: (x,y) \rightarrow \mathbf{R}$ , on la nommera *graph:Dgradientfield*. Du coup nous appellerons ce fichier : *luadraw\_fields.lua*.

### Contenu du fichier :

```

1  -- luadraw_fields.lua
2  -- ajout de méthodes à la classe graph du paquet luadraw
3  -- pour dessiner des champs de vecteurs ou de gradient
4  function field(f,x1,x2,y1,y2,grid,long) -- fonction mathématique, indépendante du graphique
5  -- calcule un champ de vecteurs dans le pavé [x1,x2]x[y1,y2]
6  -- f fonction de deux variables à valeurs dans  $\mathbf{R}^2$ 
7  -- grid = {nbx, nby} : nombre de vecteurs suivant x et suivant y
8  -- long = longueur d'un vecteur
9      if grid == nil then grid = {25,25} end
10     local deltax, deltay = (x2-x1)/(grid[1]-1), (y2-y1)/(grid[2]-1) -- pas suivant x et y
11     if long == nil then long = math.min(deltax,deltay) end -- longueur par défaut
12     local vectors = {} -- contiendra la liste des vecteurs
13     local x, y, v = x1
14     for _ = 1, grid[1] do -- parcours suivant x
15         y = y1
16         for _ = 1, grid[2] do -- parcours suivant y
17             v = f(x,y) -- on suppose que v est bien défini
18             v = Z(v[1],v[2]) -- passage en complexe
19             if not cpx.isNul(v) then
20                 v = v/cpx.abs(v)*long -- normalisation de v
21                 table.insert(vectors, {Z(x,y), Z(x,y)+v}) -- on ajoute le vecteur
22             end
23             y = y+deltay
24         end
25         x = x+deltax
26     end
27     return vectors -- on renvoie le résultat (ligne polygonale)
28 end
29
30 function graph:Dvectorfield(f,args) -- ajout d'une méthode à la classe graph
31 -- dessine un champ de vecteurs
32 -- f fonction de deux variables à valeurs dans  $\mathbf{R}^2$ 
33 -- args table à 4 champs :
34 -- { view={x1,x2,y1,y2}, grid={nbx,nby}, long=, draw_options="" }
35     args = args or {}
36     local view = args.view or {self:Xinf(),self:Xsup(),self:Yinf(),self:Ysup()} -- repère utilisateur par défaut
37     local vectors = field(f,view[1],view[2],view[3],view[4],args.grid,args.long) -- calcul du champ
38     self:Dpolyline(vectors,false,args.draw_options) -- le dessin (ligne polygonale non fermée)
39 end
40
41 function graph:Dgradientfield(f,args) -- ajout d'une autre méthode à la classe graph
42 -- dessine un champ de gradient
43 -- f fonction de deux variables à valeurs dans  $\mathbf{R}$ 
44 -- args table à 4 champs :
45 -- { view={x1,x2,y1,y2}, grid={nbx,nby}, long=, draw_options="" }
46     local h = 1e-6
47     local grad_f = function(x,y) -- fonction gradient de f
48         return { (f(x+h,y)-f(x-h,y))/(2*h), (f(x,y+h)-f(x,y-h))/(2*h) }
49     end
50     self:Dvectorfield(grad_f,args) -- on utilise la méthode précédente

```

51 `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 *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 à chaque utilisation de `<chemin>`, et d'autre part l'instruction *dofile* ne vérifie pas 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*.

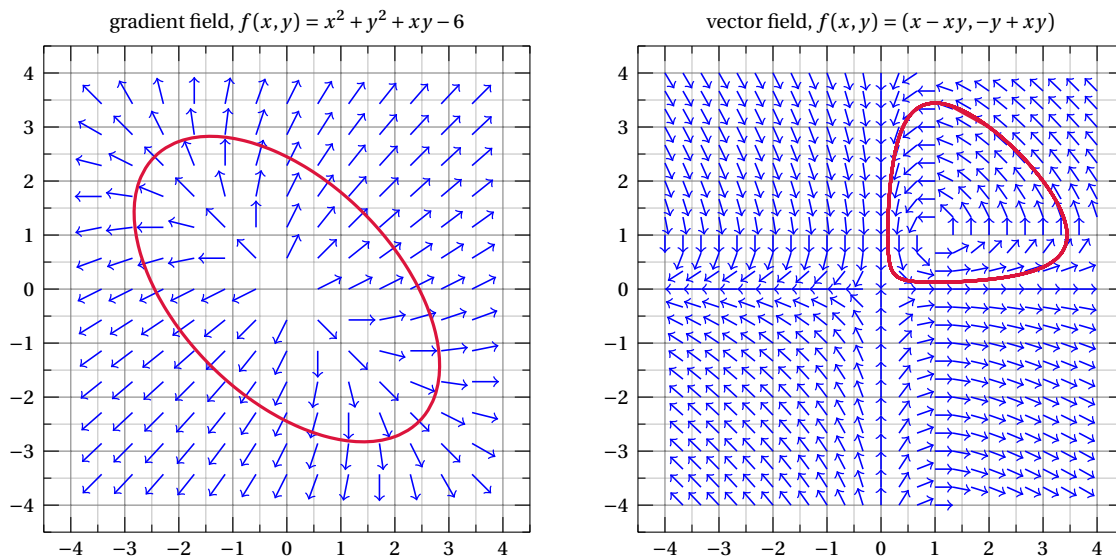
```
1 \begin{luadraw}{name=fields}
2 require "luadraw_fields" -- import des nouvelles méthodes
3 local g = graph:new{window={0,21,0,10},size={16,10}}
4 local i = cpx.I
5 g:Labelsize("footnotesize")
6 local f = function(x,y) return {x-x*y,-y+x*y} end -- Volterra
7 local F = function(x,y) return x^2+y^2+x*y-6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

```

23 g:Show()
24 \end{luadraw}

```

FIGURE 34 : Utilisation des nouvelles méthodes



### 3) Modifier une méthode existante

Prenons par exemple la méthode `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  $X$  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) :

```

1 function graph:DplotXY(X,Y,draw_options)
2 -- X est une liste de réels ou de chaînes
3 -- Y est une liste de réels de même longueur que X
4 local L = {} -- liste des points à dessiner
5 if type(X[1]) == "number" then -- liste de réels
6     for k,x in ipairs(X) do
7         table.insert(L,Z(x,Y[k]))
8     end
9 else
10    local noms = {} -- liste des labels à placer
11    for k = 1, #X do
12        table.insert(L,Z(k,Y[k]))
13        insert(noms,{X[k],k,{pos="E",node_options="rotate=-90"}})
14    end
15    self:Dlabel(table.unpack(noms)) --dessin des labels
16 end
17 self:Dpolyline(L,draw_options) -- dessin de la courbe
18 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 ...).

```

1 \begin{luadraw}{name=newDplotXY}
2 require "luadraw_fields" -- import de la méthode modifiée
3 local g = graph:new{window={-0.5,11,-1,20}, margin={0.5,0.5,0.5,1}, size={10,10,0}}
4 g:Labelsize("scriptsize")
5 local X, Y = {}, {} -- on définit deux listes X et Y, on pourrait aussi les lire dans un fichier
6 for k = 1, 10 do
7     table.insert(X,"nom"..k)
8     table.insert(Y,math.random(1,20))

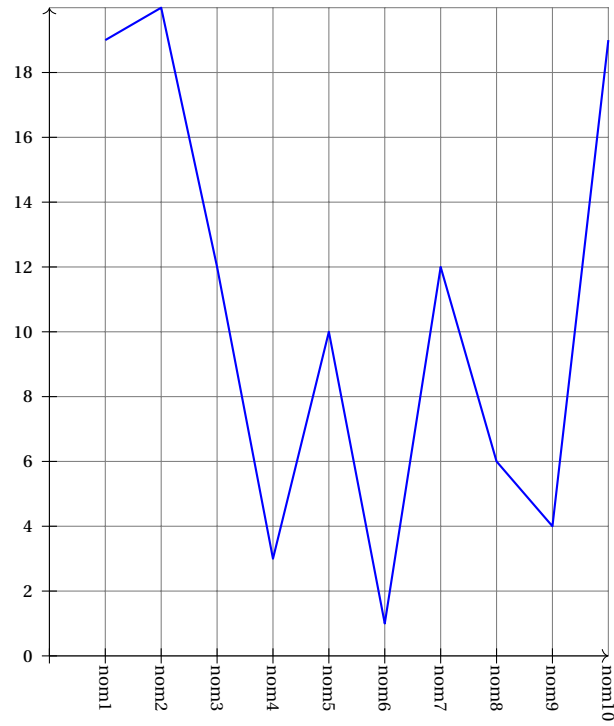
```

```

9  end
10 defaultlabelshift = 0
11 g:Daxes({0,1,2},{limits={{0,10},{0,20}}, labelpos={"none","left"},arrows="->", grid=true})
12 g:DplotXY(X,Y,"line width=0.8pt, blue")
13 g:Show()
14 \end{luadraw}

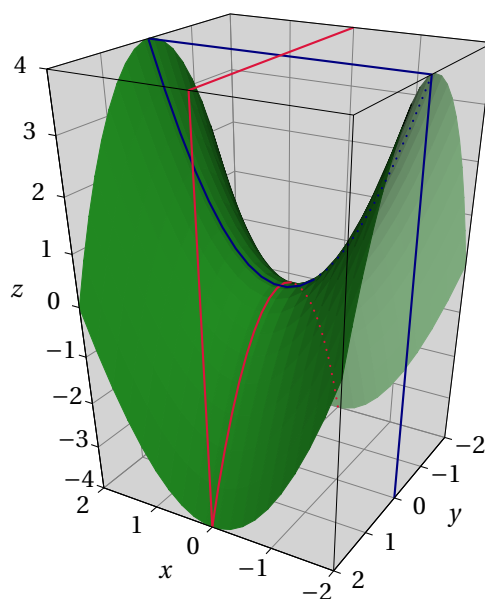
```

FIGURE 35 : Modification d'une méthode existante



## Dessin 3d

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



### I Introduction

#### 1) Prérequis

- Ce document 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 *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 *graph3d* (qui s'appuie sur la classe *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*
  - *plotmarks*

- *arrows.meta*
- *decorations.markings*
- 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 }
-- création d'un nouveau graphique en lui donnant un nom local
local g = graph3d:new{ window3d={x1,x2,y1,y2,z1,z2}, 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}
```

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 :

```
1 local g = graph3d:new{ window3d={x1,x2,y1,y2,z1,z2}, 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 *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).

- Le paramètre (facultatif) *window3d* définit le pavé de  $\mathbf{R}^3$  correspondant au graphique : c'est  $[x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$ . Par défaut c'est  $[-5, 5] \times [-5, 5] \times [-5, 5]$ .
- Le paramètre (facultatif) *adjust2d* indique si la fenêtre 2d qui va contenir la projection orthographique du dessin 3d, doit être déterminée automatiquement (false par défaut). Cette fenêtre 2d correspond à l'argument *window*.
- Le paramètre (facultatif) *viewdir* est une table qui définit les deux angles de vue (en degrés) utilisés pour la projection orthographique, qui est la projection par défaut (*viewdir={30,60}* par défaut). La figure suivante montre à quoi correspondent ces deux angles.

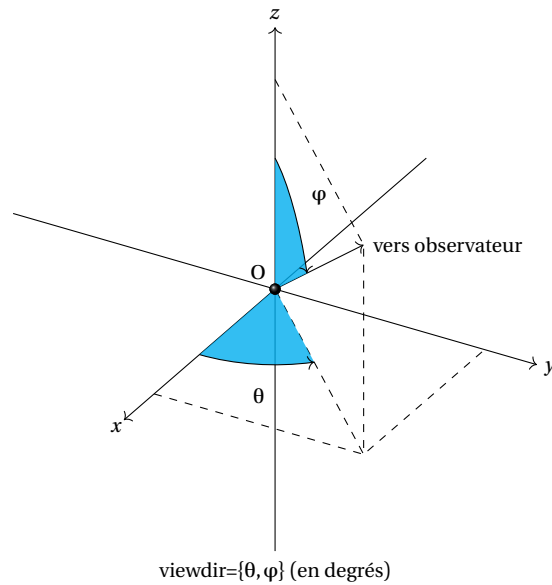


FIGURE 2 : Angles de vue

- Les autres paramètres sont ceux de la classe *graph*, ils ont été décrits dans le chapitre 1.

#### Construction du graphique.

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

#### 4) Modes de projection affine

Par défaut *luadraw* utilise la projection orthographique (projection orthogonale sur l'écran), celle-ci est définie par deux angles qui sont donnés à l'option *viewdir* lors de la création, ou bien à la méthode *g:Setviewdir()*.

Il y a trois autres modes de projection affine possible, mais qui ne sont pas des projections orthogonales :

- trois perspectives cavalières : sur le plan *yz*, ou sur le plan *xz*, ou sur le plan *xy*. Celles-ci 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.
- une perspective isométrique.

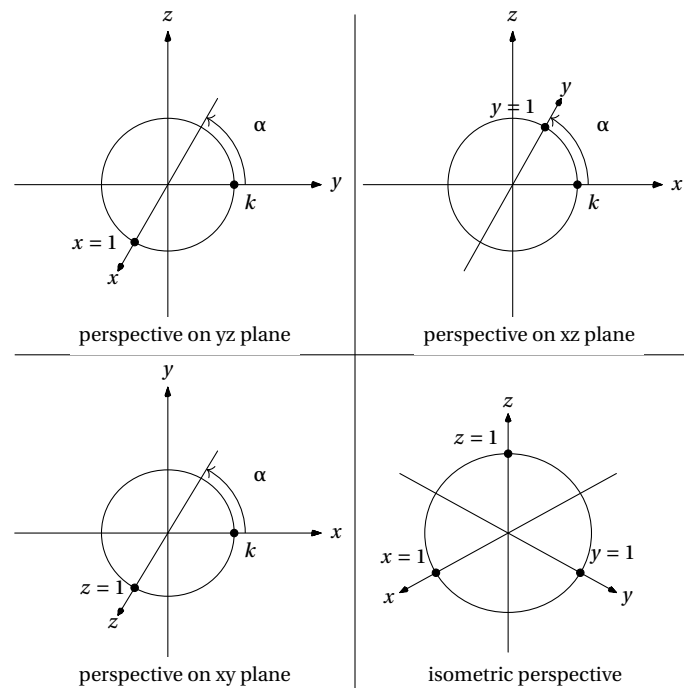


FIGURE 3 : Modes de projection affine



Ces modes de projection sont accessibles par la fonction **perspective(mode,k,alpha)**, où l'argument *mode* peut valoir "yz", ou "xz" ou "xy" (pour les trois perspectives cavalières) ou "iso" pour la perspective isométrique, si *mode* a une valeur non reconnue alors c'est la projection orthographique qui est sélectionnée. Pour les trois premiers modes, on donne également les valeurs des paramètres *k* (0.5 par défaut) et *alpha* (45 par défaut), ces valeurs sont inutiles pour le quatrième mode.

Cette fonction s'utilise soit avec l'option *viewdir* à la création de l'objet graphique, par exemple :

```
1 local g = graph3d:new{ viewdir = perspective("yz",0.65,60) }
```

ou bien pendant la création du graphique avec la méthode *g:Setviewdir()* :

```
1 g:Setviewdir(perspective("yz",0.65,60))
```

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

Le plan passant par la *target* et orthogonal à l'axe *target - camera* est le plan de la projection, il représente l'écran. Comme pour les modes précédents, la projection centrale est accessible par la fonction *perspective* :

**perspective("central",camera,target),**

ou bien

**perspective("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 :

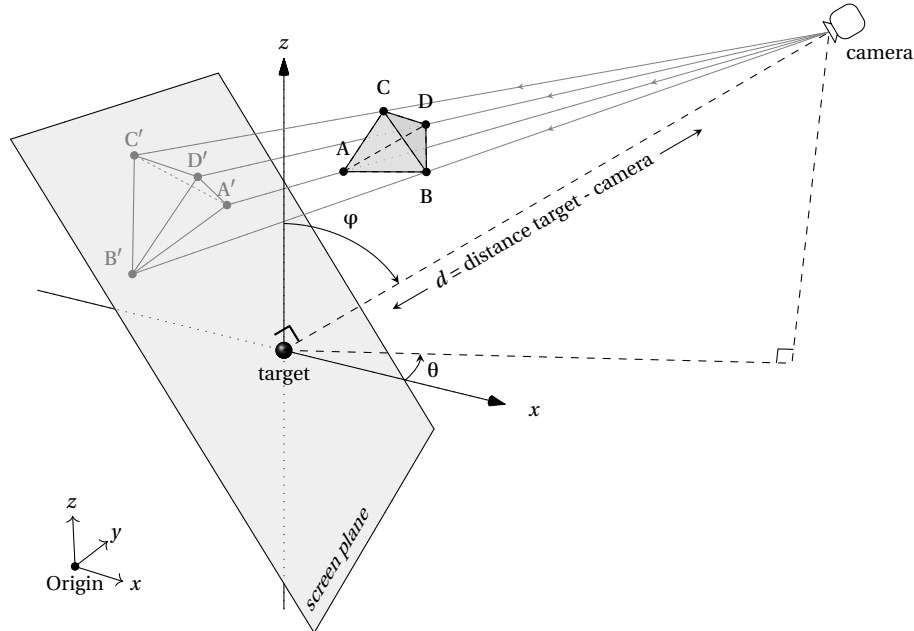


FIGURE 4 : Projection centrale

Les valeurs par défaut sont : *theta*=30 (degrés), *phi*=60, *d*=15, *target*=Origin. Cette fonction s'utilise soit avec l'option *viewdir* à la création de l'objet graphique, par exemple :

```
1 local g = graph3d:new{ viewdir = perspective("central",40,60) }
```

ou bien pendant la création du graphique avec la méthode *g:Setviewdir()* :

```
1 g:Setviewdir(perspective("central",40,60))
```

## 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 (appelés points 3d). Quatre triplets portent un nom spécifique (variables prédéfinies), il s'agit de :
  - **Origin**, qui représente le triplet (0, 0, 0).
  - **vecI**, qui représente le triplet (1, 0, 0).
  - **vecJ**, qui représente le triplet (0, 1, 0).
  - **vecK**, qui représente le triplet (0, 0, 1).

À cela s'ajoute la variable **ID3d** qui est la table  $\{Origin, vecI, vecJ, vecK\}$  représentant la matrice unité 3d. Par défaut c'est la matrice de transformation du graphe 3d.

- La classe *pt3d* (qui est automatiquement chargée) définit les triplets de réels, les opérations possibles, et un certain nombre de méthodes. Pour créer un point 3d, il y a trois méthodes :
  - Définition en cartésien : la fonction **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 **Mc(r,θ,z)** (angle exprimé en radians) renvoie le triplet  $(r \cos(\theta), r \sin(\theta), z)$ .
  - Définition en sphérique : la fonction **Ms(r,θ,φ)** 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 **isPoint3d()** qui renvoie un booléen.

Conversion : pour convertir un réel ou un complexe en point 3d, on dispose de la fonction **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  $M(kx,ky,kz)$ .
- 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  $u = M(x, y, z)$ , 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.

### 4) Fonctions mathématiques

Dans le fichier définissant la classe *pt3d*, quelques fonctions mathématiques sont introduites :

- **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).
- **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.
- **polyline2path3d(L)** : cette fonction renvoie *L* qui est une liste de points 3d ou une liste de listes de points 3d, sous la forme d'un chemin (que l'on peut dessiner avec la méthode *g:Dpath3d()*).

## 5) Afficher une variable dans le terminal

L'instruction **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 appliqué 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 relais.

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 n'est pas une transformation 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).

## 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

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.

### Droite : Dline3d

La méthode **g:Dline3d(d,draw\_options,clip)** trace la droite  $d$ , celle-ci est une liste du type  $\{A,u\}$  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 **arc3d(B,A,C,r,sens,normal)** renvoie la liste des points de cet arc (ligne polygonale 3d).
- La fonction **arc3db(B,A,C,r,sens,normal)** renvoie cet arc sous forme d'un chemin 3d (voir Dpath3d) 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 **circle3d(I,R,normal)** renvoie la liste des points de ce cercle (ligne polygonale 3d).
- La fonction **circle3db(I,R,normal)** renvoie ce cercle sous forme d'un chemin 3d (voir Dpath3d) 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 de caractères qui sera passée directement 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  $p1$ .
- `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  $c1$  et  $c2$  sont les deux points (3d) de contrôle.
- `p1,n,"c"` (cercle), dessine le cercle de centre  $p1$  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  $p1$ , de rayon  $r$ , allant de *last* vers  $p2$ , 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*,  $p1$  et  $p2$  sont alignés (et dans ce cas, le 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.

```

1 \begin{luadraw}{name=viewdir}
2 local g = graph3d:new{ size={8,8} }
3 local i = cpx.I
4 local O, A = Origin, M(4,4,4)
5 local B, C, D, E = pxy(A), px(A), py(A), pz(A) --projeté de A sur le plan xOy et sur les axes
6 g:Dpolyline3d( {{O,A},{-5*vecI,5*vecI},{-5*vecJ,5*vecJ},{-5*vecK,5*vecK}}, "->" -- axes
7 g:Dpolyline3d( {{E,A,B,O}}, {C,B,D}}, "dashed")
8 g:Dpath3d( {C,O,B,2.5,1,"ca",0,"l","cl"}, "draw=none,fill=cyan,fill opacity=0.8") --secteur angulaire
9 g:Darc3d(C,O,B,2.5,1,"->") -- arc de cercle pour theta
10 g:Dpath3d( {E,O,A,2.5,1,"ca",0,"l","cl"}, "draw=none,fill=cyan,fill opacity=0.8") --secteur angulaire
11 g:Darc3d(E,O,A,2.5,1,"->") -- arc de cercle pour phi
12 g:Dballdots3d(O) -- le point origine sous forme d'une petite sphère
13 g:Labelsize("footnotesize")
14 g:Dlabel3d(
15     "$x$", 5.25*vecI,{}, "$y$", 5.25*vecJ,{}, "$z$", 5.25*vecK,{},
16     "vers observateur", A, {pos="E"},
17     "$O$", O, {pos="NW"},
18     "$\\theta$", (B+C)/2, {pos="N", dist=0.15},
19     "$\\varphi$", (A+E)/2, {pos="S",dist=0.25}
20 )
21 g:Dlabel("viewdir=\\{$\\theta$,\\varphi$\\} (en degrés)",-5*i,{pos="N"}) -- label 2d
22 g:Show()
23 \end{luadraw}

```

### 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  $P$ ,  $L_1$  et  $L_2$  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 : *top* (=8), *right* (=4), *bottom* (=2), *left* (=1) et *all* (=15), que l'on peut ajouter entre elles, par exemple :

- mode = bottom+left : pour les côtés bas et gauche
- mode = top+right+bottom : pour les côtés haut, droit et bas
- etc

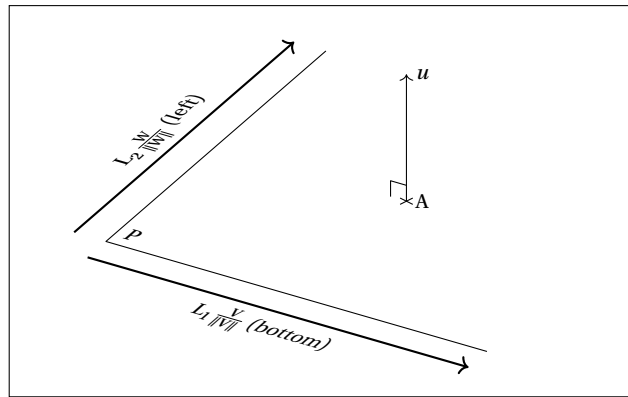
Par défaut le mode vaut *all* ce qui correspond à *top+right+bottom+left*.

```

1 \begin{luadraw}{name=Dplane}
2 local g = graph3d:new{size={8,8},window={-5.25,3,-2.5,2.5},margin={0,0,0,0},border=true}
3 local i = cpx.I
4 g:Labelsize("footnotesize")
5 local A = Origin
6 local P = {A, vecK}
7 g:Dplane(P, vecJ, 6, 6, left+bottom)
8 g:Dcrossdots3d({A,vecK},nil,0.75)
9 g:Dseg3d({A,A+2*vecK},"->")
10 g:Dangle3d(-vecJ,A,vecK,0.25)
11 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")
12 g:Dlabel3d("$A$",A,{pos="E"},
13     "$u$",2*vecK,{},
14     "$P$", M(3,-3,0),{pos="NE", dir={vecJ,-vecI}},
15     "$L_1\\frac{V}{\\|V\\|}$ (bottom)", M(3.5,0,0), {pos="S"},
16     "$L_2\\frac{W}{\\|W\\|}$ (left)", M(0,-3.5,0), {pos="N",dir={-vecI,-vecJ}}
17 )
18 g:Show()
19 \end{luadraw}

```

FIGURE 5 : 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

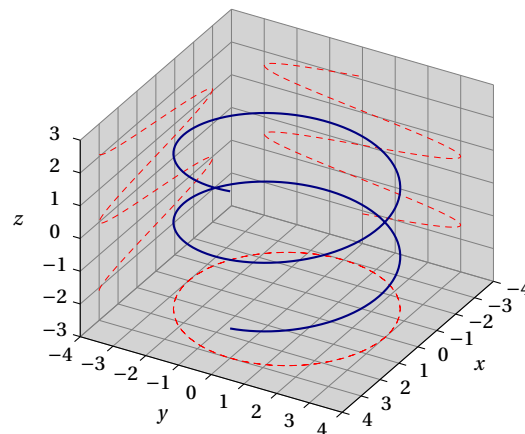
- La fonction **parametric3d(p,t1,t2,nbdots,discont,nbdiv)** fait le calcul des points de la courbe et renvoie une ligne polygonale 3d (pas de dessin).
  - L'argument  $p$  est le paramétrage, ce doit être une fonction d'une variable réelle  $t$  et à valeurs dans  $\mathbf{R}^3$  (les images sont des points 3d), par exemple : `local p = function(t) return Mc(3,t,t/3) end`
  - Les arguments  $t1$  et  $t2$  sont obligatoires avec  $t1 < t2$ , 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:Dparametric3d(p,args)** fait le calcul des points et le dessin de la courbe paramétrée par  $p$ . Le paramètre  $args$  est une table à 6 champs :

```
{ t={t1,t2}, nbdots=40, discont=true/false, clip=true/false, nbdiv=5, draw_options="" }
```

- Par défaut, le champ  $t$  est égal à  $\{g:Xinf(),g:Xsup()\}$ ,
- le champ  $nbdots$  vaut 40,
- le champ  $discont$  vaut *false*,
- le champ  $nbdiv$  vaut 5,
- le champ  $clip$  vaut *false*, il indique si la courbe doit être clippée avec la fenêtre 3d courante.
- le champ  $draw\_options$  est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction `\draw`).

```
1 \begin{luadraw}{name=Dparametric3d}
2 local g = graph3d:new{window3d={-4,4,-4,4,-3,3}, window={-7.5,6.5,-7,6}, size={8,8}}
3 local pi = math.pi
4 g:Labelsize("footnotesize")
5 local p = function(t) return Mc(3,t,t/3) end
6 local L = parametric3d(p,-2*pi,2*pi,25,false,2)
7 g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
8 g:Lineoptions("dashed","red",2)
9 -- projection sur le plan y=-4
10 g:Dpolyline3d(proj3d(L,{M(0,-4,0),vecJ}))
11 -- projection sur le plan x=-4
12 g:Dpolyline3d(proj3d(L,{M(-4,0,0),vecI}))
13 -- projection sur le plan z=-3
14 g:Dpolyline3d(proj3d(L,{M(0,0,-3),vecK}))
15 -- dessin de la courbe
16 g:Lineoptions("solid","Navy",8)
17 g:Dparametric3d(p,{t={-2*pi,2*pi}})
18 g:Show()
19 \end{luadraw}
```

FIGURE 6 : 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 **curvilinear\_param3d(L,close)** renvoie une fonction d'une variable  $t \in [0;1]$  et à valeurs sur la ligne  $L$  (points 3d), la valeur en  $t = 0$  est le premier point de  $L$ , et la valeur en  $t = 1$  est le dernier point; cette fonction est suivie d'un nombre qui représente la longueur totale de  $L$ . L'argument optionnel *close* indique si la ligne  $L$  doit être refermée (*false* par défaut).

### Le repère : **Dboxaxes3d**

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

- **xaxe=true/false**, **yaxe=true/false** et **zaxe=true/false** : indique si les axes correspondant doivent être dessinés ou non (true par défaut).
- **drawbox=true/false** : indique si une boîte doit être dessinée avec les axes (false par défaut).
- **grid=true/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 par défaut) indique l'épaisseur de trait de la grille en dixième de point.
  - **gridcolor** ("black" par défaut) indique la couleur de la grille.
  - **fillcolor** ("") par défaut permet de peindre ou non le fond des grilles.
- **xlimits={x1,x2}**, **ylimits={y1,y2}**, **zlimits={z1,z2}** : permet de définir les trois intervalles utilisés pour les longueurs des 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** : indique le pas des graduations sur les trois axes (1 par défaut).
- **xstep**, **ystep**, **zstep** : indique le pas des graduations sur chaque axe (valeur de **xyzstep** par défaut).
- **xyzticks** (0.2 par défaut) : indique la longueur des graduations.
- **labels** (true par défaut) : indique si la valeur des graduations doit être affichée ou non.
- **xlabelsep**, **ylabelsep**, **zlabelsep** : indique la distance entre les labels et les graduations (0.25 par défaut).
- **xlabelstyle**, **ylabelstyle**, **zlabelstyle** : 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\$" par défaut), **ylegend** ("y\$" par défaut), **zlegend** ("z\$" par défaut) : permet de définir une légende pour les axes.
- **xlegendsep**, **ylegendsep**, **zlegendsep** : indique la distance entre les legendes et les graduations (0.5 par défaut).

## 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 :

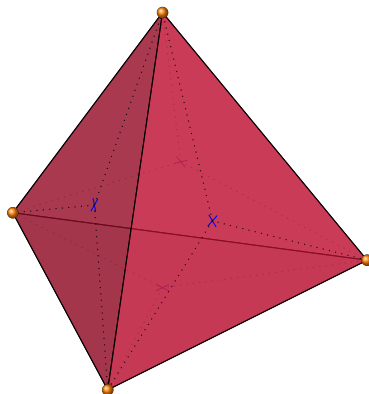
- La méthode **g:Ddots3d(L, mark\_options, clip)**. Le principe est le même que dans la version 2d, les points sont dessinés dans la couleur courante du tracé de lignes avec le style courant. L'argument *mark\_options* est une chaîne de caractères facultative qui sera passée telle quelle à l'instruction *\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).

```

1 \begin{luadraw}{name=Ddots3d}
2 local g = graph3d:new{viewdir={15,60},bbox=false,size={8,8}}
3 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)
4 local u, v, w = B-A, C-A, D-A
5 -- centres de gravité faces cachées
6 for _, F in ipairs({{A,B,C},{B,C,D}}) do
7   local G, u = isobar3d(F), pt3d.prod(F[2]-F[1],F[3]-F[1])
8   g:Dcrossdots3d({G,u}, "blue", 0.75)
9   g:Dpolyline3d({{F[1],G,F[2]},{G,F[3]}}, "dotted")
10 end
11 -- dessin du tétraèdre construit sur A, B, C et D
12 g:Dpoly(tetra(A,u,v,w), {mode=mShaded,opacity=0.7,color="Crimson"})
13 -- centres de gravité faces visibles
14 for _, F in ipairs({{A,B,D},{A,C,D}}) do
15   local G, u = isobar3d(F), pt3d.prod(F[2]-F[1],F[3]-F[1])
16   g:Dcrossdots3d({G,u}, "blue", 0.75)
17   g:Dpolyline3d({{F[1],G,F[2]},{G,F[3]}}, "dotted")
18 end
19 g:Dballdots3d({A,B,C,D}, "orange") --sommets
20 g:Show()
21 \end{luadraw}

```

FIGURE 7 : Un tétraèdre et les centres de gravité de chaque face



### Labels 3d : Dlabel3d

La méthode pour placer un label dans l'espace est :

**g:Dlabel3d(text1, anchor1, args1, text2, anchor2, args2, ...).**

- 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 *args1*, *args2*,... permettent de définir localement les paramètres des labels, ce sont des tables à 4 champs :

```
{ pos=nil, dist=0, dir={dirX,dirY,dep}, node_options="" }
```

- Le champ *pos* 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.
- Le champ *dist* est une distance en cm (dans le plan de l'écran) qui vaut 0 par défaut, c'est la distance entre le label et son point d'ancrage lorsque *pos* n'est pas égal à *center*.
- *dir*=*{dirX,dirY,dep}* est la direction de l'écriture dans l'espace (*nil*, valeur par défaut, pour le sens par défaut). 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.
- L'argument *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'argument *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)

#### Cylindre : Dcylinder

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

- Ancienne syntaxe : **g:Dcylinder(A,V,r,args)** dessine un cylindre droit, où *A* est un point 3d représentant le centre d'une des faces circulaires, *V* est un point 3d, c'est un vecteur représentant l'axe du cône, le centre de la face circulaire opposée est le point *A + V* (cette face est orthogonale à *V*), et *r* est le rayon de la base circulaire.
- La syntaxe : **g:Dcylinder(A,r,B,args)** dessine un cylindre droit, où *A* est un point 3d représentant le centre d'une des faces circulaires, *B* est le centre de la face opposée, et *r* 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,args)**, où *A* est un point 3d représentant le centre d'une des faces circulaires, *B* est le centre de la face circulaire opposée, *r* est le rayon, et *V* est un vecteur 3d non nul orthogonal au plan des faces circulaires.

Pour les trois syntaxes, *args* est une table dont les champs définissent les options de tracé. Ces options sont :

- *mode=mWireframe* ou *mGrid* (*mWireframe* par défaut). En mode *mWireframe* c'est un dessin en fil de fer, en mode *mGrid* c'est un dessin en grille (comme s'il y avait des facettes).
- *hiddenstyle*, 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 *Hiddenlinestyle* qui est elle même initialisée avec la valeur "dotted".
- *hiddencolor*, définit la couleur des lignes cachées (égale à *edgecolor* par défaut).
- *edgecolor*, définit la couleur des lignes (couleur courante par défaut).
- *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 un gradient linéaire.
- *opacity=1*, définit la transparence du dessin.
- Paramètres pour le gradient : lorsqu'il y a un remplissage avec la couleur, 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 utilisée, 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*, sa valeur par défaut est *{50,10,100}*.
- Pour la section du cylindre, c'est l'option *gradsection*, sa valeur par défaut est *{25,18,50}*.

#### Cône : Dcone

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

- Ancienne syntaxe : **g:Dcone(A,V,r,args)** 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,args)** 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,args)**, 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, *args* est une table dont les champs définissent les options de tracé. Ces options sont :

- *mode=mWireframe* ou *mGrid* (*mWireframe* par défaut). En mode *mWireframe* c'est un dessin en fil de fer, en mode *mGrid* c'est un dessin en grille (comme s'il y avait des facettes).
- *hiddenstyle*, 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 *Hiddenlinestyle* qui est elle même initialisée avec la valeur "dotted".
- *hiddencolor*, définit la couleur des lignes cachées (égale à *edgecolor* par défaut).
- *edgecolor*, définit la couleur des lignes (couleur courante par défaut).
- *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 un gradient linéaire.
- *opacity=1*, définit la transparence du dessin.
- Paramètres pour le gradient : lorsqu'il y a un remplissage avec la couleur, un gradient linéaire est utilisé pour le côté du cône 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 utilisée, 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 cône, c'est l'option *gradside*, sa valeur par défaut est {50,10,100}.
- Pour la section du cône, c'est l'option *gradsection*, sa valeur par défaut est {25,18,50}.

### Tronc de cône : Dfrustum

Dessiner un tronc de cône à base circulaire (droit ou penché). Deux syntaxes possibles : La méthode **g:Dfrustum(A,V,R,r,args)** dessine un tronc de cône à bases circulaires.

- La syntaxe : **g:Dfrustum(A,R,r,V,args)** pour un tronc de cône droit,  $A$  est un point 3d représentant le centre de la face de rayon  $R$ ,  $V$  est un vecteur 3d représentant l'axe du tronc de cône, le centre de la deuxième face circulaire est le point  $A + V$ , et son rayon est  $r$ , (les faces sont orthogonales à  $V$ ). Lorsque  $R = r$  on a simplement un cylindre.
- La syntaxe : **g:Dfrustum(A,R,r,V,B,args)** 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.

Dans les deux cas, *args* est une table dont les champs définissent les options de tracé. Ces options sont :

- *mode=mWireframe* ou *mGrid* (*mWireframe* par défaut). En mode *mWireframe* c'est un dessin en fil de fer, en mode *mGrid* c'est un dessin en grille (comme s'il y avait des facettes).
- *hiddenstyle*, 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 *Hiddenlinestyle* qui est elle même initialisée avec la valeur "dotted".
- *hiddencolor*, définit la couleur des lignes cachées (égale à *edgecolor* par défaut).
- *edgecolor*, définit la couleur des lignes (couleur courante par défaut).
- *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 un gradient linéaire.
- *opacity=1*, définit la transparence du dessin.
- Paramètres pour le gradient : lorsqu'il y a un remplissage avec la couleur, un gradient linéaire est utilisé pour le côté du cône 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 utilisée, 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 cône, c'est l'option *gradside*, sa valeur par défaut est {50,10,100}.

- Pour la section du cône, c'est l'option *gradsection*, sa valeur par défaut est  $\{25,18,50\}$ .

### Sphère : Dsphere

La méthode **g:Dsphere(A,r,args)** dessine une sphère

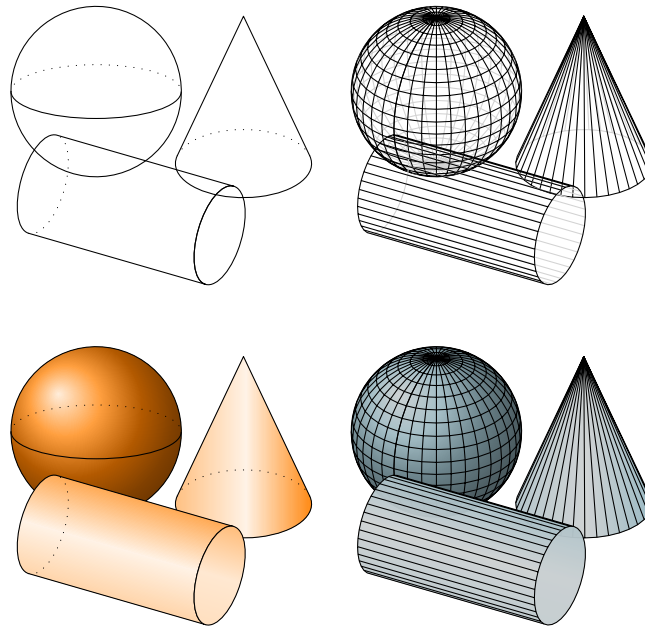
- $A$  est un point 3d représentant le centre de la sphère.
- $r$  est le rayon de la pshère
- $args$  est une table à 5 champs pour définir les options de tracé. Ces options sont :
  - *mode*=*mWireframe* ou *mGrid* ou *mBorder* (*mWireframe* par défaut). En mode *mWireframe* on dessine le contour (cercle) et l'équateur, en mode *mGrid* c'est le contour avec méridiens et fuseaux (grille), et en mode *mBorder* c'est le contour uniquement.
  - *hiddenstyle*, 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 *Hiddenlinestyle* qui est elle même initialisée avec la valeur "dotted".
  - *hiddencolor*, définit la couleur des lignes cachées (égale à *edgecolor* par défaut).
  - *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.
  - *edgestyle*, définit le style de ligne pour les arêtes visibles, par défaut c'est le style courant.
  - *edgecolor*, définit la couleur des arêtes visibles (couleur courante par défaut).
  - *edgewidth*, définit l'épaisseur des des arêtes visible en dixième de point (épaisseur courante par défaut).

```

1  \begin{luadraw}{name=cylindre_cone_sphere}
2  local g = graph3d:new{ size={10,10} }
3  local dessin = function(args)
4      g:Dsphere(M(-1,-2.5,1),2.5, args)
5      g:Dcone(M(-1,2.5,5),-5*vecK,2, args)
6      g:Dcylinder(M(3,-2,0),6*vecJ,1.5, args)
7  end
8  -- en haut à gauche, options par défaut
9  g:Saveattr(); g:Viewport(-5,0,0,5); g:Coordsystem(-5,5,-5,5,true); dessin(); g:Restoreattr()
10 g:Saveattr(); g:Viewport(0,5,0,5); g:Coordsystem(-5,5,-5,5,true) -- en haut à droite
11 dessin({mode=mGrid, hiddenstyle="solid", hiddencolor="LightGray"}); g:Restoreattr()
12 g:Saveattr(); g:Viewport(-5,0,-5,0); g:Coordsystem(-5,5,-5,5,true) -- en bas à gauche
13 dessin({mode=Border, color="orange"}); g:Restoreattr()
14 g:Saveattr(); g:Viewport(0,5,-5,0); g:Coordsystem(-5,5,-5,5,true) -- en bas à droite
15 dessin({mode=mGrid,opacity=0.8,hiddenstyle="noline",color="LightBlue"}); g:Restoreattr()
16 g:Show()
17 \end{luadraw}

```

FIGURE 8 : Cylindres, cônes et sphères



## 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. 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.

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 :  $T = \{\text{vertices} = \{A, B, C, D\}, \text{facets} = \{\{1, 2, 4\}, \{2, 3, 4\}, \{3, 1, 4\}, \{1, 3, 2\}\}\}$ .

#### Fonctions de conversion entre les deux définitions

- La fonction **poly2facet(P)** où P est un polyèdre, renvoie ce solide sous forme d'une liste de facettes.
- La fonction **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.

### 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 :

- **mode=** : définit le mode de représentation.

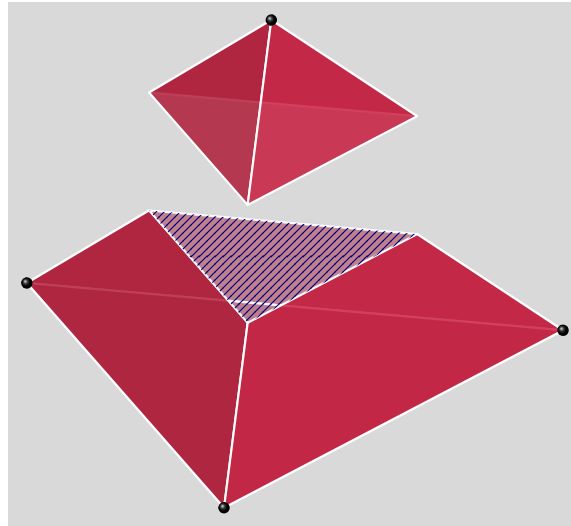
- `mode=mWireframe` : mode fil de fer, on dessine les arêtes visibles et cachées.
- `mode=mFlat` : on dessine les faces de couleur unie, ainsi que les arêtes visibles.
- `mode=mFlatHidden` : on dessine les faces de couleur unie, les arêtes visibles, et les arêtes cachées.
- `mode=mShaded` : on dessine les faces de couleur nuancée en fonction de leur inclinaison, ainsi que les arêtes visibles. C'est le mode par défaut.
- `mode=mShadedHidden` : on dessine les faces de couleur nuancée en fonction de leur inclinaison, les arêtes visibles et cachées.
- `mode=mShadedOnly` : on dessine les faces de couleur nuancée en fonction de leur inclinaison, mais pas les arêtes.
- `contrast` : c'est un nombre qui vaut 1 par défaut. Ce nombre permet d'accentuer ou diminuer la nuance des couleurs des facettes dans les modes `mShaded`, `mShadedHidden`, `mShadedOnly`.
- `edgestyle` : est une chaîne qui définit le style de ligne des arêtes. C'est le style courant par défaut.
- `edgecolor` : est une chaîne qui définit la couleur des arêtes. C'est la couleur courante des lignes par défaut.
- `hiddenstyle` : est une chaîne qui définit le style de ligne des arêtes cachées. Par défaut c'est la valeur contenue dans la variable globale `Hiddenlinestyle` (qui vaut elle-même "dotted" par défaut).
- `hiddencolor` : est une chaîne qui définit la couleur des arêtes cachées. C'est la couleur courante des lignes par défaut.
- `edgewidth` : épaisseur de trait des arêtes en dixième de point. C'est l'épaisseur courante par défaut.
- `opacity` : 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` : booléen qui vaut false par défaut. Lorsqu'il a 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` : 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.
- `color` : chaîne définissant la couleur de remplissage des facettes, c'est "white" par défaut.
- `usepalette` (*nil* par défaut), 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}`, 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, et *mode* qui est une chaîne qui peut être soit "x", soit "y", ou soit "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.

```

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

```

FIGURE 9 : Section d'un tétraèdre par un plan



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

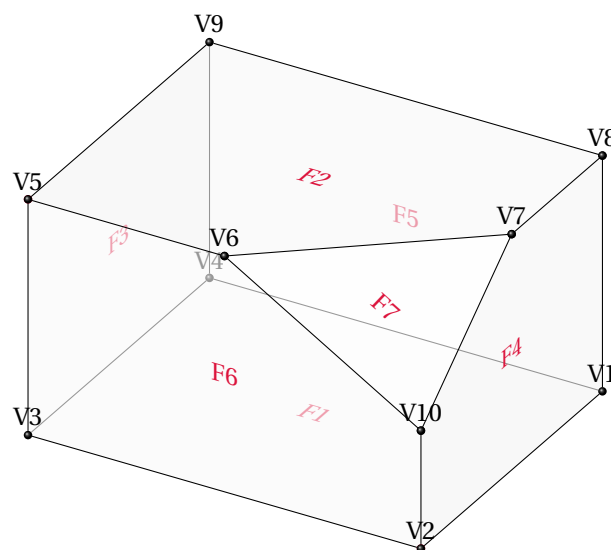
La méthode **g:Dpolynames(Poption,opacity)** affiche le polyèdre  $P$  avec la possibilité d'ajouter sur chaque face son numéro (qui est son rang d'apparition dans la liste  $P_{\text{facets}}$ ) précédé de la lettre  $F$ , et éventuellement le numéro de chaque sommet (qui est son rang d'apparition dans la liste  $P_{\text{vertices}}$ ) 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.

```

1 \begin{luadraw}{name=show_facet_number}
2 local g = graph3d:new{viewdir={30,60}, window={-2.5,5,-3,3},size={10,10}}
3 P = parallelep(Origin, 4*vecI,5*vecJ,3*vecK)
4 local A, B, C = M(4,2.5,3), M(2,5,3), M(4,5,1.5)
5 P = cutpoly(P, plane(A,B,C), true) -- we cut P with the plane, and add a facet in place of the section
6 g:Dpolynames(P) -- we want to see facets and vertices numbers of P
7 g:Show()
8 \end{luadraw}

```

FIGURE 10 : 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*.

- **tetra(S,v1,v2,v3)** renvoie le tétraèdre de sommets S (point 3d),  $S + v1$ ,  $S + v2$ ,  $S + v3$ . Les trois vecteurs  $v1$ ,  $v2$ ,  $v3$  (points 3d) sont supposés dans le sens direct.
- **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.
- **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*.
- **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.
- **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).
- **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.
- **cylinder(A,V,R,nbfacet,open)** renvoie un cylindre de rayon R, d'axe {A,V} où A est un point 3d, centre d'une des bases circulaires et V 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.
- **cylinder(A,R,B,nbfacet,open)** renvoie un cylindre de rayon R, d'axe (AB) où A est un point 3d, centre d'une des bases circulaires et B le centre de la seconde base. Le cylindre est droit. 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.
- **cylinder(A,R,V,B,nbfacet,open)** renvoie un cylindre de rayon R, d'axe (A) où A est un point 3d, centre d'une des bases circulaires, 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.
- **cone(A,V,R,nbfacet,open)** renvoie un cône de sommet A (point 3d), d'axe {A,V}, de base circulaire le cercle de centre  $A + V$  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.
- **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.
- **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.
- **frustum(C,R,r,V,nbfacet,open)** renvoie un tronc de cône droit. Le point C (point 3d) est le centre de la base circulaire de rayon R, le vecteur V dirige l'axe du tronc de cône. Le centre de l'autre base circulaire est le point  $C + V$ , et son rayon est *r* (les bases sont orthogonales à V). 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.

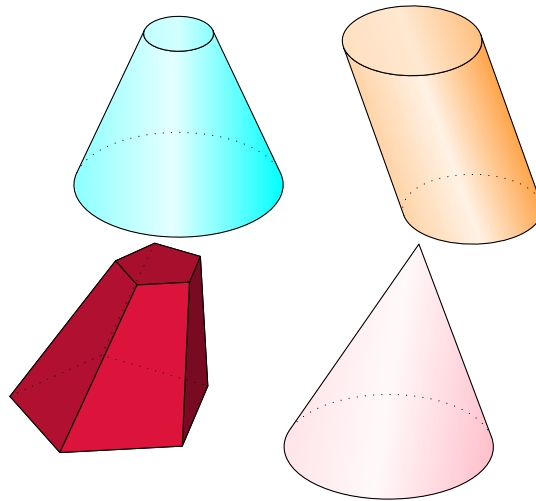
- **frustum(C,R,r,V,A,nbfacet,open)** renvoie un tronc de cône droit. Le point C (point 3d) est le centre de la base circulaire de rayon R, le centre de l'autre base circulaire est le point A, et son rayon est  $r$ , les bases sont orthogonales au vecteur V, mais pas forcément orthogonales à l'axe (AC). L'argument facultatif *nbfacet* vaut 35 par défaut (nombre de facettes latérales). L'argument facultatif *open* est un booléen indiquant si le tronc de cône est ouvert ou non (false par défaut). Dans le cas où il est ouvert, seules les facettes latérales sont renvoyées.
- **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).

```

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

```

FIGURE 11 : 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.

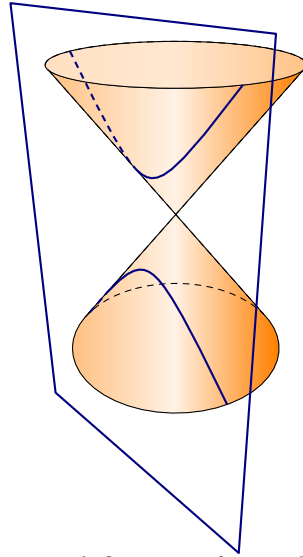
```

1 \begin{luadraw}{name=hyperbole}
2 local g = graph3d:new{window={-8,6,-9,9},bbox=false, viewdir=perspective("central",45,65), size={10,10}}
3 Hiddenlinestyle = "dashed"; Hiddenlines = true
4 local C1 = cone(Origin,4*vecK,3,35,true)
5 local C2 = cone(Origin, -4*vecK,3,35,true)
6 local P = {M(1,-1,-2),vecI} -- sectional plan
7 local I1 = g:Intersection3d(C1,P) -- intersection between cone C1 and plane P
8 local I2 = g:Intersection3d(C2,P) -- intersection between cone C2 and plane P
9 -- I1 et I2 sont de type Edges (arêtes)
10 g:Dcone(Origin,4*vecK,3,{color="orange"}); g:Dcone(Origin,-4*vecK,3,{color="orange"})
11 g:Lineoptions("solid","Navy",8)
12 g:Dedges(I1); g:Dedges(I2) -- drawing of edges I1 and I2
13 g:Dplane(P, vecK,14,9)
14 g:Show()
15 \end{luadraw}

```



FIGURE 12 : Hyperbole : intersection cône - plan



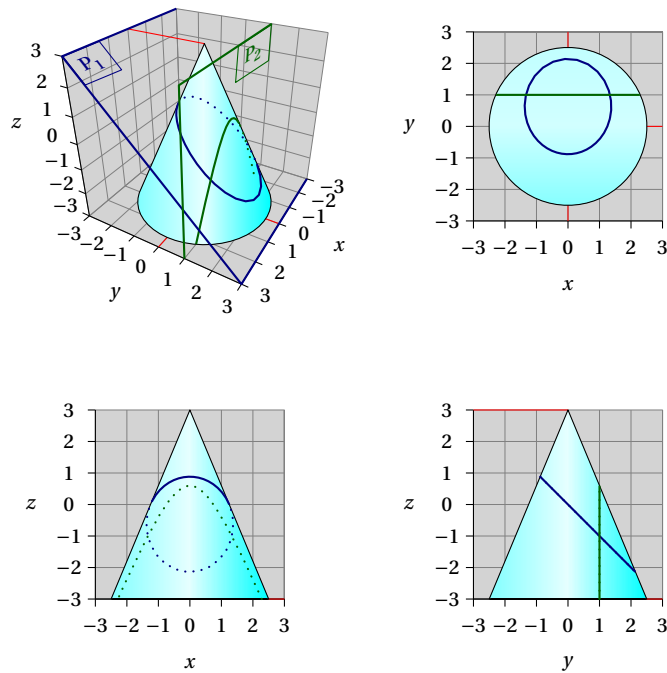
Dans cet exemple, les cônes  $C_1$  et  $C_2$  sont définis sous forme de polyèdres pour déterminer leur intersection avec le plan  $P$ , mais pas pour les dessiner. La méthode **g:Intersection3d(C1,P)** renvoie l'intersection du polyèdre  $C_1$  avec le plan  $P$  sous la forme d'une table à deux champs,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

```

1  \begin{luadraw}{name=several_views}
2  local g = graph3d:new{window3d={-3,3,-3,3,-3,3}, size={10,10}, margin={0,0,0,0}}
3  g:Labelsize("footnotesize")
4  local y0, R = 1, 2.5
5  local C = cone(M(0,0,3),-6*vecK,R,35,true) -- cone ouvert
6  local P1 = {M(0,0,0),vecK+vecJ} -- 1er plan de coupe
7  local P2 = {M(0,y0,0),vecJ} -- 2ieme plan de coupe
8  local I, I2
9  local dessin = function() -- un dessin par vue
10 g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
11 I1 = g:Intersection3d(C,P1) -- intersection entre le cône C et les plans P1 et P2
12 I2 = g:Intersection3d(C,P2) -- I1 et I2 sont de type Edges
13 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" )
14 g:Dcone( M(0,0,3),-6*vecK,R, {color="cyan"})
15 g:Dedges(I1, {hidden=true,color="Navy",width=8})
16 g:Dedges(I2, {hidden=true,color="DarkGreen",width=8})
17 end
18 -- en haut à gauche, vue dans l'espace, on ajoute les plans au dessin
19 g:Saveattr(); g:Viewport(-5,0,0,5); g:Coordsystem(-7,6,-6,5,1); g:Setviewdir(perspective("central")); dessin()
20 g:Dpolyline3d( {M(-3,-3,3),M(3,-3,3),M(3,3,-3),M(-3,3,-3)},"Navy,line width=0.8pt")
21 g:Dpolyline3d( {M(-3,y0,3),M(3,y0,3),M(3,y0,-3)},"DarkGreen,line width=0.8pt")
22 g:Dlabel3d( "$P_1$",M(3,-3,3),{pos="SE",dir={-vecI,-vecJ+vecK},node_options="Navy,draw"})
23 g:Dlabel3d( "$P_2$",M(-3,y0,3),{pos="SW",dir={-vecI,vecK},node_options="DarkGreen,draw"})
24 g:Restoreattr()
25 -- en haut à droite, projection sur le plan xOy
26 g:Saveattr(); g:Viewport(0,5,0,5); g:Coordsystem(-6,6,-6,5,1); g:Setviewdir("xOy"); dessin()
27 g:Restoreattr()
28 -- en bas à gauche, projection sur le plan xOz
29 g:Saveattr(); g:Viewport(-5,0,-5,0); g:Coordsystem(-6,6,-6,5,1); g:Setviewdir("xOz"); dessin()
30 g:Restoreattr()
31 -- en bas à droite, projection sur le plan yOz
32 g:Saveattr(); g:Viewport(0,5,-5,0); g:Coordsystem(-6,6,-6,5,1); g:Setviewdir("yOz"); dessin()
33 g:Restoreattr()
34 g:Show()
35 \end{luadraw}

```

FIGURE 13 : Section de cône avec plusieurs vues



## 5) Lecture dans un fichier obj

La fonction **red\_obj\_file(file)**<sup>1</sup> 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 quatre réels  $\{x1,x2,y1,y2,z1,z2\}$  représentant la boîte 3d englobante (bounding box) du polyèdre.

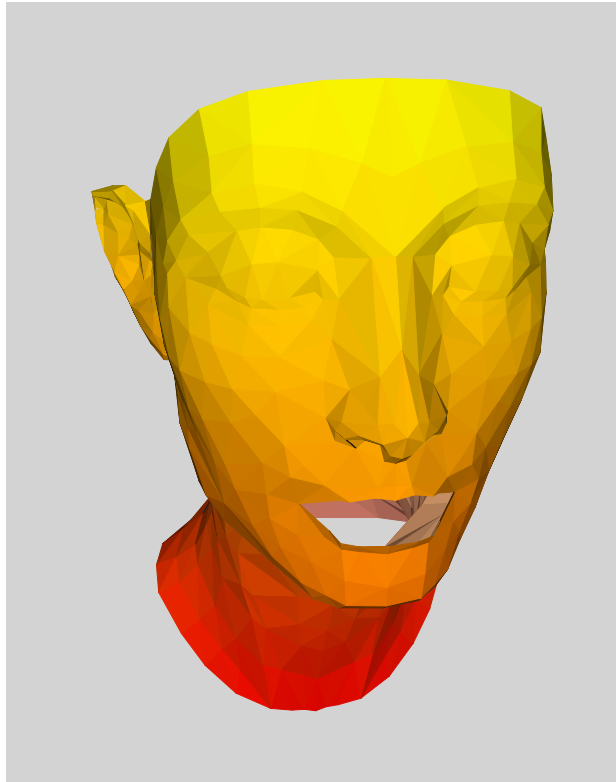
```

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

```

1. Cette fonction est une contribution de Christophe BAL.

FIGURE 14 : Masque de Nefertiti



## 6) 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 S est la liste de facettes et *options* une table définissant les options. Celles-ci sont :

- **mode=** : définit le mode de représentation.
  - *mode=mWireframe* : mode fil de fer, on dessine les arêtes seulement.
  - *mode=mFlat* ou *mFlatHidden* : on dessine les faces de couleur unie, ainsi que les arêtes.
  - *mode=mShaded* ou *mShadedHidden* : on dessine les faces de couleur nuancée en fonction de leur inclinaison, ainsi que les arêtes. Le mode par défaut est 3.
  - *mode=mShadedOnly* : on dessine les faces de couleur nuancée en fonction de leur inclinaison, mais pas les arêtes.
- **contrast** : c'est un nombre qui vaut 1 par défaut. Ce nombre permet d'accentuer ou diminuer la nuance des couleurs des facettes dans les modes *mShaded*, *mShadedHidden*, *mShadedOnly*.
- **edgestyle** : est une chaîne qui définit le style de ligne des arêtes. C'est le style courant par défaut.
- **edgecolor** : est une chaîne qui définit la couleur des arêtes. C'est la couleur courante des lignes par défaut.
- **hiddenstyle** : est une chaîne qui définit le style de ligne des arêtes cachées. Par défaut c'est la valeur contenue dans la variable globale *Hiddenlinestyle* (qui vaut elle-même "dotted" par défaut).
- **hiddencolor** : est une chaîne qui définit la couleur des arêtes cachées. C'est la couleur courante des lignes par défaut.
- **edgewidth** : épaisseur de trait des arêtes en dixième de point. C'est l'épaisseur courante par défaut.
- **opacity** : 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** : booléen qui vaut false par défaut. Lorsqu'il a 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** : booléen qui vaut false par défaut. Lorsqu'il a 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.

- **color** : chaîne définissant la couleur de remplissage des facettes, c'est "white" par défaut.
- **usepalette** (*nil* par défaut), 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}*, 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, et *mode* qui est une chaîne qui peut être soit "x", soit "y", ou soit "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.

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.

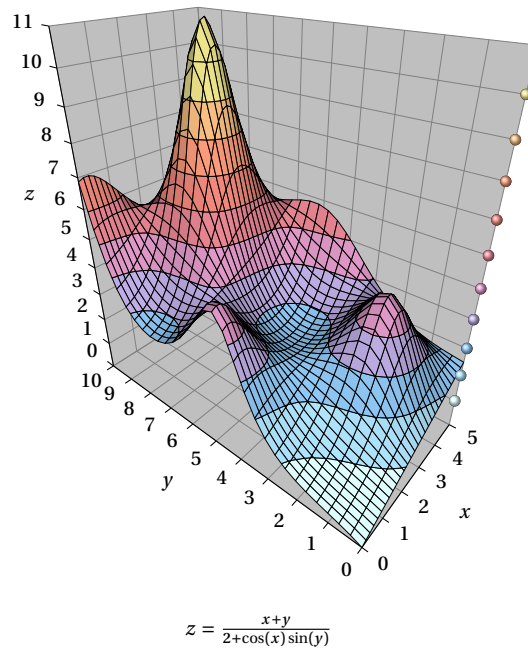
Cette méthode est utile pour dessiner plusieurs solides ensemble, à condition qu'il n'y ait pas d'intersections entre les objets, car celles-ci ne sont pas gérées ici.

```

1 \begin{luadraw}{name=courbes_niv}
2 local cos, sin = math.cos, math.sin, math.pi
3 local g = graph3d:new{window3d={0,5,0,10,0,11}, adjust2d=true, size={10,10},
4   viewdir=perspective("central",220,60,15,M(2.5,5,5.5))}
5 g:Labelsize("footnotesize")
6 local S = cartesian3d(function(u,v) return (u+v)/(2+cos(u)*sin(v)) end,0,5,0,10,{30,30})
7 local n = 10 -- nombre de niveaux
8 local Colors = getpalette(palGasFlame,n,true) -- liste de 10 couleurs au format table
9 local niv, S1 = {}
10 for k = 1, n do
11   S1, S = cutfacet(S,{M(0,0,k),-vecK}) -- section de S avec le plan z=k
12   insert(niv,{S1, {color=Colors[k],mode=mShaded,edgewidth=0.5}}) -- S1 est la partie sous le plan et S au dessus
13 end
14 insert(niv,{S, {color=Colors[n+1]}}) -- insertion du dernier niveau
15 -- niv est une liste du type {facettes1, options1, facettes2, options2, ...}
16 g:Dboxaxes3d({grid=true, gridcolor="gray",fillcolor="lightgray"})
17 g:Dmixfacet(table.unpack(niv))
18 for k = 1, n do
19   g:Dballdots3d( M(5,0,k), rgb(Colors[k]))
20 end
21 g:Dlabel("$z=\frac{x+y}{2+\cos(x)\sin(y)}$", Z((g:Xinf()+g:Xsup())/2, g:Yinf()), {pos="N"})
22 g:Show()
23 \end{luadraw}

```

FIGURE 15 : Exemple de courbes de niveaux sur une surface



## 7) Fonctions de construction de listes de facettes

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

### surface()

La fonction **surface(f,u1,u2,v1,v2,grid)** renvoie la surface paramétrée par la fonction  $f: (u, v) \mapsto f(u, v) \in \mathbb{R}^3$ . L'intervalle pour le paramètre  $u$  est donné par  $u1$  et  $u2$ . L'intervalle pour le paramètre  $v$  est donné par  $v1$  et  $v2$ . Le paramètre facultatif *grid* 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  $v$ .

Il y a deux variantes pour les surfaces :

### cartesian3d()

La fonction **cartesian3d(f,x1,x2,y1,y2,grid,addWall)** renvoie la surface cartésienne d'équation  $z = f(x, y)$  où  $f: (x, y) \mapsto f(x, y) \in \mathbb{R}$ . L'intervalle pour  $x$  est donné par  $x1$  et  $x2$ . L'intervalle pour  $y$  est donné par  $y1$  et  $y2$ . Le paramètre facultatif *grid* 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 *addWall* 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  $x$ <sup>2</sup>. Avec la valeur "y" (ou "xy") c'est une liste de facettes séparatrices (murs) entre chaque "couche" correspond à deux valeurs consécutives du paramètre  $y$ <sup>3</sup>. 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 :

```
1 \begin{luadraw}{name=point_col}
2 local g = graph3d:new{window3d={-2,2,-2,2,-4,4}, window={-3.5,3,-5,5}, size={8,9,0}, viewdir={120,60}}
3 local S = cartesian3d(function(u,v) return u^2-v^2 end, -2,2,-2,2,{20,20}) -- surface of equation z=x^2-y^2
4 local Tx = g:Intersection3d(S, {Origin,vecI}) --intersection of S with the yOz plane
5 local Ty = g:Intersection3d(S, {Origin,vecJ}) --intersection of S with the xOz plane
6 g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray",drawbox=true})
```

2. Ces cloisons sont en fait des plans d'équation  $x = \text{constante}$

3. Ces cloisons sont en fait des plans d'équation  $y = \text{constante}$

```

7 g:Dfacet(S,{mode=mShadedOnly,color="ForestGreen"}) -- surface drawing
8 g:Dedges(Tx, {color="Crimson", hidden=true, width=8}) -- intersection with yOz
9 g:Dedges(Ty, {color="Navy",hidden=true, width=8}) -- intersection with xOz
10 g:Dpolyline3d( {M(2,0,4),M(-2,0,4),M(-2,0,-4)}, "Navy,line width=.8pt")
11 g:Dpolyline3d( {M(0,-2,4),M(0,2,4),M(0,2,-4)}, "Crimson,line width=.8pt")
12 g:Show()
13 \end{luadraw}

```

### cylindrical\_surface()

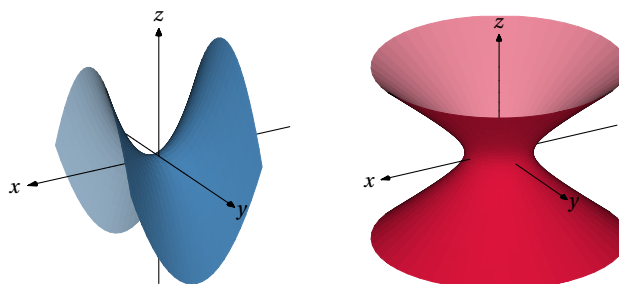
La fonction **cylindrical\_surface**(*r,z,u1,u2,theta1,theta2,grid,addWall*) renvoie la surface paramétrée en cylindrique par  $r(u,theta)$ ,  $theta$ ,  $z(u,theta)$ . Les arguments *r* et *z* sont donc deux fonctions de *u* et  $\theta$ , à valeurs réelles. L'intervalle pour *u* est donné par *u1* et *u2*. L'intervalle pour  $\theta$  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  $\theta$ <sup>4</sup>. 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*<sup>5</sup>, 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 *theta*). 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.

```

1 \begin{luadraw}{name=surface_with_addWall}
2 local pi, ch, sh = math.pi, math.cosh, math.sinh
3 local g = graph3d:new{window3d={-4,4,-4,4,-5,5}, window={-10,10,-4,4}, size={10,10}, viewdir={60,60}}
4 g:Labelsize("footnotesize")
5 local S,wall = cartesian3d(function(x,y) return x^2-y^2 end,-2,2,-2,2,nil,"xy")
6 g:Saveattr(); g:Viewport(-10,0,-4,4); g:Coordsystem(-4.5,4.5,-4.5,4.75)
7 g:Dscene3d(
8   g:addWall(wall), -- 2 facet cutouts with this instruction, and 529 facet cutouts without it
9   g:addFacet(S,{color="SteelBlue"}),
10  g:addAxes(Origin,{arrows=1}) )
11 g:Restoreattr()
12 g:Saveattr(); g:Viewport(0,10,-4,4); g:Coordsystem(-5,5,-5,5)
13 local r = function(u,v) return ch(u) end
14 local z = function(u,v) return sh(u) end
15 S,wall = cylindrical_surface(r,z,2,-2,-pi,pi,{25,51},"zv")
16 g:Dscene3d(
17   g:addWall(wall), -- 13 facet cutouts with this instruction, and more than 17000 facet cutouts without it ...
18   g:addFacet(S,{color="Crimson"}),
19   g:addAxes(Origin,{arrows=1}) )
20 g:Restoreattr()
21 g:Show()
22 \end{luadraw}

```

FIGURE 16 : Surfaces utilisant l'option *addWall*



4. Ces cloisons sont en fait des plans d'équation  $\theta = \text{constante}$

5. Ces cloisons sont en fait des plans d'équation  $z = \text{constante}$

**curve2cone()**

La fonction **curve2cone(f,t1,t2,S,args)** construit un cône de sommet S (point 3d) et de base la courbe paramétrée par  $f: t \mapsto f(t) \in \mathbb{R}^3$  sur l'intervalle défini par  $t1$  et  $t2$ . L'argument *args* est une table facultative pour définir les options, qui sont :

- **nbdots** qui représente le nombre minimal de points de la courbe à calculer (15 par défaut).
- **ratio** qui est un nombre représentant le rapport d'homothétie (de centre le sommet S) pour construire l'autre partie du cône. Par défaut *ratio* vaut 0 (pas de deuxième partie).
- **nbddiv** qui est un 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. Par défaut *nbddiv* vaut 0.

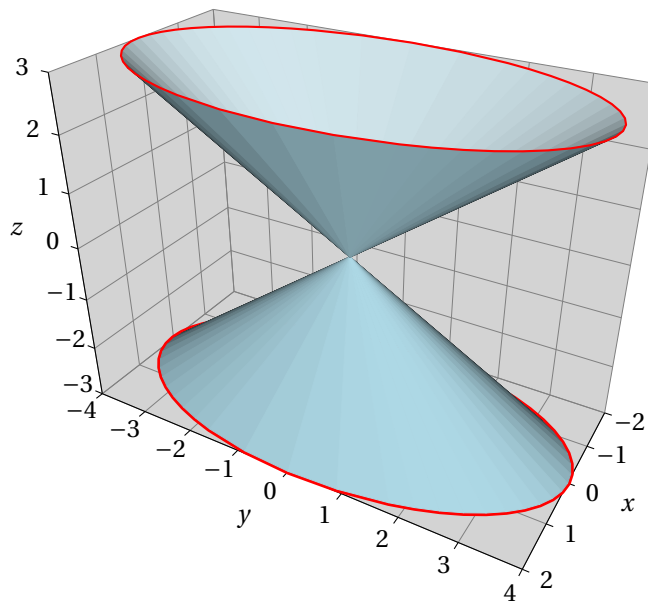
Cette fonction renvoie une liste de facettes, suivie d'une ligne polygonale 3d qui représente les bords du cône.

```

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

```

FIGURE 17 : Exemple de cône elliptique

**curve2cylinder()**

La fonction **curve2cylinder(f,t1,t2,V,args)** construit un cylindre d'axe dirigé par le vecteur V (point 3d) et de base la courbe paramétrée par  $f: t \mapsto f(t) \in \mathbb{R}^3$  sur l'intervalle défini par  $t1$  et  $t2$ . La seconde base est la translatée de la première avec le vecteur V. L'argument *args* est une table facultative pour définir les options, qui sont :

- **nbdots** qui représente le nombre minimal de points de la courbe à calculer (15 par défaut).
- **nbddiv** qui est un 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. Par défaut *nbddiv* vaut 0.

Cette fonction renvoie une liste de facettes, suivie d'une ligne polygonale 3d qui représente les bords du cylindre.

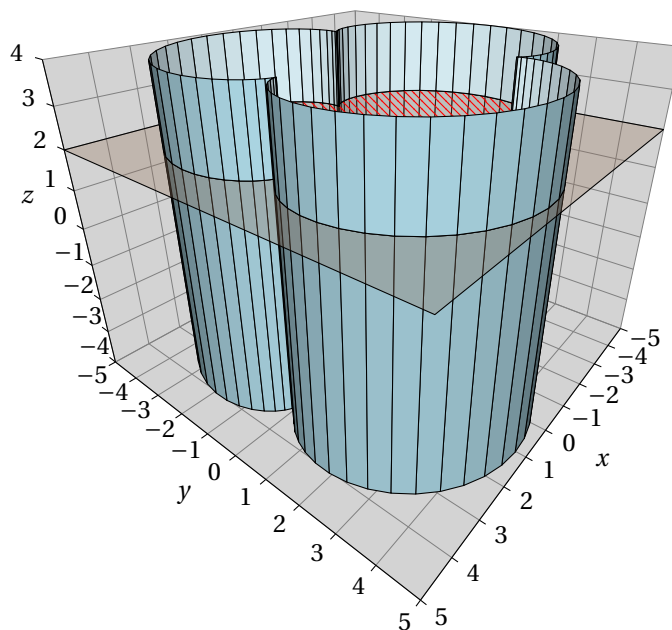


```

1 \begin{luadraw}{name=curve2cylinder}
2 local cos, sin, pi = math.cos, math.sin, math.pi
3 local g = graph3d:new{
4   window3d={-5,5,-5,5,-4,4},window={-9,8,-10.5,5.5},viewdir=perspective("central",39,64),size={10,10}}
5 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
6 local V = 8*vecK
7 local C = curve2cylinder(f,-pi,pi,V,{nbdots=25,nbdiv=2})
8 local plan = {M(0,0,2), -vecK} -- plan de coupe z=2
9 local C1, C2, section = cutfacet(C,plan)
10 g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
11 g:Dfacet(C1, {mode=mShaded,color="LightBlue"}) -- partie sous le plan
12 g:Dfacet(g:Plane2facet(plan), {opacity=0.3,color="Chocolate"}) -- dessin du plan sous forme d'une facette
13 g:Filloptions("fdiag","red"); g:Dpolyline3d(section) -- dessin de la section
14 g:Dfacet(C2, {mode=3,color="LightBlue"}) -- partie du cylindre au dessus du plan
15 g:Show()
16 \end{luadraw}

```

FIGURE 18 : Section d'un cylindre non circulaire



### line2tube(); section2tube()

La fonction **line2tube(L,r,args)** construit (sous forme d'une liste de facettes) un tube centré sur  $L$  qui doit être une ligne polygonale 3d (liste de points 3d ou liste de listes de points 3d), l'argument  $r$  représente le rayon de ce tube. L'argument  $args$  est une table pour définir les options, qui sont :

- **nbfacet** : nombre indiquant le nombre de facettes latérales du tube (3 par défaut).
- **close** : booléen indiquant si la ligne polygonale  $L$  doit être refermée (false par défaut).
- **hollow** : booléen indiquant si les deux extrémités du tube doivent être ouvertes ou non (false par défaut). Lorsque l'option **close** vaut true, l'option **hollow** prend automatiquement la valeur true.
- **addwall** : nombre qui vaut 0 ou 1 (0 par défaut). Lorsque cette option vaut 1, la fonction renvoie, après la liste des facettes composant le tube, une liste de facettes séparatrices (murs) entre chaque "tronçon" du tube, ce qui peut être utile avec la méthode **g:Dscene3d** (uniquement).

La fonction **section2tube(section,L,args)** 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  $args$  est une table pour définir les options, qui sont :

- **close** : booléen indiquant si la ligne polygonale  $L$  doit être refermée (false par défaut).
- **hollow** : booléen indiquant si les deux extrémités du tube doivent être ouvertes ou non (false par défaut). Lorsque l'option **close** vaut true, l'option **hollow** prend automatiquement la valeur true.



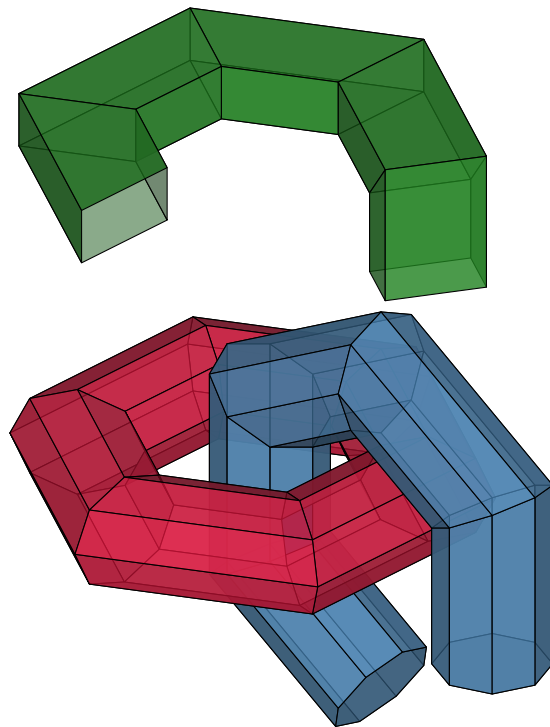
- **addwall** : nombre qui vaut 0 ou 1 (0 par défaut). Lorsque cette option vaut 1, la fonction renvoie, après la liste des facettes composant le tube, une liste de facettes séparatrices (murs) entre chaque "tronçon" du tube, ce qui peut être utile avec la méthode **g:Dscene3d** (uniquement).

```

1 \begin{luadraw}{name=line2tube_section2tube}
2 local g = graph3d:new{window={-5,6,-4.5,8}, viewdir={45,60}, margin={0,0,0,0}, size={10,10}}
3 local L1 = map(toPoint3d,polyreg(0,3,6)) -- hexagone régulier dans le plan xOy, centre O de sommet M(3,0,0)
4 local L2 = shift3d(rotate3d(L1,90,{Origin,vecJ}),3*vecJ)
5 local L3 = shift3d(reverse(L1),6*vecK)
6 L3[6] = L3[5]-2*vecK -- modification of the last point
7 local section = shift3d({M(2,0,0.5),M(4,0,0.5),M(4,0,-0.5),M(2,0,-0.5)},6*vecK)
8 local T1 = line2tube(L1,1,{nbfacet=8,close=true}) -- tube 1 refermé
9 local T2 = line2tube(L2,1,{nbfacet=8}) -- tube 2 non refermé
10 local T3 = section2tube(section, L3,{hollow=true})
11 g:Dmixfacet( T1, {color="Crimson",opacity=0.8}, T2, {color="SteelBlue"}, T3, {color="ForestGreen"} )
12 g:Show()
13 \end{luadraw}

```

FIGURE 19 : Exemple avec line2tube et section2tube



### rotcurve()

La fonction **rotcurve(p,t1,t2,axe,angle1,angle2,args)** 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 {point3d, vecteur 3d} représentant une droite orientée de l'espace), d'un angle allant de *angle1* (en degrés) à *angle2*. L'argument *args* est une table pour définir les options, qui sont :

- **grid** : 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. Par défaut la valeur de **grid** est {25,25}.
- **addwall** : nombre qui vaut 0 ou 1 ou 2 (0 par défaut). 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).

```

1 \begin{luadraw}{name=rotcurve}
2 local cos, sin, pi, i = math.cos, math.sin, math.pi, cpx.I

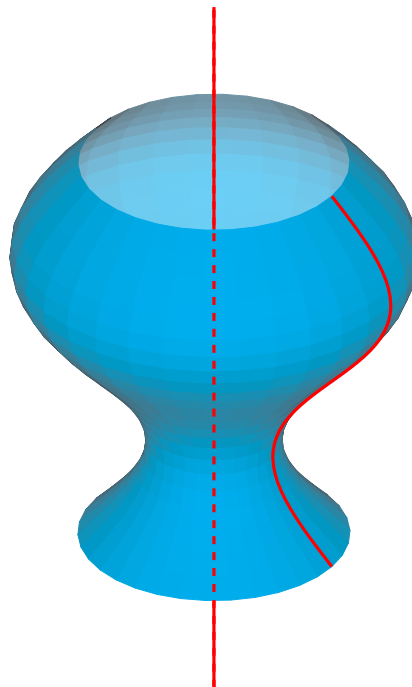
```

```

3 local g = graph3d:new{viewdir={30,60},size={10,10}}
4 local p = function(t) return M(0,sin(t)+2,t) end -- courbe dans le plan yOz
5 local axe = {Origin,vecK}
6 local S = rotcurve(p,pi,-pi,axe,0,360,{grid={25,35}})
7 local visible, hidden = g:Classifyfacet(S)
8 g:Dfacet(hidden, {mode=mShadedOnly,color="cyan"})
9 g:Dline3d(axe,"red,line width=1.2pt")
10 g:Dfacet(visible, {mode=5,color="cyan"})
11 g:Dline3d(axe,"red,line width=1.2pt,dashed")
12 g:Dparametric3d(p,{t={-pi,pi}},draw_options="red,line width=1.2pt")
13 g:Show()
14 \end{luadraw}

```

FIGURE 20 : 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 **rotline(L,axe,angle1,angle2,args)** 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 {point3d, vecteur 3d} représentant une droite orientée de l'espace), d'un angle allant de *angle1* (en degrés) à *angle2*. L'argument *args* est une table pour définir les options, qui sont :

- **nbdots** : qui est le nombre de points calculés pour le paramètre angulaire. Par défaut la valeur de **nbdots** est 25.
- **close** : booléen qui indique si L doit être refermée (false par défaut).
- **addwall** : nombre qui vaut 0 ou 1 ou 2 (0 par défaut). 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 points consécutifs dans la liste L), 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).

```

1 \begin{luadraw}{name=rotline}
2 local g = graph3d:new{window={-4,4,-4,4},size={10,10}}
3 local L = {M(0,0,4),M(0,4,0),M(0,0,-4)} -- liste de points dans le plan yOz

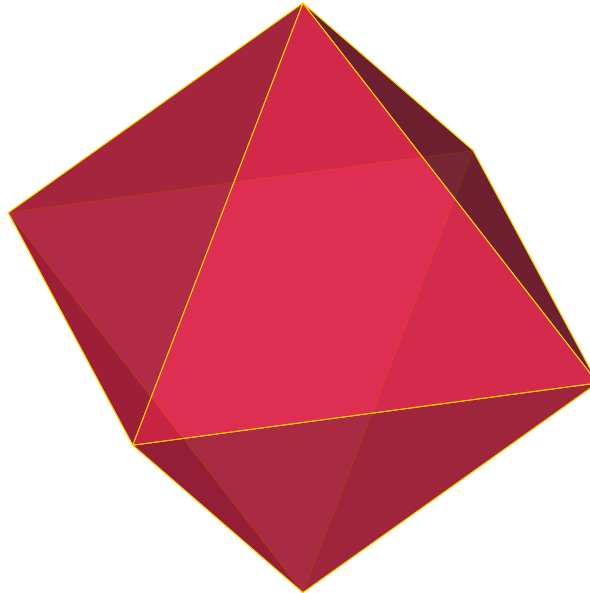
```

```

4 local axe = {Origin,vecK}
5 local S = rotline(L,axe,0,360,{nbdots=5}) -- le point 1 et le point 5 sont confondus
6 g:Dfacet(S,{color="Crimson",edgecolor="Gold",opacity=0.8})
7 g:Show()
8 \end{luadraw}

```

FIGURE 21 : Exemple avec rotline



## 8) 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 P est visible lorsqu'elle appartient à au moins une face visible.
- La méthode **g:Intersection3d(P,plane)** où P est un polyèdre ou bien une liste de facettes, renvoie sous forme d'objet de type "edge" l'intersection entre P 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 définissant les options, celles-ci sont :
  - **hidden** : booléen qui indique si les arêtes cachées doivent être dessinées (false par défaut).
  - **visible** : booléen qui indique si les arêtes visibles doivent être dessinées (true par défaut).
  - **clip** : booléen qui indique si les arêtes doivent être clippées par la fenêtre 3d (false par défaut).
  - **hiddenstyle** : chaîne de caractères définissant le style de ligne des arêtes cachées, par défaut cette option contient la valeur de la variable globale *Hiddenlinestyle* (qui vaut "dotted" par défaut).
  - **hiddencolor** : chaîne de caractères définissant la couleur des arêtes cachées, par défaut cette option contient la même couleur que l'option **color**.
  - **style** : chaîne de caractères définissant le style de ligne des arêtes visibles, par défaut cette option contient le style courant du dessin de lignes.
  - **color** : chaîne de caractères définissant la couleur des arêtes visibles, par défaut cette option contient la couleur courante de dessin de lignes.
  - **width** : nombre représentant l'épaisseur de trait des arêtes (en dixième de point), par défaut cette variable contient l'épaisseur courante du dessin de lignes.
- **Complément :**

- La fonction **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**.
- La fonction **facetvertices(F)** où F est une liste de facettes ou bien un polyèdre, renvoie la liste de tous les sommets de F (points 3d).

## 9) 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'une 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 (*Dpoly*, *Dfacet* et *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 **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 **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 **poly2facet(P)**).
- La fonction **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 **reverse\_face\_orientation(F)** où F est 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 a été modifiée.

```

1 \begin{luadraw}{name=sphere_octaedre}
2 require "luadraw_polyhedrons"
3 local g = graph3d:new{ window3d={-3,3,-3,3,-3,3}, size={10,10}}
4 local P = octahedron(Origin,M(0,0,3)) -- polyèdre défini dans le module luadraw_polyhedrons
5 P = rotate3d(P,-10,{Origin,vecK}) -- rotate3d sur un polyèdre renvoie un polyèdre
6 local V, H = g:Classifyfacet(P) -- V pour facettes visibles, H pour hidden
7 local S = map(function(p) return {proj3d(Origin,p),p[2]} end, facet2plane(V) )
8 -- S contient la liste de : {projeté, vecteur normal} (projetés de Origin sur les faces visibles)
9 local R = pt3d.abs(S[1][1]) -- rayon de la sphère
10 g:Dboxaxes3d({grid=true, gridcolor="gray", fillcolor="LightGray"})
11 g:Dfacet(H, {color="blue",opacity=0.9}) -- dessin des facettes non visibles
12 g:Dsphere(Origin,R,{mode=mBorder,color="orange"}) -- dessin de la sphère
13 g:Dballdots3d(Origin,"gray",0.75) -- centre de la sphère

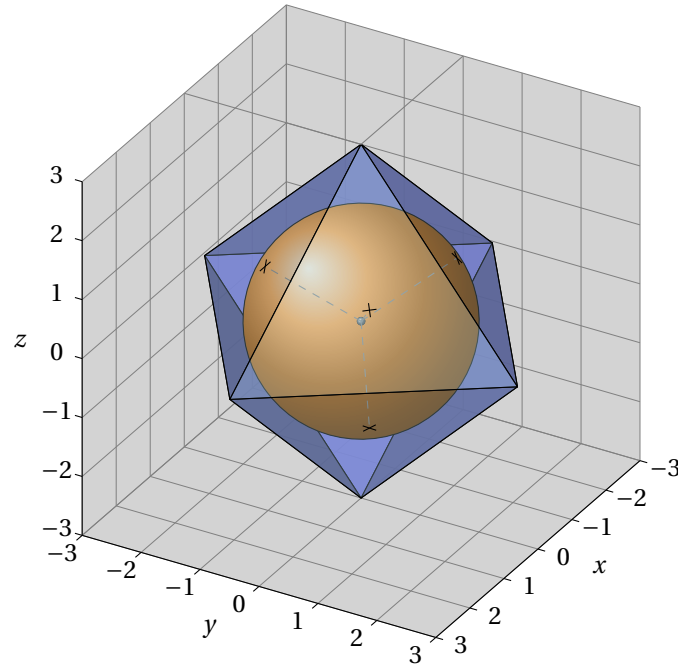
```

```

14 for _,D in ipairs(S) do -- segments reliant l'origine aux projetés
15     g:Dpolyline3d( {0,Origin,D[1]}, "dashed,gray")
16 end
17 g:Dfacet(V,{opacity=0.4, color="LightBlue"}) -- facettes visibles de l'octaèdre
18 g:Dcrossdots3d(S,nil,0.75) -- dessin des projetés sur les faces
19 g:Dpolyline3d( {M(0,-3,3), M(0,0,3), M(-3,0,3)}, "gray")
20 g:Show()
21 \end{luadraw}

```

FIGURE 22 : Sphère inscrite dans un octaèdre avec projection du centre sur les faces



## 10) Découper un solide : cutpoly et cutfacet

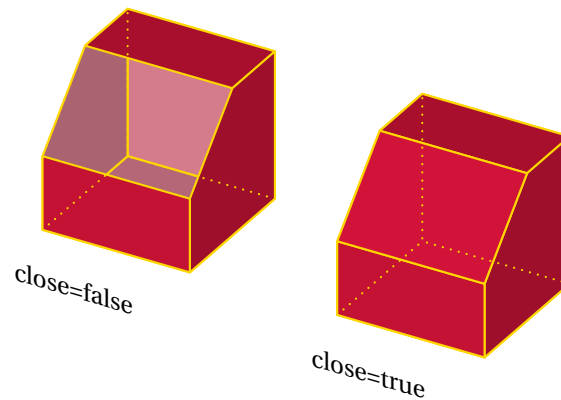
- La fonction **cutpoly(P,plane,close)** permet de découper le polyèdre  $P$  avec le plan  $plane$  (table du type  $\{A,n\}$  où  $A$  est un point du plan et  $n$  un vecteur normal au plan). La fonction renvoie 3 choses : la partie située dans le demi-espace contenant le vecteur  $n$  (sous forme 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.

```

1 \begin{luadraw}{name=cutpoly}
2 local g = graph3d:new{window3d={-3,3,-3,3,-3,3}, window={-4,4,-3,3},size={10,10}}
3 local P = parallelep(M(-1,-1,-1),2*vecI,2*vecJ,2*vecK)
4 local A, B, C = M(0,-1,1), M(0,1,1), M(1,-1,0)
5 local plane = {A, pt3d.prod(B-A,C-A)}
6 local P1 = cutpoly(P,plane)
7 local P2 = cutpoly(P,plane,true)
8 g:Lineoptions(nil,"Gold",8)
9 g:Dpoly( shift3d(P1,-2*vecJ), {color="Crimson",mode=mShadedHidden} )
10 g:Dpoly( shift3d(P2,2*vecJ), {color="Crimson",mode=mShadedHidden} )
11 g:Dlabel3d(
12     "close=false", M(2,-2,-1), {dir={vecJ,vecK}},
13     "close=true", M(2,2,-1), {}
14 )
15 g:Show()
16 \end{luadraw}

```

FIGURE 23 : Cube coupé par un plan (cutpoly), avec *close=false* et avec *close=true*

- La fonction **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 15.

## 11) Clipper des facettes avec un polyèdre convexe : clip3d

La fonction **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 polyèdre) avec la fenêtre 3d courante peut se faire avec cette fonction de la manière suivante :

$$S = \text{clip3d}(S, g:\text{Box3d}())$$

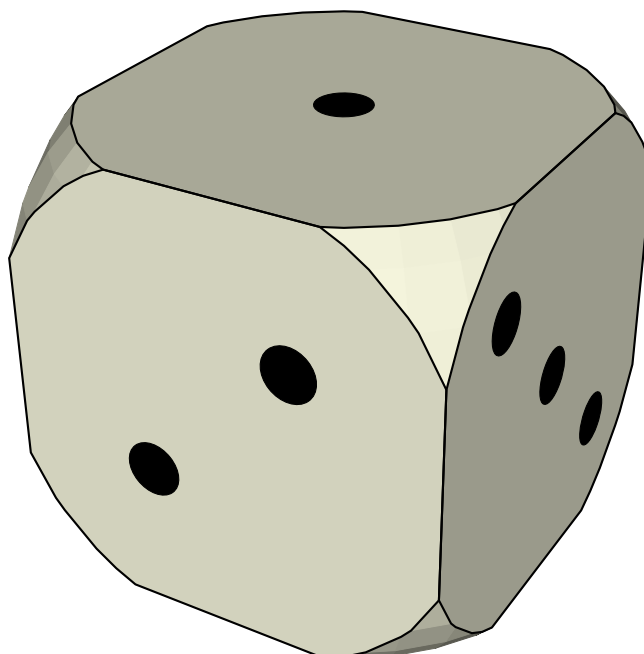
En effet, la méthode **g:Box3d()** renvoie la fenêtre 3d courante sous forme d'un parallélépipède.

```

1 \begin{luadraw}{name=clip3d}
2 local g = graph3d:new{window={-3,3,-3,3},size={10,10},viewdir=perspective("central")}
3 local S = sphere(Origin,3)
4 local C = parallelep(M(-2,-2,-2),4*vecI,4*vecJ,4*vecK)
5 local C1 = clip3d(S,C) -- sphère clippée par le cube
6 local C2 = clip3d(C,S) -- cube clippé par la sphère
7 local V = g:Classifyfacet(C2) -- facettes visibles de C2
8 g:Dfacet( concat(C1,C2), {color="Beige",mode=mShadedOnly,backcull=true} ) -- que les faces visibles
9 g:Dpolyline3d(V,true,"line width=0.8pt") -- contour des faces visibles de C2
10 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
11 g:Filloptions("full","black")
12 g:Dcircle3d( D,0.25,vecK); g:Dcircle3d( (2*A+B)/3,0.25,vecI)
13 g:Dcircle3d( (A+2*B)/3,0.25,vecI); g:Dcircle3d( (3*B+C)/4,0.25,vecJ)
14 g:Dcircle3d( (B+C)/2,0.25,vecJ); g:Dcircle3d( (B+3*C)/4,0.25,vecJ)
15 g:Show()
16 \end{luadraw}

```

FIGURE 24 : Exemple avec clip3d : construction d'un dé à partir d'un cube et d'une sphère



## 12) Clipper un plan avec un polyèdre convexe : clipplane

La fonction **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<sup>6</sup>, 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 vaud se tourner alors vers des d'autres outils comme povray 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 :

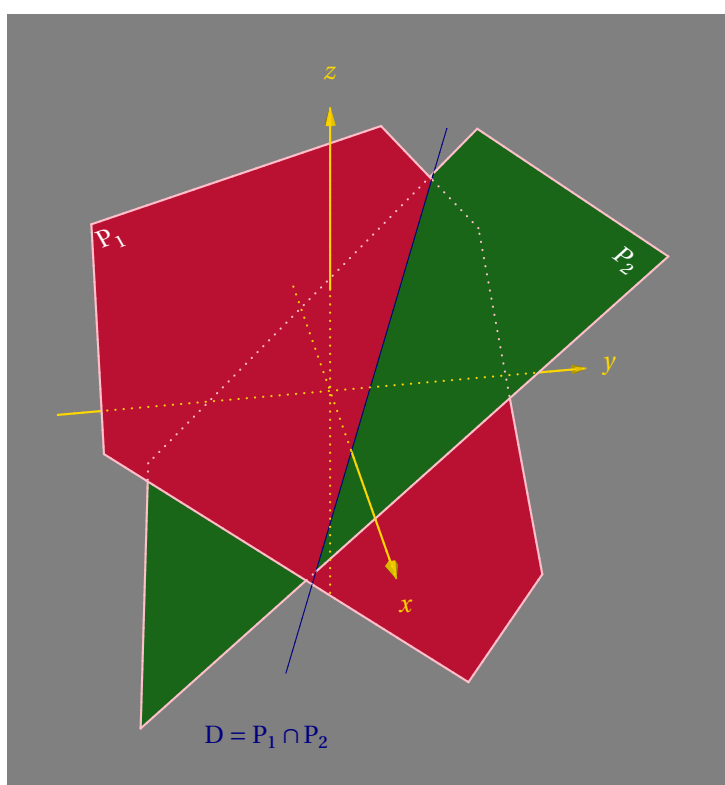
- des polyèdres,
- des listes de facettes (avec point3d),
- des lignes polygonales 3d,
- des points 3d,
- des labels,
- des axes,
- des plans, des droites,
- des angles,
- des cercles, des arcs de cercle.

```

1 \begin{luadraw}{name=intersection_plans}
2 local g = graph3d:new{viewdir=perspective("central",-10,60), window={-5,6.5,-6.5,6},bg="gray", size={10,10}}
3 local P1 = planeEq(1,1,1,-2) -- plan d'équation x+y+z-2=0
4 local P2 = {Origin, vecK-vecJ} -- plan passant par O et normal à (1,1,1)
5 local D = interPP(P1,P2) -- droite d'intersection entre P1 et P2 (D = {A,u})
6 local posD = D[1]+1.85*D[2] -- pour placer le label
7 Hiddenlines = true; Hiddenlinestyle = "dotted" -- affichage des lignes cachées en pointillées
8 g:Dscene3d(
9   g:addPlane(P1, {color="Crimson",edge=true,edgecolor="Pink",edgewidth=8}), -- ajout du plan P1
10  g:addPlane(P2, {color="ForestGreen",edge=true,edgecolor="Pink",edgewidth=8}), -- ajout du plan P2
11  g:addLine(D, {color="Navy",edgewidth=12}), -- ajout de la droite D
12  g:addAxes(Origin, {arrows=1, color="Gold",width=8}), -- ajout des axes fléchés
13  g:addLabel( -- ajout de labels, ceux-ci auraient pu être ajoutés par dessus la scène
14    "$D=P_1 \cap P_2$",posD,{color="Navy"},
15    "$P_2$", M(3,0,0)+3.5*M(0,1,1)+0.2*vecI,{color="white",dir={vecI,vecJ+vecK}},
16    "$P_1$",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)}}
17  )
18 )
19 g:Show()
20 \end{luadraw}

```

FIGURE 25 : 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(list,options)** permet d'ajouter le polyèdre P à la scène.

Dans les deux cas, l'argument facultatif *options* est une table à 12 champs, 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* par défaut), 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}*, 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, et *mode* qui est une chaîne qui peut être soit "x", soit "y", ou soit "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.
- **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.
- **edge=false** : booléen qui indique si les arêtes doivent être ajoutées à la scène.
- **edgecolor=** : indique la couleur des arêtes lorsqu'elles sont dessinées, c'est la couleur courante par défaut.
- **edgewidth=** : indique l'épaisseur de trait (en dixième de point) des arêtes, c'est l'épaisseur courante par défaut.
- **hidden=Hiddenlines** : booléen qui indique si les arêtes cachées doivent être représentées. *Hiddenlines* est une variable globale qui vaut *false* par défaut.
- **hiddenstyle=Hiddenlinestyle** : chaîne définissant le style de ligne des arêtes cachées. *Hiddenlinestyle* est une variable globale qui vaut "dotted" par défaut.
- **matrix=ID3d** : matrice 3d de transformation des facettes, par défaut celle-ci est la matrice 3d de l'identité, c'est à dire la table  $\{M(0,0,0), \text{vecI}, \text{vecJ}, \text{vecK}\}$ .

#### Ajouter un plan : **g:addPlane** et **g:addPlaneEq**

La méthode **g:addPlane(P,options)** permet d'ajouter le plan P 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'argument *window3d* (lui-même défini 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 *coef* 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 *options* est une table à 12 champs, ces options sont celles de la méthode **g:addFacet**, plus l'option **scale=1** : ce nombre est un rapport d'homothétie, 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 L est une liste de points 3d, ou une liste de listes de points 3d, permet d'ajouter L à la scène. L'argument facultatif *options* est une table à 10 champs, ces options (avec leur valeur par défaut) sont :

- **style="solid"** : pour définir le style de ligne, c'est le style courant par défaut.

- `color=` : couleur de la ligne, c'est la couleur courante par défaut.
- `close=false` : indique si la ligne L (ou chaque composante de L) doit être refermée.
- `clip=false` : indique si la ligne L (ou chaque composante de L) doit être clippée par la fenêtre 3d.
- `width=` : 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=Hiddenlines` : booléen qui indique si les parties cachées de la ligne doivent être représentées. *Hiddenlines* est une variable globale qui vaut false par défaut.
- `hiddenstyle=Hiddenlinestyle` : chaîne définissant le style de ligne des parties cachées. *Hiddenlinestyle* est une variable globale qui vaut "dotted" 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` : permet de réduire ou augmenter la taille des flèches.
- `matrix=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  $\{M(0,0,0), \text{vecI}, \text{vecJ}, \text{vecK}\}$ .

### Ajouter des axes : `g:addAxes`

La méthode `g:addAxes(O,options)` permet d'ajouter les axes :  $(O, \text{vecI})$ ,  $(O, \text{vecJ})$  et  $(O, \text{vecK})$  à la scène 3d, où l'argument *O* est un point 3d. Les options sont celles de la méthode `g:addPolyline`, plus l'option `legend=true` qui permet d'ajouter automatiquement le nom de chaque axe (*x*, *y* et *z*) à l'extrémité. Ces axes ne sont pas gradués.

### Ajouter une droite : `g:addLine`

La méthode `g:addLine(d,options)` permet d'ajouter la droite *d* à la scène, cette droite *d* 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 *options* est une table à 10 champs, ces options sont celles de la méthode `g:addPolyline`, plus l'option `scale=1` : ce nombre est un rapport d'homothétie, on applique à la facette 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'argument *window3d* (lui-même défini 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  $\widehat{BAC}$  sous forme d'un parallélogramme de côté *r* (*r* vaut 0.25 par défaut), seuls deux côtés sont représentés. les arguments *B*, *A* et *C* 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 *A* (point 3d), de rayon *r*, allant de *B* vers *C* (points 3d) dans le sens direct si *sens* vaut 1 (indirect sinon). L'arc est tracé dans le plan passant par *A* et orthogonal au vecteur *normal* (point 3d non nul), c'est ce même vecteur qui oriente le plan. 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 *A* (point 3d) et de rayon *r* dans le plan passant par *A* et orthogonal au vecteur *normal* (point 3d non nul). Les options sont celles de la méthode `g:addPolyline`.

```

1 \begin{luadraw}{name=cylindres_imbriques}
2 local g = graph3d:new{window={-5,5,-7,5}, viewdir=perspective("central",30,65,20),size={10,10},margin={0,0,0,0}}
3 Hiddenlines = false
4 local R, r, A, B = 3, 1.5
5 local C1 = cylinder(M(0,0,-5),5*vecK,R) -- pour modéliser l'eau
6 local C2 = cylinder(Origin,2*vecK,R,35,true) -- partie du contenant au dessus de l'eau (cylindre ouvert)
7 local C3 = cylinder(M(0,0,-3),7*vecK,r) -- petit cylindre plongé dans l'eau
8 -- sous la scène 3d

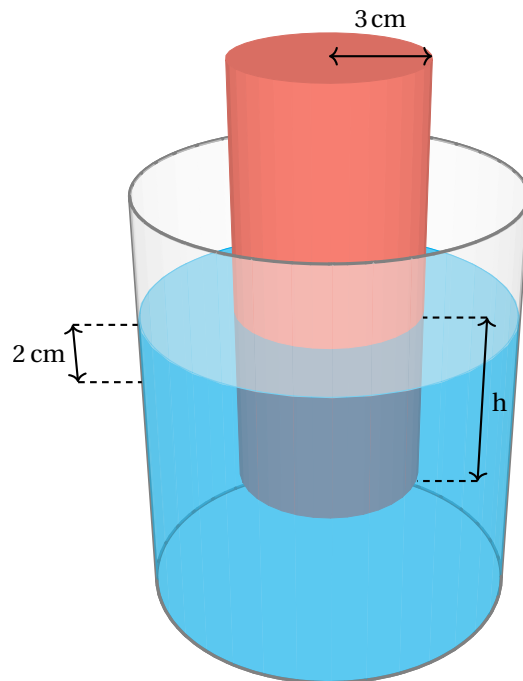
```

```

9  g:Lineoptions(nil,"gray",12)
10 g:Dcylinder(M(0,0,-5),7*vecK,R,{hiddenstyle="noline"}) -- contour du contenant (grand cylindre)
11 -- scène 3d
12 g:Dscene3d(
13     g:addPoly(C1,{contrast=0.125,color="cyan",opacity=0.5}), -- eau
14     g:addPoly(C2,{contrast=0.125,color="WhiteSmoke", opacity=0.5}), -- partie du contenant au-dessus de l'eau
15     g:addPoly(C3,{contrast=0.25,color="Salmon",backcull=true}), -- petit cylindre dans l'eau
16     g:addCircle(M(0,0,2),R,vecK,{color="gray"}), -- bord supérieur du contenant
17     g:addCircle(M(0,0,-5),R,vecK,{color="gray"}), -- bord inférieur du contenant
18     g:addCircle(Origin,R-0.025,vecK, {width=2,color="cyan"}) -- bord supérieur eau
19 )
20 -- par dessus la scène 3d
21 g:Lineoptions(nil,"black",8); A = 4*vecK; B = A+r*g:ScreenX()
22 g:Dpolyline3d( {A,B}, "<->"); g:Dlabel3d("$3\\$,cm", (A+B)/2, {pos="N", dist=0.25})
23 A = Origin+(r+1)*g:ScreenX(); A = rotate3d(A,-10,{Origin,vecK})
24 B = A-3*vecK
25 g:Dpolyline3d( {A,B}, "<->"); g:Dlabel3d("h", (A+B)/2, {pos="E"})
26 g:Lineoptions("dashed")
27 g:Dpolyline3d({{A,A-g:ScreenX()}},{B,B-g:ScreenX()})
28 A = Origin-(R+1)*g:ScreenX(); A = rotate3d(A,10,{Origin,vecK})
29 B = A-vecK
30 g:Dpolyline3d({{A,A+g:ScreenX()}},{B,B+g:ScreenX()})
31 g:Linestyle("solid")
32 g:Dpolyline3d( {A,B}, "<->"); g:Dlabel3d("$2\\$,cm", (A+B)/2, {pos="W"})
33 g:Show()
34 \end{luadraw}
35

```

FIGURE 26 : 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 i 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 à quatre champs, ces options sont :

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

### Ajouter des labels : `g:addLabels`

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 à 7 champs. Ces options sont :

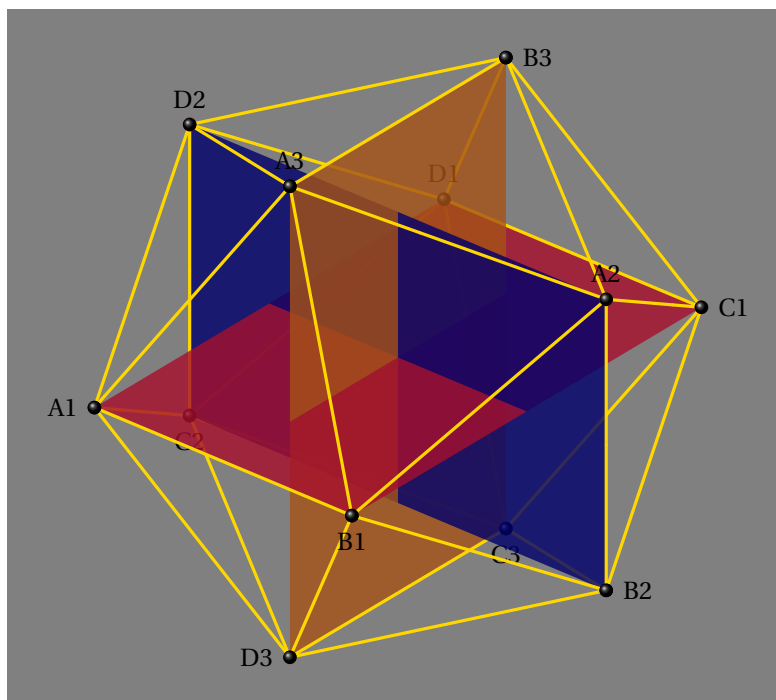
- `color` : chaîne définissant la couleur du label, initialisée à la couleur en cours des labels.
- `pos` : chaîne définissant la position du label par rapport au point d'ancrage (comme en 2d : "N", "NW", "W", ...), initialisée au style en cours des labels.
- `dist=0` : exprime la distance entre le label et son point d'ancrage (dans le plan de l'écran).
- `size` : chaîne définissant la taille du label, initialisée à la taille en cours des labels.
- `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=ID3d` : matrice 3d de transformation, par défaut celle-ci est la matrice 3d de l'identité, c'est à dire la table  $\{M(0,0,0), \text{vecI}, \text{vecJ}, \text{vecK}\}$ .

```

1 \begin{luadraw}{name=icosaedre}
2 local g = graph3d:new{window={-2.25,2.25,-2,2}, viewdir={40,60},bg="gray",size={10,10},margin={0,0,0,0}}
3 Hiddenlines = false
4 local phi = (1+math.sqrt(5))/2 -- nombre d'or
5 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
6 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
7 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
8 local ico = {
9     {A1,B1,A3}, {B1,A1,D3}, {D1,C1,C3}, {C1,D1,B3},
10    {B2,A2,B1}, {A2,B2,C1}, {D2,C2,A1}, {C2,D2,D1},
11    {B3,A3,A2}, {A3,B3,D2}, {D3,C3,B2}, {C3,D3,C2},
12    {A1,A3,D2}, {B1,A2,A3}, {A2,C1,B3}, {D1,D2,B3},
13    {B2,B1,D3}, {A1,C2,D3}, {B2,C3,C1}, {C2,D1,C3} }
14 g:Dscene3d(
15     g:addFacet({A2,B2,C2,D2},{color="Navy",twoside=false,opacity=0.8}),
16     g:addFacet({A1,B1,C1,D1},{color="Crimson",twoside=false,opacity=0.8}),
17     g:addFacet({A3,B3,C3,D3},{color="Chocolate",twoside=false,opacity=0.8}),
18     g:addPolyline(facetedges(ico), {color="Gold",width=12}), -- dessin des arêtes uniquement
19     g:addDots({A1,B1,C1,D1,A2,B2,C2,D2,A3,B3,C3,D3}, {color="black",scale=1.2}),
20     g:addLabel("A1",A1,{style="W",dist=0.1}, "B1",B1,{style="S"}, "C2",C2,{}, "C3",C3,{}, "A3",A3,{style="N"},
21     "D1",D1,{}, "A2",A2,{}, "D2",D2,{}, "B3",B3,{style="E"}, "C1",C1,{}, "B2",B2,{}, "D3",D3,{style="W"} )
22 )
23 g:Show()
24 \end{luadraw}

```

FIGURE 27 : 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 `C` est soit un plan, soit une liste de plans, soit une facette, soit une liste de facettes. L'argument *options* est une table. La seule option disponible est

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

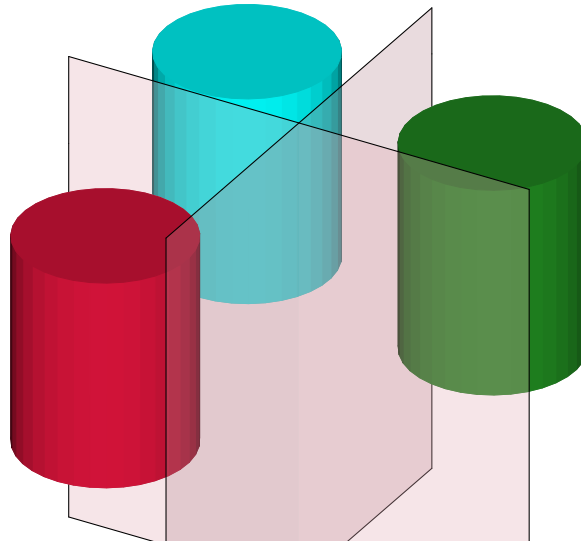
Dans l'exemple suivant les deux cloisons séparatrices ont été dessinées afin de les visualiser, mais normalement elles sont invisibles :

```

1 \begin{luadraw}{name=addWall}
2 local g = graph3d:new{size={10,10},window={-8,8,-4,8}, margin={0,0,0,0}}
3 local C = cylinder(M(0,0,-1),5*vecK,2)
4 g:Dscene3d(
5   g:addWall( {{0Origin,vecI}}, {{0Origin,vecJ}}),
6   g:addPlane({0Origin,vecI}, {color="Pink",opacity=0.3,scale=1.125,edge=true}), -- to show the first wall
7   g:addPlane({0Origin,vecJ}, {color="Pink",opacity=0.3,scale=1.125,edge=true}), -- to show the second wall
8   g:addPoly( shift3d(C,M(-3,-3,1)), {color="Cyan"} ),
9   g:addPoly( shift3d(C,M(-3,3,0.5)), {color="ForestGreen"} ),
10  g:addPoly( shift3d(C,M(3,-3,-0.5)), {color="Crimson"} )
11 )
12 g:Show()
13 \end{luadraw}

```

FIGURE 28 : 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.

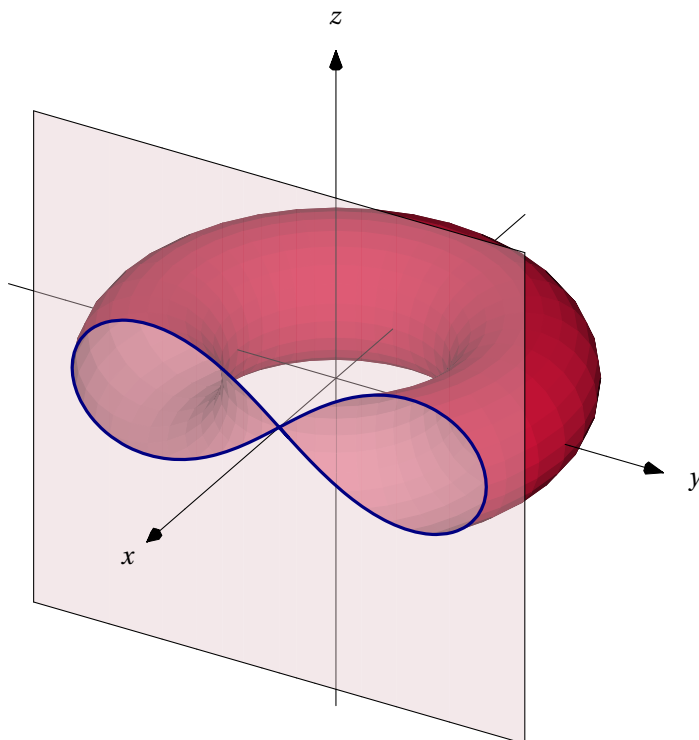
Voici un autre exemple bien plus probant où l'utilisation de cloisons séparatrices est indispensable pour avoir un dessin de taille raisonnable. Il s'agit de l'obtention d'une lemniscate comme intersection d'un tore avec un certain plan. Le tore étant non convexe le nombre de découpage inutile de facettes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

```

1  \begin{luadraw}{name=torus}
2  local g = graph3d:new{size={10,10}, margin={0,0,0,0}}
3  local cos, sin, pi = math.cos, math.sin, math.pi
4  local R, r = 2.5, 1
5  local x0 = R-r
6  local f = function(t) return M(0,R+r*cos(t),r*sin(t)) end
7  local plan = {M(x0,0,0),-vecI} -- plan dont la section avec le tore donne la lemniscate
8  local C, wall = rotcurve(f,-pi,pi,{Origin,vecK},360,0,{grid={25,37},addwall=2})
9  local C1 = cutfacet(C,plan) -- partie du tore dans le demi espace contenant -vecI
10 g:Dscene3d(
11   g:addWall(plan), g:addWall(wall), -- ajout de cloisons séparatrices
12   g:addFacet( C1, {color="Crimson", backcull=false}),
13   g:addPlane(plan, {color="Pink",opacity=0.4,edge=true}), -- plan de coupe
14   g:addAxes( Origin, {arrows=1})
15 )
16 -- équation cartésienne du tore : (x^2+y^2+z^2+R^2-r^2)^2-4*R^2*(x^2+y^2) = 0
17 -- 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)
18 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
19 local I = implicit(h,-4,4,-3,3,{50,50}) -- ligne polygonale 2d (liste de listes de complexes)
20 local lemniscate = map(function(z) return M(x0,z.re,z.im) end, I[1]) -- conversion en coordonnées spatiales
21 g:Dpolyline3d(lemniscate,"Navy,line width=1.2pt")
22 g:Show()
23 \end{luadraw}

```

FIGURE 29 : 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 *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 :

```
1 g:Dfacet(C1, {mode=mShadedOnly,color="Crimson"} )
2 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 :

```
1 \begin{luadraw}{name=section_sphere}
2 local g = graph3d:new{ window3d={-4,4,-4,4,-4,4}, window={-5.5,5.5,-4,5}, viewdir={30,75}, size={10,10}}
3 local O, R = Origin, 2.5 -- center et rayon
4 local S, P = sphere(O,R), {M(0,0,1.5),vecK+vecJ/2} -- la sphère et le plan de coupe
5 local w, n = pt3d.normalize(P[2]), g.Normal -- vecteurs unitaires normaux à P pour w et à l'écran pour n
6 local I, r = interPS(P,{O,R}) -- centre et rayon du petit cercle (intersection entre le plan et la sphère)
7 local C = g:Intersection3d(S,P) -- C est une liste d'arêtes
8 local N = I-O
9 local J = I+r*pt3d.normalize(vecJ-vecK/2) -- un point sur le petit cercle
10 local a = R/pt3d.abs(N)
11 local A, B = O+a*N, O-a*N -- points d'intersection de l'axe (O,I) avec la sphère
12 local c1, alpha = Orange, 0.4
13 local coul = {c1[1]*alpha, c1[2]*alpha,c1[3]*alpha} -- pour simuler la transparence
14 g:Dhline( g:Proj3d({B,-N})) -- demi-droite (le point B est non visible)
15 g:Dsphere(O,R,{mode=mBorder,color="orange"})
16 g:Dline3d(A,B,"dotted") -- droite (A,B) en pointillés
```

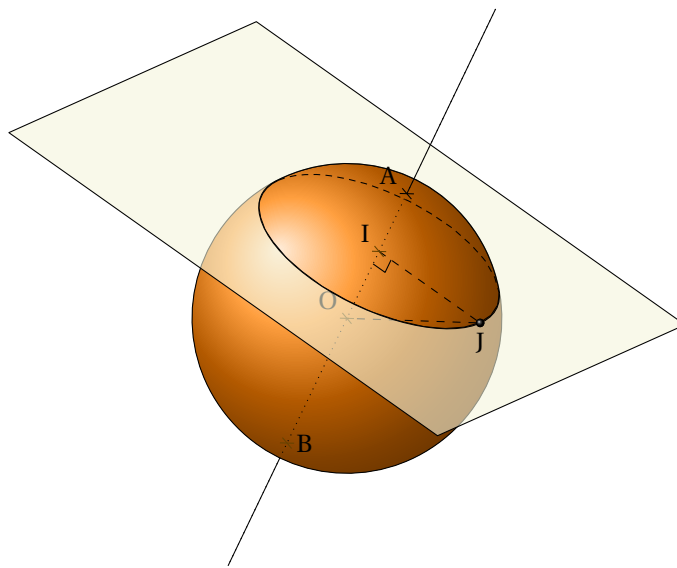


```

17 g:Dedges(C, {hidden=true,hiddenstyle="dashed"}) -- dessin de l'intersection
18 g:Dpolyline3d({I,J,O},"dashed")
19 g:Dangle3d(O,I,J) -- angle droit
20 g:Dcrossdots3d({B,N},{I,N},{O,N},rgb(coul),0.75) -- points dans la sphère
21 g:Dlabel3d("$O$", O, {pos="NW"})
22 local L = C.visible[1] -- partie visible de l'intersection (arc de cercle)
23 A1 = L[1]; A2 = L[#L] -- extrémités de L
24 local F = g:Plane2facet(P) -- plan converti en facette
25 -- plan troué sous forme de chemin 3d, le trou est le contour de la partie de la sphère au-dessus du plan
26 insert(F,{ "l", "c1", A1, "m", I, A2, r, -1, w, "ca", Origin, A1, R, -1, n, "ca" })
27 g:Dpath3d( F, "fill=Beige,fill opacity=0.6") -- dessin du plan troué
28 g:Dhline( g:Proj3d({A,N}) ) -- demi-droite, partie supérieure de l'axe (AB)
29 g:Dcrossdots3d({A,N}, "black", 0.75); g:Dballdots3d(J, "black", 0.75)
30 g:Dlabel3d("$A$", A, {pos="NW"}, "$I$", I, {}, "$B$", B, {pos="E"}, "$J$", J, {pos="S"})
31 g:Show()
32 \end{luadraw}

```

FIGURE 30 : 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 **circumcircle3d(A,B,C)**, où A, B et C sont trois points 3d non alignés, renvoie le cercle circonscrit au triangle formé par ces trois points, sous la forme d'une séquence : A, R, n, où A est le centre du cercle, R son rayon, et n un vecteur normal au plan du cercle.
- La fonction **incircle3d(A,B,C)**, où A, B et C sont trois points 3d non alignés, renvoie le cercle inscrit dans le triangle formé par ces trois points, sous la forme d'une séquence : A, R, n, où A est le centre du cercle, R son rayon, et n un vecteur normal au plan du cercle.

### 2) Enveloppe convexe : `cvx_hull3d()`

La fonction **cvx\_hull3d(L)** où L est une liste de points 3d **distincts**, calcule et renvoie l'enveloppe convexe de L sous la forme d'une liste de facettes.

```

1 \begin{luadraw}{name=cvx_hull3d}
2 local g = graph3d:new{window={-2,4,-6,1},bbox=false,size={10,10}}
3 local L = {Origin, 4*vecI, M(4,4,0), 4*vecJ}
4 insert(L, shift3d(L,-3*vecK))

```

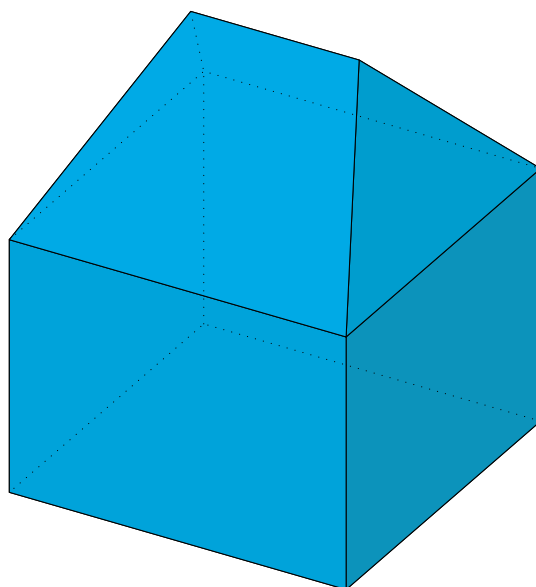


```

5 insert(L, {M(2,1,2), M(2,3,2)})
6 local V = cvx_hull3d(L)
7 local P = facet2poly(V)
8 g:Dpoly(P, {color="cyan", mode=mShadedHidden})
9 g:Show()
10 \end{luadraw}

```

FIGURE 31 : Utilisation de cvx\_hull3d()



**Cas particulier** : lorsque tous les points de  $L$  sont dans un même plan, on peut utiliser la fonction `cvx_hull3dcoplanar(L,n)` où  $n$  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 **`plane(A,B,C)`** envoie le plan passant par les trois points 3d  $A$ ,  $B$  et  $C$  (s'ils sont non alignés, sinon le résultat est *nil*).
- La fonction **`planeEq(a,b,c,d)`** envoie le plan dont une équation cartésienne est  $ax + by + cz + d = 0$  (si les coefficients  $a$ ,  $b$  et  $c$  ne sont pas tous nuls, sinon le résultat est *nil*).
- La fonction **`plane2ABC(P)`** où  $P = \{A, n\}$  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 **`orthoframe(P)`** où  $P = \{A, n\}$  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.

```

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

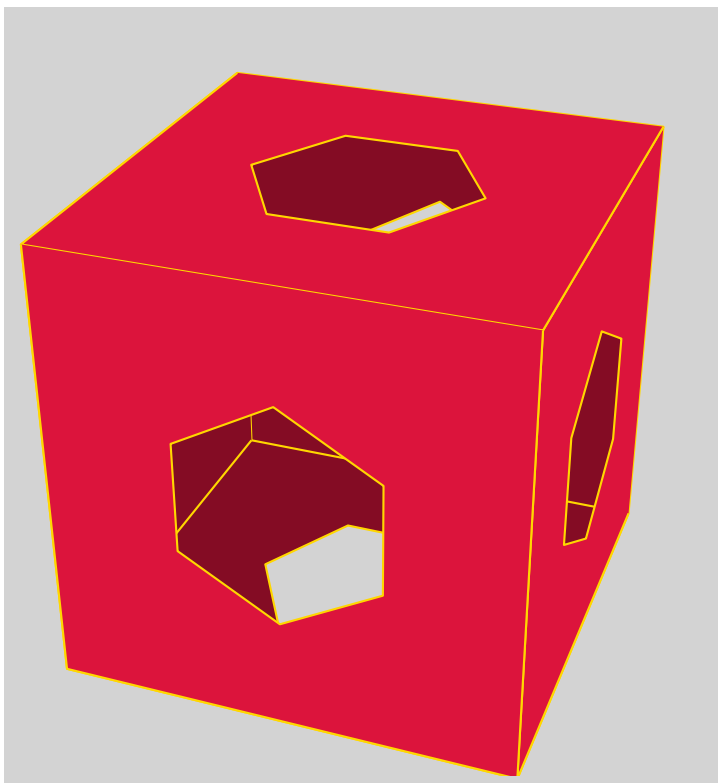
```

```

17 g:Dpath3d( concat(F,{"1"},p1,{"1","c1"}), "fill="..color ) -- dessin de la facette "trouée" avec l'hexagone
18 end
19 g:Show()
20 \end{luadraw}

```

FIGURE 32 : Faces d'un cube trouées avec un hexagone régulier



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

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

#### 5) Tétraèdre à longueurs fixées : `tetra_len()`

La fonction **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  $AB = ab$ ,  $AC = ac$ ,  $AD = ad$ ,  $BC = bc$ ,  $BD = bd$  et  $CD = cd$ . La fonction renvoie la séquence de quatre points A, B, C, D. Le sommet A est toujours le point  $M(0,0,0)$  (*Origin*) et le sommet B est toujours le point  $ab \cdot \text{vecI}$  et le sommet C dans le plan  $xOy$ . Le tétraèdre en tant que polyèdre peut ensuite être construit avec la fonction **tetra(A,B-A,C-A,D-A)**.

```

1 \begin{luadraw}{name=tetra_len}
2 local g = graph3d:new{window={-4,4,-4,4},margin={0,0,0,0},viewdir={25,65},size={10,10}}
3 Hiddenlines = true; Hiddenlinestyle = "dashed"
4 require 'luadraw_spherical'
5 local R = 4
6 local A,B,C,D = tetra_len(R,R,R,R,R,R)
7 local T = tetra(A,B-A,C-A,D-A)
8 g:Define_sphere({radius=R})
9 g:DSpolyline( facetedges(T), {color="DarkGreen"})
10 g:DSbigcircle( {B,C},{color="Blue"} )
11 g:DSbigcircle( {B,D},{color="Blue"} )
12 g:DSbigcircle( {C,D},{color="Blue"} )

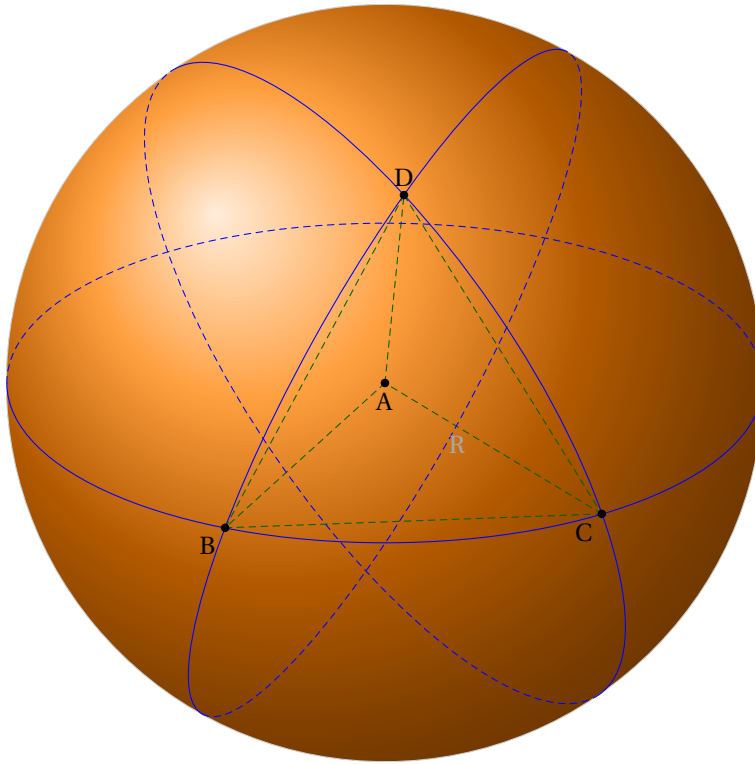
```

```

13 g:DLabel("$R$", (2*A+C)/3, {pos="S"})
14 g:Dspherical()
15 g:Ddots3d({A,B,C,D})
16 g:Dlabel3d("$A$", A, {pos="S"}, "$B$", B, {pos="SW"}, "$C$", C, {}, "$D$", D, {pos="N"})
17 g:Show()
18 \end{luadraw}

```

FIGURE 33 : 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 **`sss_triangle3d(ab, bc, ca)`** où *ab*, *bc* et *ca* 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  $AB = ab$ ,  $BC = bc$  et  $CA = ca$ , lorsque cela est possible. Le sommet A est toujours le point  $M(0, 0, 0)$  (*Origin*) et le sommet B est toujours le point  $ab \cdot \text{vecI}$ . Ce triangle peut être dessiné avec la méthode **`g:Dpolyline3d`**.
- La fonction **`sas_triangle3d(ab, alpha, ca)`** où *ab* et *ca* sont deux longueurs, *alpha* 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  $AB = ab$ ,  $CA = ca$ , et tel que l'angle  $(\vec{AB}, \vec{AC})$  a pour mesure *alpha*, lorsque cela est possible. Le sommet A est toujours le point  $M(0, 0, 0)$  (*Origin*) et le sommet B est toujours le point  $ab \cdot \text{vecI}$ . Ce triangle peut être dessiné avec la méthode **`g:Dpolyline3d`**.
- La fonction **`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  $M(0, 0, 0)$  (*Origin*) et le sommet B est toujours le point  $ab \cdot \text{vecI}$ . 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 **ftransform3d(L,f)** renvoie l'image de  $L$  par la fonction  $f$ , celle-ci doit être une fonction de  $\mathbf{R}^3$  vers  $\mathbf{R}^3$ .

### Projections : **proj3d**, **proj3dO**, **dproj3d**

- La fonction **proj3d(L,P)** renvoie l'image de  $L$  par la projection orthogonale sur le plan  $P$ .
- La fonction **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 **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 **pxy(L,z0)** renvoie l'image de  $L$  par la projection orthogonale sur le plan  $z = z_0$  (par défaut  $z_0 = 0$ ).
- La fonction **pyz(L,x0)** renvoie l'image de  $L$  par la projection orthogonale sur le plan  $x = x_0$  (par défaut  $x_0 = 0$ ).
- La fonction **pxz(L,y0)** renvoie l'image de  $L$  par la projection orthogonale sur le plan  $y = y_0$  (par défaut  $y_0 = 0$ ).
- La fonction **px(L)** renvoie l'image de  $L$  par la projection orthogonale sur l'axe  $Ox$ .
- La fonction **py(L)** renvoie l'image de  $L$  par la projection orthogonale sur l'axe  $Oy$ .
- La fonction **pz(L)** renvoie l'image de  $L$  par la projection orthogonale sur l'axe  $Oz$ .

### Symétries : **sym3d**, **sym3dO**, **dsym3d**, **psym3d**

- La fonction **sym3d(L,P)** renvoie l'image de  $L$  par la symétrie orthogonale par rapport au plan  $P$ .
- La fonction **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 **dsym3d(L,d)** renvoie l'image de  $L$  par la symétrie orthogonale par rapport la droite  $d$ .
- La fonction **psym3d(L,point)** renvoie l'image de  $L$  par la symétrie par rapport à *point* (point 3d).

### Rotation : **rotate3d**, **rotateaxe3d**

- La fonction **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 **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 *Origin*.

### Homothétie : **scale3d**

La fonction **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 vaut  $M(0,0,0)$  par défaut (origine).

### Inversion : **inv3d**

La fonction **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 vaut  $M(0,0,0)$  par défaut (origine).

### Stéréographie : **projstereo** et **inv\_projstereo**

Fonction **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  $I = C + h \frac{CN}{CN}$  (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 **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 **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) :

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

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

### applymatrix3d et applyLmatrix3d

- La fonction **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 **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 **composematrix3d(M1,M2)** effectue le produit matriciel  $M1 \times M2$  et renvoie le résultat.

### invmatrix3d

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

### matrix3dof

La fonction **matrix3dof(f)** calcule et renvoie la matrice de  $f$  (qui doit être une application affine de l'espace  $\mathbf{R}^3$ ).

### mtransform3d et mLtransform3d

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

## 3) Matrice associée au graphe 3d

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

```
1 local g = 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. Une réserve cependant : les méthodes *Dcylinder*, *Dcone* et *Dsphere* ne donnent le bon résultat qu'avec la matrice de transformation égale à l'identité. 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 *M* (avec *M* à droite) et le résultat est affecté à la matrice 3d du graphe. L'argument *M* 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 *L* et renvoie le résultat, l'argument *L* 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 *L* et renvoie le résultat. L'argument *L* 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 *angle* (en degrés) et d'axe *axe*.

**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 *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 **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 = clippolyline3d(L, g:Box3d())**

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

**clipline3d()**

La fonction **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 :

$$d = \text{clipline3d}(d, \text{g:Box3d}())$$

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

**cutpolyline3d()**

La fonction **cutpolyline3d(L, plane, close)** coupe la ligne polygonale 3d *L* avec le plan *plane*, si l'argument facultatif *close* 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 **getbounds3d(L)** renvoie 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 **interDP(d,P)** calcule et renvoie (si elle existe) l'intersection entre la droite *d* et le plan P.

**interPP()**

La fonction **interPP(P1,P2)** calcule et renvoie (si elle existe) l'intersection entre les plans P<sub>1</sub> et P<sub>2</sub>.

**interDD()**

La fonction **interDD(D1,D2,epsilon)** calcule et renvoie (si elle existe) l'intersection entre les droites D<sub>1</sub> et D<sub>2</sub>. L'argument *epsilon* vaut 10<sup>-10</sup> par défaut (sert à tester si un certain flottant est nul).

**interCS()**

La fonction **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 **interDS(d,S)** calcule et renvoie (si elle existe) l'intersection entre la droite *d* et la sphère S où S est une table S = {C, *r*} avec C le centre (point 3d) et *r* le rayon. La fonction renvoie soit *nil* (intersection vide), soit un seul point, soit deux points.

**interPS()**

La fonction **interPS(P,S)** calcule et renvoie (si elle existe) l'intersection entre le plan  $P$  et la sphère  $S$  où  $S$  est une table  $S = \{C, r\}$  avec  $C$  le centre (point 3d) et  $r$  le rayon. 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 **interSS(S1,S2)** calcule et renvoie (si elle existe) l'intersection entre la sphère  $S1 = \{C1, r1\}$  et  $S2 = \{C2, r2\}$ . 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 **interSSS(S1,S2,S3)** calcule et renvoie (si elle existe) l'intersection entre les sphères  $S1 = \{C1, r1\}$ ,  $S2 = \{C2, r2\}$  et  $S3 = \{C3, r3\}$ . La fonction renvoie soit *nil* (intersection vide), soit un seul point, soit deux points (séquence).

**merge3d()**

La fonction **merge3d(L)** 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.

**split\_points\_by\_visibility()**

La fonction **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*.

```

1  \begin{luadraw}{name=curve_on_cylinder}
2  local g = graph3d:new{adjust2d=true,bbox=false,size={10,10}, viewdir=perspective("central")}
3  g:Labelsize("footnotesize")
4  Hiddenlines = true; Hiddenlinestyle = "dashed"
5  local curve_on_cylinder = function(curve,cylinder)
6      -- curve is a 3d polyline on a cylinder,
7      -- cylinder = {A,r,V,B}
8      local A,r,V,B = table.unpack(cylinder)
9      if B == nil then B = V; V = B-A end
10     local U = B-A
11     local visible_function
12     if projection_mode == "central" then
13         visible_function = function(N)
14             local I = dproj3d(N,{A,U})
15             local M1, M2 = interCS({I,r,U},{ (I+camera)/2, pt3d.abs(I-camera)/2})
16             local alpha = angle3d(M1-I,camera-I)
17             return angle3d(N-I,camera-I) <= alpha
18         end
19     else
20         visible_function = function(N)
21             local I = dproj3d(N,{A,U})
22             return (pt3d.dot(N-I,g.Normal) >= 0)
23         end
24     end
25     return split_points_by_visibility(curve,visible_function)
26 end
27 -- test
28 local A, r, B = -5*vecJ, 4, 5*vecJ -- cylinder
29 local p = function(t) return Mc(r,t,t/3) end
30 local Curve = rotate3d( parametric3d(p,-4*math.pi,4*math.pi),90,{Origin,vecI})
31 local Vi, Hi = curve_on_cylinder(Curve,{A,r,B})

```

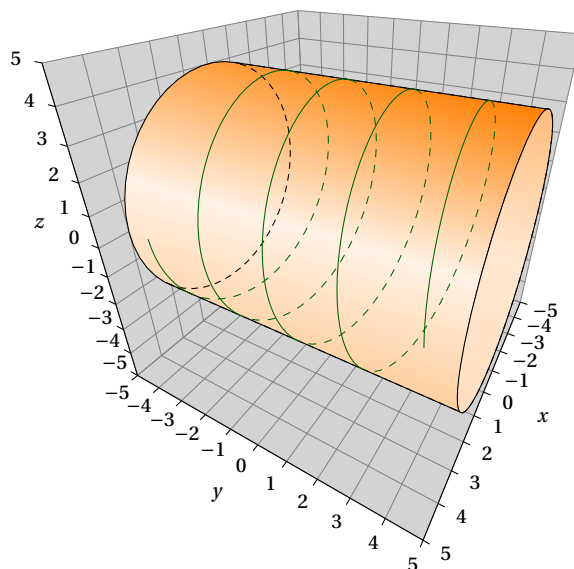


```

32 local curve_color = "DarkGreen"
33 g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
34 g:Dcylinder(A,r,B,{color="orange"})
35 g:Dpolyline3d(Vi,curve_color)
36 g:Dpolyline3d(Hi,curve_color.."","..Hiddenlinestyle)
37 g:Show()
38 \end{luadraw}

```

FIGURE 34 : Une courbe sur un cylindre



## VIII Exemples plus poussés

### 1) La boîte de sucre

Le problème<sup>7</sup> 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 boîte<sup>8</sup> 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 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<sup>9</sup>, 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.

```

1 \begin{luadraw}{name=boite_sucres}
2 local g = graph3d:new{window={-9,8,-10,4},size={10,10}}
3 Hiddenlines = false
4 local Lg, lg, ht = 5, 4, 3 -- longueur, largeur, hauteur (taille de la boîte)
5 local positions = {} -- matrice de dimension 3 initialisée avec des 0
6 for L = 1, Lg do
7   local X = {}
8   for l = 1, lg do

```

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

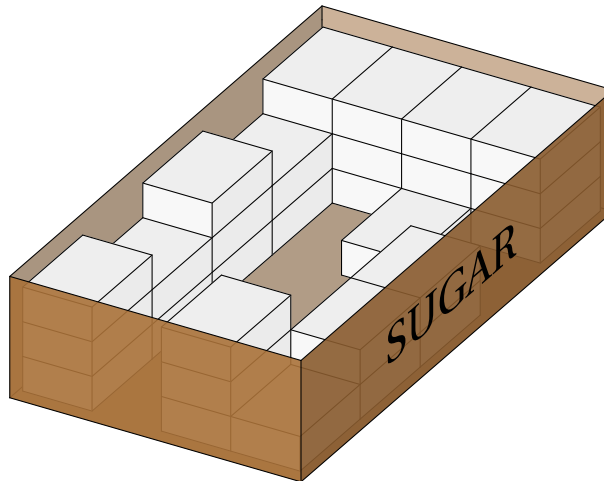
9. À condition de ne pas changer les angles de vue!

```

9      local Y = {}
10      for h = 1, ht do table.insert(Y,0) end
11      table.insert(X,Y)
12  end
13  table.insert(positions,X)
14 end
15 local facetList = function() -- renvoie la liste des facettes à dessiner (attention à l'orientation)
16     local facet = {}
17     for x = 1, Lg do -- parcours de la matrice positions
18         for y = 1, lg do
19             for z = 1, ht do
20                 if positions[x][y][z] == 1 then -- il y a un sucre en (x,y,z)
21                     if (z == ht) or (positions[x][y][z+1] == 0) then -- pas de sucre au-dessus donc face du dessus
22                         ↪ visible
23                         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
24                     end
25                     if (y == lg) or (positions[x][y+1][z] == 0) then -- pas de sucre à droite donc face de droite
26                         ↪ visible
27                         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
28                     end
29                     if (x == Lg) or (positions[x+1][y][z] == 0) then -- pas de sucre devant donc face de devant visible
30                         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
31                     end
32                 end
33             end
34         end
35     end
36     return facet
37 end
38 -- création de la boîte (parallélépipède)
39 local O = Origin -0.1*M(1,1,1) -- pour ne pas que la boîte soit collée aux sucres
40 local boite = parallelep(O, (Lg+0.2)*vecI, (lg+0.2)*vecJ, (ht+0.5)*vecK)
41 table.remove(boite.facets,2) -- on retire le dessus de la boîte, c'est la facette numéro 2
42 -- on positionne des sucres
43 for y = 1, 4 do for z = 1, 3 do positions[1][y][z] = 1 end end
44 for x = 2, 5 do for z = 1, 2 do positions[x][1][z] = 1 end end
45 for z = 1, 3 do positions[5][3][z] = 1 end
46 for z = 1, 2 do positions[4][4][z] = 1 end
47 for z = 1, 2 do positions[3][4][z] = 1 end
48 positions[5][1][3] = 1; positions[3][1][3] = 1; positions[5][4][1] = 1; positions[2][3][1] = 1
49 g:Setmatrix3d({Origin,3*vecI,2*vecJ,vecK}) -- dilatation sur Ox et Oy pour "allonger" les cubes ...
50 g:Dscene3d( -- dessin
51     g:addPoly(boite,{color="brown",edge=true,opacity=0.9}),
52     g:addFacet(facetList(), {backcull=true,contrast=0.25,edge=true})
53 )
54 g:Labelsize("huge"); g:Dlabel3d( "SUGAR", M(Lg/2+0.1,lg+0.1,ht/2+0.1), {dir={-vecI,vecK}})
55 g:Show()
56 \end{luadraw}

```

FIGURE 35 : 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 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).

```

1  \begin{luadraw}{name=cubes_empiles}
2  local g = graph3d:new{window3d={-6,6,-6,6,-6,6},size={10,10}}
3  Hiddenlines = false
4  local Lg, lg, ht, a = 5, 5, 5, 2 -- longueur, largeur, hauteur de l'espace à remplir, taille d'un cube
5  local positions = {} -- matrice de dimension 3 initialisée avec des 0
6  for L = 1, Lg do
7      local X = {}
8      for l = 1, lg do
9          local Y = {}
10         for h = 1, ht do table.insert(Y,0) end
11         table.insert(X,Y)
12     end
13     table.insert(positions,X)
14 end
15 for x = 1, Lg do -- positionnement aléatoire de cubes
16     for y = 1, lg do
17         local nb = math.random(0,ht) -- on met nb cubes dans la colonne (x,y,*) en partant du bas
18         for z = 1, nb do positions[x][y][z] = 1 end
19     end
20 end
21 local dessus,gauche,devant = {},{},{} -- pour mémoriser les facettes
22 for x = 1, Lg do -- parcours de la matrice positions pour déterminer les facettes à dessiner
23     for y = 1, lg do
24         for z = 1, ht do
25             if positions[x][y][z] == 1 then -- il y a un cube en (x,y,z)
26                 if (z == ht) or (positions[x][y][z+1] == 0) then -- pas de cube au-dessus donc face visible
27                     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
28                 end
29                 if (y == lg) or (positions[x][y+1][z] == 0) then -- pas de cube à droite donc face visible
30                     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
31                 end
32                 if (y == 1) or (positions[x][y-1][z] == 0) then -- pas de cube à gauche donc face visible

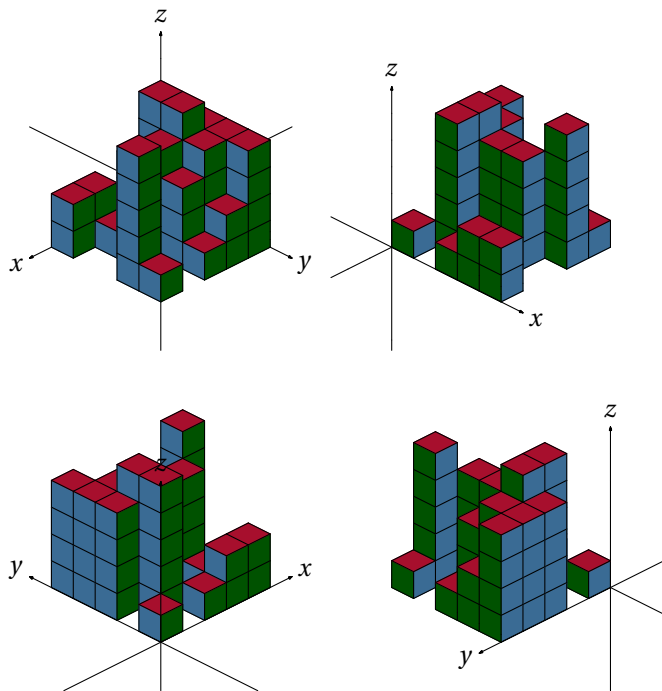
```

```

33     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
34     end
35     if (x == Lg) or (positions[x+1][y][z] == 0) then -- pas de cube devant donc face visible
36         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
37     end
38     if (x == 1) or (positions[x-1][y][z] == 0) then -- pas de cube derrière donc face de derrière visible
39         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)}) -- insertion face
40         -- arrière
41     end
42     end
43     end
44 end
45 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
46 local dessin = function()
47     g:Dscene3d(
48         g:addFacet(dessus, {backcull=true,color="Crimson"}), g:addFacet(gauche, {backcull=true,color="DarkGreen"}),
49         g:addFacet(devant, {backcull=true,color="SteelBlue"}),
50         g:addPolyline(facetedges(concat(dessus,gauche,devant))), -- dessin des arêtes
51         g:addAxes(Origin,{arrows=1})
52     end
53 g:Saveattr(); g:Viewport(-5,0,0,5); g:Coordsystem(-11,11,-11,11); g:Setviewdir(45,60) -- en haut à gauche
54 dessin(); g:Restoreattr()
55 g:Saveattr(); g:Viewport(0,5,0,5);g:Coordsystem(-11,11,-11,11); g:Setviewdir(-45,60) -- en haut à droite
56 dessin(); g:Restoreattr()
57 g:Saveattr(); g:Viewport(-5,0,-5,0);g:Coordsystem(-11,11,-11,11); g:Setviewdir(-135,60) -- en bas à gauche
58 dessin(); g:Restoreattr()
59 g:Saveattr(); g:Viewport(0,5,-5,0);g:Coordsystem(-11,11,-11,11); g:Setviewdir(135,60) -- en bas à droite
60 dessin(); g:Restoreattr()
61 g:Show()
62 \end{luadraw}

```

FIGURE 36 : Empilement de cubes



### 3) Illustration du théorème de Dandelin

```

1 \begin{luadraw}{name=Dandelin}
2 local g = graph3d:new{window3d={-5,5,-5,5,-5,5}, window={-5,5,-5,6}, bg="lightgray",viewdir={-10,85}}
3 g:Linewidth(8)
4 local sqrt = math.sqrt

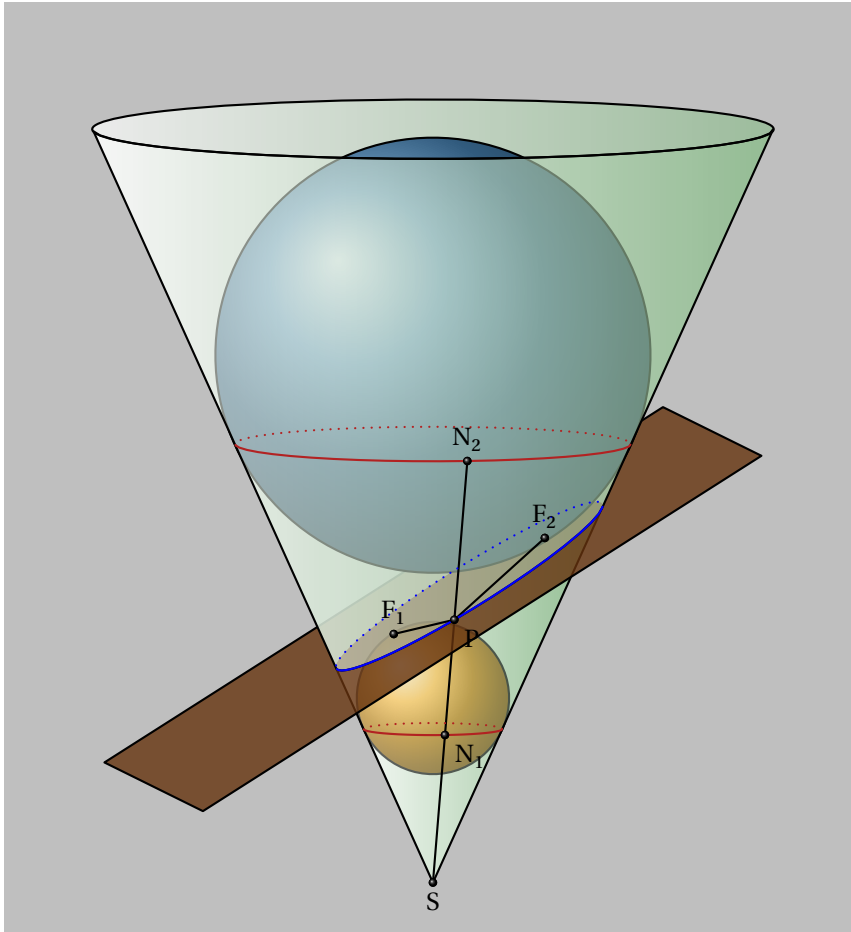
```

```

5 local sqr = function(x) return x*x end
6 local L, a = 4.5, 2
7 local R = (a+5)*L/sqrt(100+L^2) --grosse sphère centre=M(0,0,a) rayon=R
8 local S2 = sphere(M(0,0,a),R,45,45)
9 local k = 0.35 --rapport d'homothetie
10 local b, r = (a+5)*k-5, k*R -- petite sphère centre=M(0,0,b) rayon=r
11 local S1 = sphere(M(0,0,b),r,45,45)
12 local c = (b+k*a)/(1+k) --deuxieme centre d'homothetie
13 local z = a+sqr(R)/(c-a) --image de c par l'inversion par rapport à la grosse sphère
14 local M1 = M(0,sqr(sqr(R)-sqr(z-a)),z)--point de la grosse sphère et du plan tangent
15 local N = M1-M(0,0,a) -- vecteur normal au plan tangent
16 local plan = {M(0,0,c),-N} -- plan tangent
17 local z2 = a+sqr(R)/(-5-a) --image du sommet par l'inversion par rapport à la grosse sphère
18 local z1 = b+sqr(r)/(-5-b) -- image du sommet par l'inversion par rapport à la petite sphère
19 local P2 = M(sqrt(R^2-(z2-a)^2),0,z2)
20 local P1 = M(sqrt(r^2-(z1-b)^2),0,z1)
21 local S = M(0,0,-5)
22 local P = interDP({P1,P2-P1},plan)
23 local C = cone(M(0,0,-5),10*vecK,L,45,true)
24 local ellips = g:Intersection3d(C,plan)
25 local plan1 = {M(0,0,z1),vecK}
26 local plan2 = {M(0,0,z2),vecK}
27 local L1, L2 = g:Intersection3d(S1,plan1), g:Intersection3d(S2,plan2)
28 local F1, F2 = proj3d(M(0,0,b), plan), proj3d(M(0,0,a), plan) --foyers
29 local s1, s2 = g:Proj3d(M(0,0,a)), g:Proj3d(M(0,0,b))
30 local V, H = g:Classifyfacet(C) -- on sépare facettes visibles et les autres
31 local V1, V2 = cutfacet(V,plan)
32 local H1, H2 = cutfacet(H,plan)
33 -- Dessin
34 g:Dpolyline3d( border(H2),"left color=white, right color=DarkSeaGreen, draw=none" ) -- faces non visibles sous le plan,
  ↳ remplissage seulement
35 g:Dsphere( M(0,0,b), r, {mode=mBorder,color="Orange"}) -- petite sphère
36 g:Dpolyline3d( border(V2),"left color=white, right color=DarkSeaGreen, fill opacity=0.4" ) -- faces visibles sous le
  ↳ plan
37 g:Dpolyline3d({S,P}) -- segment [S,P] qui est sous le plan en partie
38 g:Dfacet( g:Plane2facet(plan,0.75), {color="Chocolate", opacity=0.8}) -- le plan
39 g:Dpolyline3d( border(H1),"left color=white, right color=DarkSeaGreen,draw=none,fill opacity=0.7" ) -- contour faces
  ↳ non visibles au dessus du plan, remplissage seulement
40 g:Dsphere( M(0,0,a),R, {mode=2,color="SteelBlue"}) -- grosse sphère
41 g:Dpolyline3d( border(V1),"left color=white, right color=DarkSeaGreen, fill opacity=0.6" ) -- contour faces visibles au
  ↳ dessus du plan
42 g:Dcircle3d(M(0,0,5),L,vecK) -- ouverture du cône
43 g:Dpolyline3d({{P,F1},{F2,P,P2}})
44 g:Dedges(L1,{hidden=true,color="FireBrick"})
45 g:Dedges(L2,{hidden=true,color="FireBrick"})
46 g:Dedges(ellips,{hidden=true, color="blue"})
47 g:Dballdots3d({F1,F2,S,P1,P,P2},nil,0.75)
48 g:Dlabel3d(
49   "$F_1$",F1,{pos="N"}, "$F_2$",F2,{}, "$N_2$",P2,{}, "$S$",S,{pos="S"}, "$N_1$",P1,{pos="SE"}, "$P$",P,{pos="SE"} )
50 g:Show()
51 \end{luadraw}

```

FIGURE 37 : 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

```

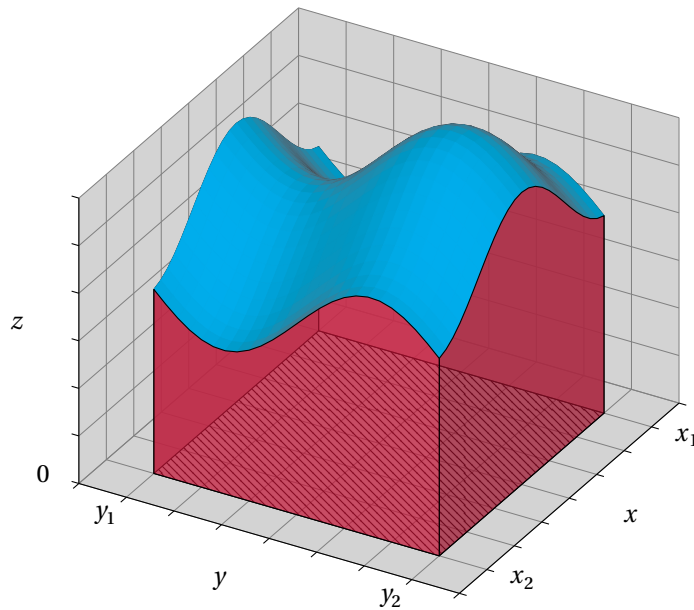
1 \begin{luadraw}{name=volume_integrale}
2 local i, pi, sin, cos = cpx.I, math.pi, math.sin, math.cos
3 local g = graph3d:new{window3d={-4,4,-4,4,0,6},adjust2d=true,margin={0,0,0,0},size={10,10}}
4 local x1, x2, y1, y2 = -3,3,-3,3 -- bornes
5 local f = function(x,y) return cos(x)+sin(y)+5 end -- fonction à intégrer
6 local p = function(u,v) return M(u,v,f(u,v)) end -- paramétrage surface z=f(x,y)
7 local Fx1 = concat({pxy(p(x1,y2)), pxy(p(x1,y1))}, parametric3d(function(t) return p(x1,t) end,y1,y2,25,false,0)[1])

```

```

8 local Fx2 = concat({pxy(p(x2,y1)), pxy(p(x2,y2))}, parametric3d(function(t) return p(x2,t) end,y2,y1,25,false,0)[1])
9 local Fy1 = concat({pxy(p(x1,y1)), pxy(p(x2,y1))}, parametric3d(function(t) return p(t,y1) end,x2,x1,25,false,0)[1])
10 local Fy2 = concat({pxy(p(x2,y2)), pxy(p(x1,y2))}, parametric3d(function(t) return p(t,y2) end,x1,x2,25,false,0)[1])
11 g:Dboxaxes3d({grid=true, gridcolor="gray",fillcolor="LightGray",labels=false})
12 g:Filloptions("fdiag","black"); g:Dpolyline3d( {M(x1,y1,0),M(x1,y2,0),M(x2,y2,0),M(x2,y1,0)} } -- dessous
13 g:Dfacet( {Fx1,Fy1},{mode=mShaded,opacity=0.7,color="Crimson"} )
14 g:Dfacet(surface(p,x1,x2,y1,y2), {mode=mShadedOnly,color="cyan"})
15 g:Dfacet( {Fx2,Fy2},{mode=mShaded,opacity=0.7,color="Crimson"} )
16 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),{ })
17 g:Show()
18 \end{luadraw}

```

FIGURE 38 : Volume correspondant à  $\int_{x_1}^{x_2} \int_{y_1}^{y_2} f(x,y) dx dy$ 

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 sûr 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

```

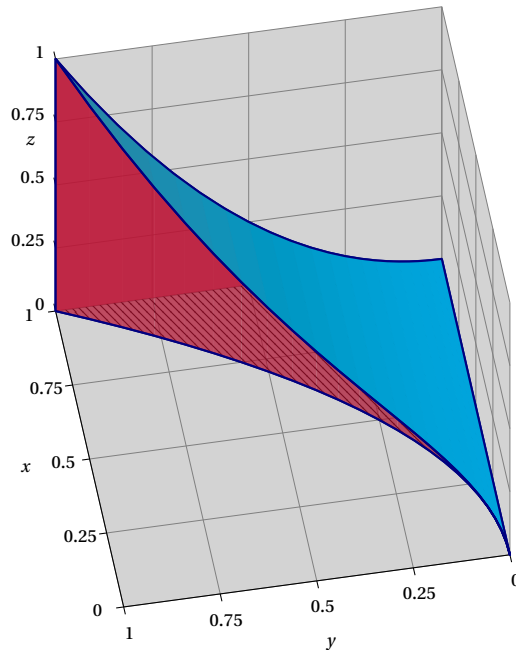
1 \begin{luadraw}{name=volume2}
2 local i = cpx.I
3 local g = graph3d:new{window3d={0,1,0,1,0,1}, margin={0,0,0,0},adjust2d=true,viewdir={170,40}, size={10,10}}
4 g:Labelsize("scriptsize")
5 local f = function(t) return M(t,t^2,0) end
6 local h = function(t) return M(1,t,t^2) end
7 local C = parametric3d(f,0,1,25,false,0)[1] -- courbe y=x^2 dans le plan z=0 (première composante connexe)
8 local D = parametric3d(h,1,0,25,false,0)[1] -- courbe z=y^2 dans le plan x=1, en sens inverse
9 local dessous = concat({M(1,0,0)},C) -- forme la face du dessous
10 local arriere = concat({M(1,1,0)},D) -- forme la face arrière
11 local avant, dessous, A, B = {}, {}, nil, C[1]
12 for k = 2, #C do --on construit les faces avant et de dessous facette par facette, en partant des points de C
13   A = B; B = C[k]
14   table.insert(avant, {B,A,M(A.x,A.y,A.y^2),M(B.x,B.y,B.y^2)})
15   table.insert(dessous, {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)})
16 end

```

```

17 g:Dboxaxes3d({grid=true, gridcolor="gray",fillcolor="LightGray", drawbox=false,
18   xyzstep=0.25, xlabelstyle="W",ylabelstyle="W",zlabelsep=0})
19 g:Lineoptions(nil,"Navy",8)
20 g:Dpolyline3d(arriere,close,"fill=Crimson, fill opacity=0.6") -- face arriere (plane)
21 g:Filloptions("fdiag","black"); g:Dpolyline3d(dessous,close) -- dessous
22 g:Dmixfacet(avant,{color="Crimson",opacity=0.7,mode=mShadedOnly}, dessus,{color="cyan",opacity=1})
23 g:Filloptions("none"); g:Dpolyline3d(concat(border(avant),border(dessous)))
24 g:Show()
25 \end{luadraw}

```

FIGURE 39 : Volume :  $0 \leq x \leq 1$ ;  $0 \leq y \leq x^2$ ;  $0 \leq z \leq y^2$ 

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  $t \mapsto M(t, t^2, t^4)$  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  $t \mapsto M(t, t^2, t^4)$ .

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) :

```

1 g:Dfacet( surface(function(u,v) return M(u,v*u^2,v^2*u^4) end, 0,1,0,1), {mode=mShadedOnly, color="cyan"})

```

mais elle comporte bien plus de facettes ( $25 \times 25$ ) que la construction sous forme de cylindre (21 facettes), ce qui est moins intéressant.



# 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](#).

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 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.

L'argument *all* 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 deux 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.

```

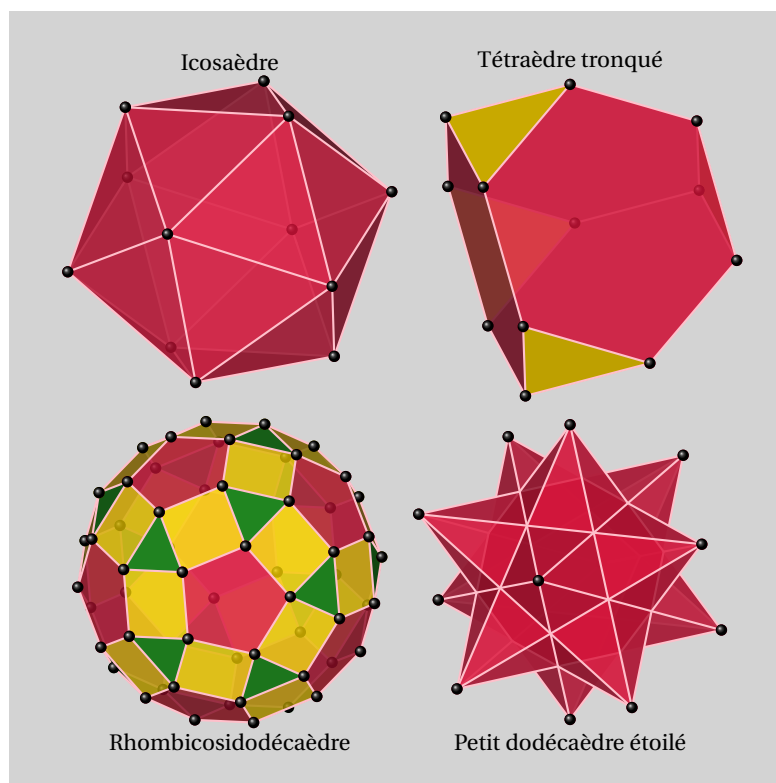
1  \begin{luadraw}{name=polyhedrons}
2  local i = cpx.I
3  require 'luadraw_polyhedrons' -- chargement du module
4  local g = graph3d:new{bg="LightGray", size={10,10}}
5  g:Labelsize("small"); Hiddenlines = false
6  -- en haut à gauche
7  g:Saveattr(); g:Viewport(-5,0,0,5); g:Coordsystem(-5,5,-5,5,true)
8  local T,S,A,F = icosahedron(Origin,M(0,2,4.5),true)
9  g:Dscene3d(
10     g:addFacet(F, {color="Crimson",opacity=0.8}),
11     g:addPolyline(A, {color="Pink", width=8}),
12     g:addDots(S) )
13  g:Dlabel("Icosaèdre",5*i,{})
14  g:Restoreattr()
15  -- en haut à droite
16  g:Saveattr()
17  g:Viewport(0,5,0,5); g:Coordsystem(-5,5,-5,5,true)
18  local T,S,A,F1,F2 = truncatedtetrahedron(Origin,M(0,0,5),true) -- sortie complète, affichage dans une scène 3d
19  g:Dscene3d(
20     g:addFacet(F1, {color="Crimson",opacity=0.8}),
21     g:addFacet(F2, {color="Gold"}),
22     g:addPolyline(A, {color="Pink", width=8}),
23     g:addDots(S) )
24  g:Dlabel("Tétraèdre tronqué",5*i,{})
25  g:Restoreattr()
26  -- en bas à gauche
27  g:Saveattr(); g:Viewport(-5,0,-5,0); g:Coordsystem(-5,5,-5,5,true)
28  local T,S,A,F1,F2,F3 = rhombicosidodecahedron(Origin,M(0,0,4.5),true)
29  g:Dscene3d(
30     g:addFacet(F1, {color="Crimson",opacity=0.8}),
31     g:addFacet(F2, {color="Gold",opacity=0.8}), g:addFacet(F3, {color="ForestGreen"}),
32     g:addPolyline(A, {color="Pink", width=8}), g:addDots(S) )

```

```

33 g:Dlabel("Rhombicosidodécaèdre",-5*i,{})
34 g:Restoreattr()
35 -- en bas à droite
36 g:Saveattr(); g:Viewport(0,5,-5,0); g:Coordsystem(-5,5,-5,5,true)
37 local T,S,A,F1 = small_stellated_dodecahedron(Origin,M(0,0,5),true)
38 g:Dscene3d(
39     g:addFacet(F1, {color="Crimson",opacity=0.8}),
40     g:addPolyline(A, {color="Pink", width=8}),
41     g:addDots(S) )
42 g:Dlabel("Petit dodécaèdre étoilé",-5*i,{})
43 g:Restoreattr()
44 g:Show()
45 \end{luadraw}
46

```

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.

### Variables et fonctions globales du module

- Variables avec leur valeur par défaut :
  - **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.
- Fonctions :

- **sm(x,y,z)** : 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  $(Mx, y, z)$  sur la sphère partant du centre.
- **sm(theta,phi)** : 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.
- **toSphere(A)** : renvoie le projeté du point A sur la sphère partant du centre.
- **clear\_spherical()** : supprime les objets qui ont été ajoutés à la scène, et remet les valeurs par défaut.

Si la variable globale **Hiddenlines** a la valeur *true*, alors les parties cachées seront dessinées dans le style défini par la variable globale **Hiddenlinestyle**, cependant on peut modifier ce comportement l'option locale *hidden=true/false*.

### Définition de la sphère

Par défaut, la sphère est centrée à l'origine, de rayon 3 et de couleur orange, mais ceci peut être modifié 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 entre parenthèses) :

- **center** = (Origin),
- **radius** = (3),
- **color** = ("Orange"),
- **opacity** = (1),
- **mode** = (*mBorder*), mode d'affichage de la sphère (*mWireframe* ou *mGrid* ou *mBorder*, voir **Dsphere**),
- **edgecolor** = ("LightGray"),
- **edgestyle** = ("solid"),
- **hiddenstyle** = (Hiddenlinestyle),
- **hiddencolor** = ("gray"),
- **edgewidth** = (4),
- **show** = (true), pour montrer ou non la sphère.

### Ajouter un cercle : g:DScircle

La méthode **g:DScircle(P,options)** permet d'ajouter un cercle sur la sphère, l'argument *P* 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 *options* est une table à 5 champs, qui sont :

- **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** = (valeur de *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.

### Ajouter un grand cercle : g:DSbigcircle

La méthode **g:DSbigcircle(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 à 5 champs, qui sont :

- **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** = (valeur de *Hiddenlines*),
- **out** = (nil), si on affecte une variable de type table 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.

### 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 à 6 champs, qui sont :

- **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** = (valeur de *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.

### 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 à 6 champs, qui sont :

- **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** = (valeur de *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 ABC lorsque ces trois points sont "alignés" sur un même grand cercle.

### Ajouter une facette sphérique : **g:DSfacet**

La méthode **g:DSfacet(F,options)** où *F* est une liste de points de la sphère, permet de dessiner la facette représentée par *F*, les arêtes étant des arcs de grands cercles. L'argument *options* est une table à 6 champs, qui sont :

- **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** = (valeur de *Hiddenlines*),
- **fill** = (""), chaîne représentant la couleur de remplissage (aucune par défaut),
- **filloppacity** = (0.3), opacité de la couleur de remplissage.

### Ajouter une courbe sphérique : **g:DScurve**

La méthode **g:DScurve(L,options)** où *L* est une liste de points de la sphère, permet de dessiner la courbe représentée par *L*. L'argument *options* est une table à 6 champs, qui sont :

- **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** = (valeur de *Hiddenlines*),
- **out** = (nil), si on affecte une variable de type table à ce paramètre *out*, alors la fonction ajoute à cette liste les points correspondant aux extrémités des parties cachées.

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 à 5 champs, qui sont :

- **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** = (valeur de *Hiddenlines*),
- **arrows** = (0), trois valeurs possibles : 0 (pas de flèche), 1 (une flèche en B), 2 (flèche en A et en B).

### 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* est 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 à 6 champs, qui sont :

- **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** = (valeur de *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.

### 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 à 6 champs, qui sont :

- **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** = (valeur de *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.

### 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 à 7 champs, qui sont :

- **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** = (valeur de *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.

### Ajouter un label : `g:DLabel`

La méthode `g:DLabel(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 la variable globale `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 à 2 champs, qui sont :

- `hidden = (valeur de 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 la variable globale `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 à 2 champs, qui sont :

- `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 = (valeur de Hiddenlines)`,
- `scale = (1)`, permet de changer la taille du parallélogramme,
- `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 précise par cette option.

Les points qui sont sur la face cachée de la sphère ont la couleur donnée par la variable globale `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  $z = \text{cte}$ , dessine sur la sphère l'image de  $L$  par stéréographie inverse, le pôle étant le point  $C+r*\text{vec}K$ , 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  $z = \text{cte}$ , dessine sur la sphère l'image de  $L$  par stéréographie inverse, le pôle étant le point  $C+r*\text{vec}K$ , 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`.

### Exemples

```

1 \begin{luadraw}{name=cube_in_sphere}
2 local g = graph3d:new{window={-9,9,-4,5},viewdir={25,70},size={16,8}}
3 require 'luadraw_spherical'
4 arrowBstyle = "-stealth"
5 g:Linewidth(6); Hiddenlinestyle = "dashed"
6 local a = 4
7 local O = Origin
8 local cube = parallelep(0,a*vecI,a*vecJ,a*vecK)

```

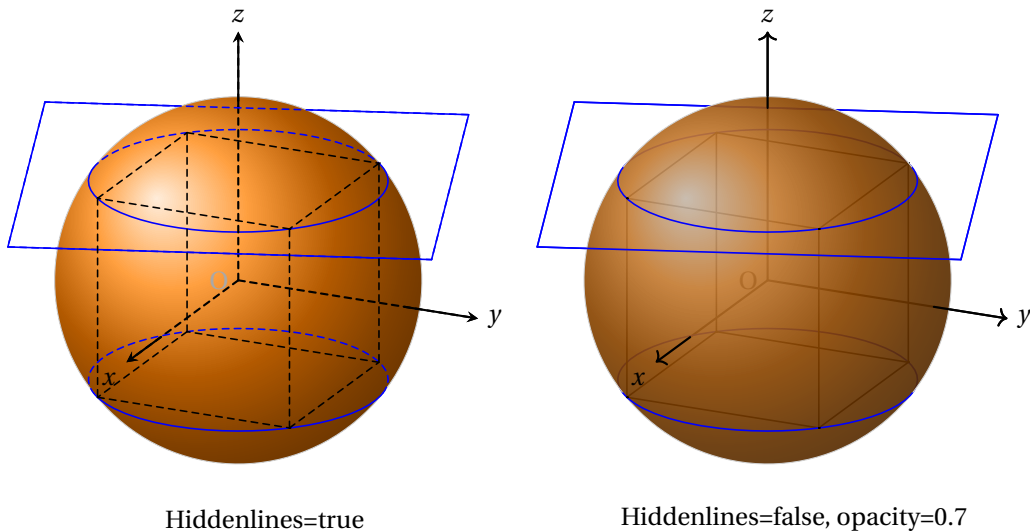


```

9 local G = isobar3d(cube.vertices)
10 cube = shift3d(cube,-G) -- pour centrer le cube à l'origine
11 local R = pt3d.abs(cube.vertices[1])
12
13 local dessin = function()
14   g:DSPolyline({{0,5*vecI},{0,5*vecJ},{0,5*vecK}}, {arrows=1, width=8}) -- axes
15   g:DSPlane({a/2*vecK,vecK},{color="blue",scale=0.9,angle=20});
16   g:DScircle({-a/2*vecK,vecK},{color="blue"})
17   g:DSPolyline(facetedges(cube)); g:DLabel("$0$",0,{pos="W"})
18   g:Dspherical()
19 end
20
21 g:Saveattr(); g:Viewport(-9,0,-4,5); g:Coordsystem(-5,5,-5,5)
22 Hiddenlines = true; g:Define_sphere({radius=R})
23 dessin()
24 g:Dlabel3d("$x$",5*vecI,{pos="SW"},"$y$",5*vecJ,{pos="E"},"$z$",5*vecK,{pos="N"})
25 g:Dlabel("Hiddenlines=true",0.5-4.5*cpx.I,{})
26 g:Restoreattr()
27
28 clear_spherical() -- supprime les objets précédemment créés
29
30 g:Saveattr(); g:Viewport(0,9,-4,5); g:Coordsystem(-5,5,-5,5)
31 Hiddenlines = false; g:Define_sphere({radius=R,opacity=0.7})
32 dessin()
33 g:Dlabel3d("$x$",5*vecI,{pos="SW"},"$y$",5*vecJ,{pos="E"},"$z$",5*vecK,{pos="N"})
34 g:Dlabel("Hiddenlines=false, opacity=0.7",0.5-4.5*cpx.I,{})
35 g:Restoreattr()
36 g:Show()
37 \end{luadraw}

```

FIGURE 2 : Cube dans une sphère



### Courbe sphérique

```

1 \begin{luadraw}{name=courbe_spherique}
2 local g = graph3d:new{window={-4.5,4.5,-4.5,4.5},viewdir={30,60},margin={0,0,0,0},size={10,10}}
3 require 'luadraw_spherical'
4 arrowBstyle = "-stealth"
5 g:Linewidth(6); Hiddenlinestyle = "dotted"
6 Hiddenlines = false;
7 local C = cylinder(M(1.5,0,-3.5),1.5,M(1.5,0,3.5),35,true)
8 local L = parametric3d( function(t) return Ms(3,t-math.pi/2,t) end, -math.pi,math.pi) -- la courbe
9 g:Define_sphere()
10 g:DSPolyline(facetedges(C),{color="gray"}) -- affichage cylindre
11 g:DSPolyline({{-5*vecI,5*vecI},{-5*vecJ,5*vecJ},{-5*vecK,5*vecK}}, {arrows=1}) -- axes

```

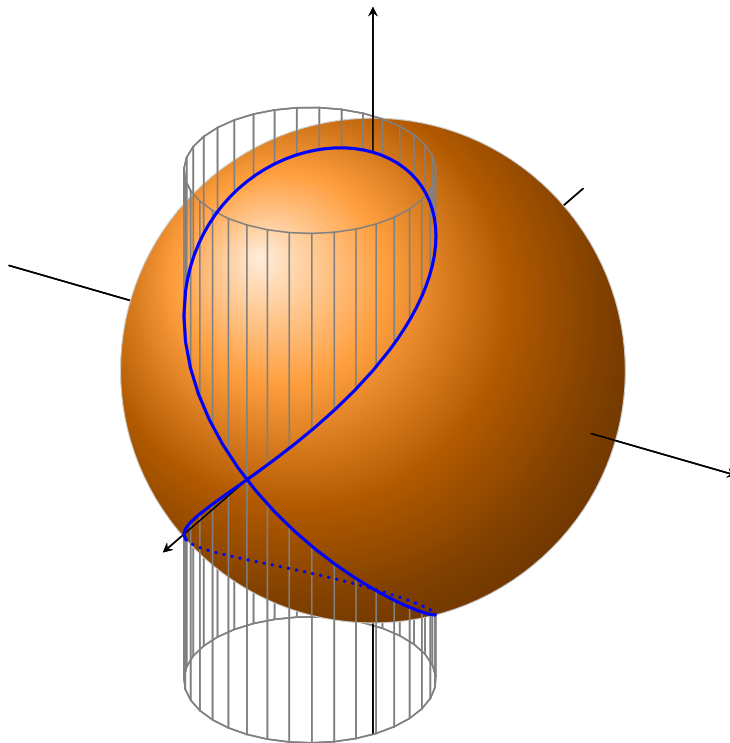


```

12 Hiddenlines=true; g:DScurve(L,{width=12,color="blue"}) -- courbe avec partie cachée
13 g:Dspherical()
14 g:Show()
15 \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.

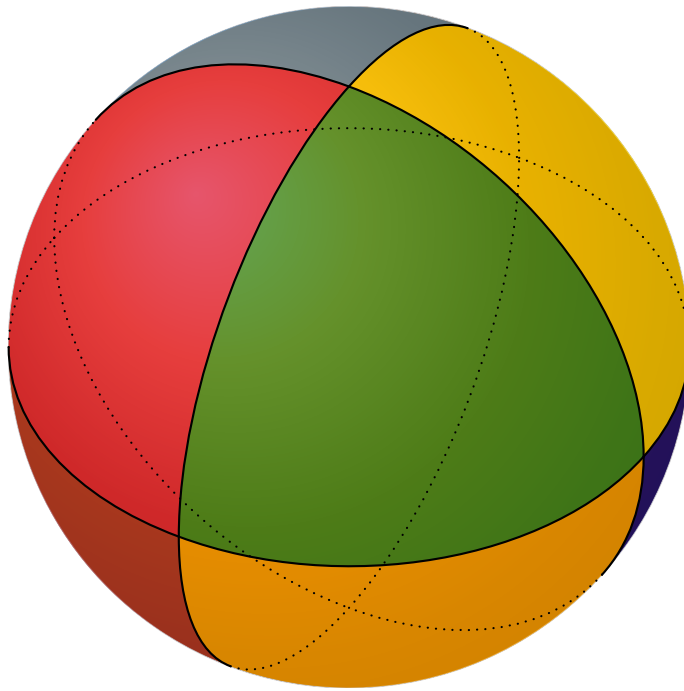
### Un pavage sphérique

```

1 \begin{luadraw}{name=pavage_spherique}
2 local g = graph3d:new{window={-3,3,-3,3},viewdir={30,60},size={10,10}}
3 require 'luadraw_spherical'
4 require "luadraw_polyhedrons"
5 g:Linewidth(6); Hiddenlines = true; Hiddenlinestyle = "dotted"
6 local P = poly2facet( octahedron(Origin,sM(30,10)) )
7 local colors = {"Crimson","ForestGreen","Gold","SteelBlue","SlateGray","Brown","Orange","Navy"}
8 g:Define_sphere()
9 for k,F in ipairs(P) do
10     g:DSfacet(F,{fill=colors[k],style="noline",fillopacity=0.7}) -- facettes sans les bords
11 end
12 for _, A in ipairs(facetedges(P)) do
13     g:DSarc(A,1,{width=8}) -- chaque arête est un arc de grand cercle
14 end
15 g:Dspherical()
16 g:Show()
17 \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).

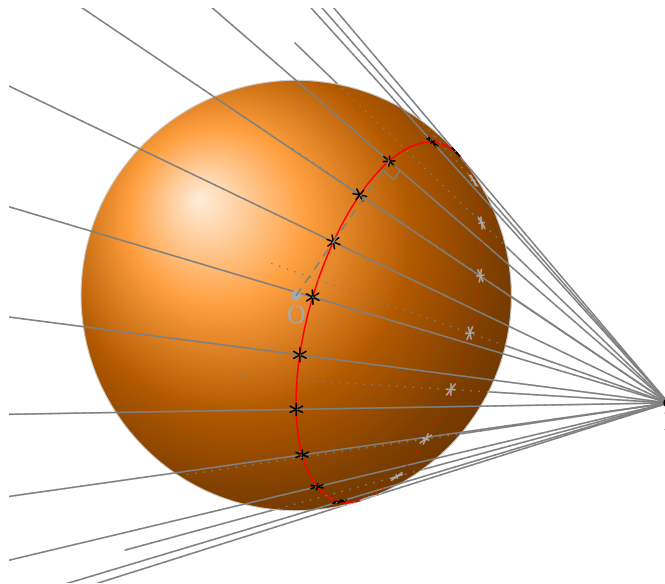
### Tangentes à la sphère issues d'un point

```

1 \begin{luadraw}{name=tangent_to_sphere}
2 local g = graph3d:new{window={-4,5.5,-4,4},viewdir={30,60},size={10,10}}
3 require 'luadraw_spherical'
4 Hiddenlines=true; g:Linewidth(6)
5 local O, I = Origin, M(0,6,0)
6 local S,S1 = {0, 3}, {(I+O)/2,pt3d.abs(I-O)/2}
7 -- the circle of tangency is the intersection between spheres S and S1
8 local C,r,n = interSS(S,S1)
9 local L = circle3d(C,r,n)[1] -- list of 3d points on the circle
10 local dots, lines = {}, {}
11 -- draw
12 g:Define_sphere({opacity=1})
13 g:DScircle({C,n},{color="red"})
14 for k = 1, math.floor(#L/4) do
15     local A = L[4*(k-1)+1]
16     table.insert(dots,A)
17     table.insert(lines,{I, 2*A-I})
18 end
19 g:DSPolyline(lines,{color="gray"})
20 g:DStars(dots) -- dessin de points sur la sphère
21 g:DSDots({O,I}); -- points dans la scène
22 g:DLabel("$I$",I,{pos="S",node_options="red"},"$O$",O,{})
23 g:Dspherical()
24 g:Dseg3d({O,dots[1]},"gray,dashed"); g:Dangle3d(O,dots[1],I,0.2,"gray")
25 g:Show()
26 \end{luadraw}

```

FIGURE 5 : Tangentes à la sphère issues d'un point

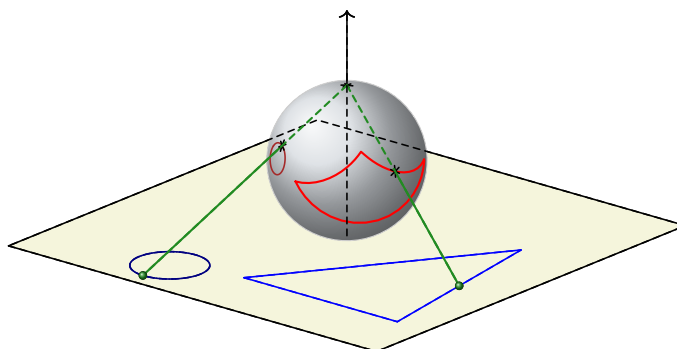


### Stéréographie inverse

```

1 \begin{luadraw}{name=stereographic_curve}
2 local g = graph3d:new{window3d={-5,5,-2,2},window={-4.25,4.25,-2.5,2},size={10,10}, viewdir={40,70}}
3 Hiddenlines = true; Hiddenlinestyle="dashed"; g:Linewidth(6)
4 require 'luadraw_spherical'
5 local C, R = Origin, 1
6 local a = -R
7 local P = planeEq(0,0,1,-a)
8 local L = {M(2,0,a), M(2,2.5,a), M(-1,2,a)}
9 local L2 = circle3d(M(2.25,-1,a),0.5,vecK)[1]
10 local A, B = (L[2]+L[3])/2, L2[20]
11 local a,b = table.unpack( inv_projstereo({A,B},{C,R},C+R*vecK) )
12 g:Dplane(P,vecJ,6,6,15,"draw=none,fill=Beige")
13 g:Define_sphere( {center=C,radius=R, color="SlateGray!30", show=true} )
14 g:DSpolyline(L,{color="blue",close=true}); g:DSinvstereo_polyline(L,{color="red",width=8,close=true})
15 g:DSpolyline(L2,{color="Navy"}); g:DSinvstereo_curve(L2,{color="Brown",width=6})
16 g:DSplane(P,{scale=1.5})
17 g:DSpolyline({{C+R*vecK,A},{C+R*vecK,B}}, {color="ForestGreen",width=8})
18 g:DSpolyline({{-vecK,2*vecK}}, {arrows=1})
19 g:DSstars({C+R*vecK,a,b}, {scale=0.75})
20 g:Dspherical()
21 g:Dballdots3d({A,B},"ForestGreen",0.75)
22 g:Show()
23 \end{luadraw}

```

FIGURE 6 : Méthodes *DSinvstereo\_curve* et *DSinvstereo\_polyline*

### 3) Le module *luadraw\_palettes*

Le module *luadraw\_palettes*<sup>1</sup> 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). Toutes les palettes ont pour préfixe "pal", la liste de ces palettes ainsi que leur rendu, peuvent être visualisés dans ce [document](#). L'extension fournit également la fonction *getPal(name,options)* dont voici un exemple d'utilisation :

```
1 BlackbodyTransformed = getPal(
2   "Blackbody", -- nom de la palette sans le préfixe pal
3   {
4     extract = {2, 5, 8, 9}, -- numéros des couleurs à extraire
5     shift = 1, -- décalage parmi les couleurs extraites, ce qui donne ici: 5,8,9,2
6     reverse = true -- inversion de l'ordre, ce qui donne ici: 2,9,8,5
7   }
8 )
```

### 4) Le module *luadraw\_compile\_tex*

**Attention** : ce module nécessite que soit installés les programmes *pdf2ps* et *pstoedit* sur votre système.

Ce module permet de :

1. compiler un fragment de texte en  $\text{\TeX}$ ,
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 **compile\_tex(text,filename)**, 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). Le processus se déroule en plusieurs étapes :

1. création du fichier tex. Celui-ci utilise deux variables globales qui sont :
 

```
preamble = "\\documentclass[12pt]{article}\n"
usepackage = "\\usepackage{amsmath,amssymb}\n\\usepackage{fourier}\n"
```

 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 *cachedir*), et tous les résidus de la compilation sont effacés.
5. Le contenu du fichier ainsi créé est automatiquement lu par la fonction *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*").

1. Ce module est une contribution de [Christophe BAL](#).

## Deuxième partie : exploitation du résultat

**En 2d** Le résultat peut être dessiné avec la méthode **g:Dcompiled\_tex(anchor,L,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 sont :

- **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), table constituée de deux vecteurs  $\{v1, v2\}$  indiquant le sens de l'écriture (nil signifie le sens habituel ce qui correspond à la table  $\{1, cpx.I\}$ ),
- **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 **compiled\_tex2polyline(L,scale)** où  $L$  est le résultat renvoyé par la fonction `compile_tex()`. L'argument optionnel *scale* permet de jouer sur la taille, ce peut être un nombre ou bien une table de deux nombres :  $\{scaleX, scaleY\}$ .

```

1 \begin{luadraw}{name=compile_tex2d}
2 local g = graph:new{bbox=false}
3 require 'luadraw_compile_tex'
4 local i = cpx.I
5 local text = "\\int_0^{+\\infty} e^{-\\frac{x^2}{2}}dx = \\frac{\\sqrt{2\\pi}}{2}" -- text to compile
6 local L = compile_tex(text,"gauss_integral") -- compile with -shell-escape the first time to create the
  ~ gauss_integral.eps file
7 g:Shift(2*i) -- a first drawing
8 g:Dcompiled_tex(0,L,{scale=2,hollow=true, drawbox=true, draw_options="fill=pink", dir={1-i/4,i}}) -- we draw L
9
10 g:Shift(-4*i) -- a second drawing
11 L = compiled_tex2polyline(L,{3,3}) -- L is converted to a polygonal line
12 local f = function(z) return Z(z.re,z.im+math.sin(z.re*1.5)) end -- this function produces sinusoidal waves
13 L = ftransform(L,f) -- we apply f to L
14 g:Dpath( polyline2path(L), 'draw=none,fill=blue') -- we draw L as a path
15 g:Show()
16 \end{luadraw}

```

FIGURE 7 : Exemple avec `compile_tex` en 2d

$$\int_0^{+\infty} e^{-\frac{x^2}{2}} dx = \frac{\sqrt{2\pi}}{2}$$

**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 `compile_tex()`. L'argument *options* est une table dont les champs sont :

- `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` = (Origin), point 3d qui représente le centre de la boîte englobante du dessin,
- `color` = (couleur courante par défaut),
- `dir` = ({vecJ,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 sont :

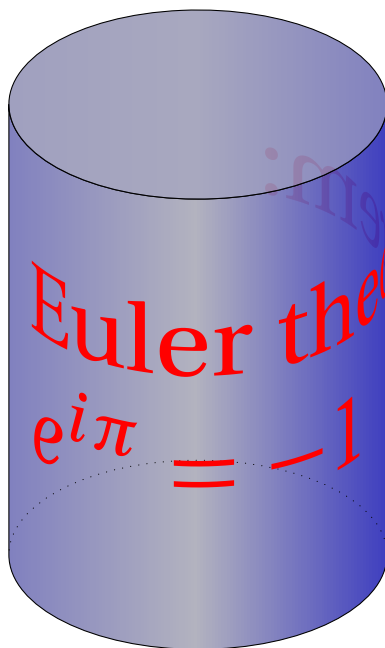
- `color` = (couleur courante par défaut),
- `hollow` = (false), permet d'activer ou désactiver le remplissages des formes. Avec la valeur *true* seuls les contours sont dessinés,
- `drawbox` = (false) : permet de dessiner ou non la boîte englobante,
- `draw_options` = ("") : chaîne contenant les options qui seront passées directement à la commande `\draw`.

```

1 \begin{luadraw}{name=compile_tex3d}
2 local g = graph3d:new{ window={-3,3,-4,4}, margin={0,0,0,0}, size={10,10}, viewdir={-50,60}}
3 require 'luadraw_compile_tex'
4
5 function curve_on_cylinder(curve,cylinder,screenNormal)
6 -- curve is a 3d polyline on a cylinder,
7 -- cylinder = {A,r,B}
8 -- this function separate the visible part from the hidden part of the curve
9 local A,r,B = table.unpack(cylinder)
10 local U = B-A
11 local visibility_function = function(N)
12 local I = dproj3d(N,{A,U})
13 return (pt3d.dot(N-I,screenNormal) >= 0)
14 end
15 return split_points_by_visibility(curve,visibility_function)
16 end
17
18 local A, r, B = -3*vecK, 2, 2.5*vecK -- the cylinder
19 local text = "Euler theorem: \par \((e^{i\pi}=-1)\)"
20 local L = compile_tex(text, "essai") -- compile with shell-escape the first time to create "essai.eps" file
21 local C = g:Compiled_tex2path3d(L,{scale=3, anchor=M(r,0,0), dir={vecJ,vecK}, polyline=true})
22 -- C is the text converted into 3d polylines, in the plane passing through anchor and basic direction dir, with a scale
23 -- of 3.
24
25 local f = function(A) return Mc(r,A.y/r,A.z) end -- returns the image of a point A on the cylinder by winding
26 C = ftransform3d(C,f) -- plane curve -> cylindrical curve transformation
27 local Cv, Ch = curve_on_cylinder(C, {A,r,B}, g.Normal) -- visible part and hidden part of C, this may take some time
28 Ch = polyline2path3d(Ch) -- hidden part, conversion to path
29 g:Dpath3d(Ch, "draw=none,fill=red!30") -- hidden part first
30 g:Dcylinder(A,r,B,{color="blue",opacity=0.5}) -- cylinder
31 Cv = polyline2path3d(Cv) -- visible part, conversion to path
32 g:Dpath3d(Cv, "draw=none,fill=red")
33 g:Show()
34 \end{luadraw}

```

FIGURE 8 : Écrire sur un cylindre



## 5) Le module *luadraw\_cvx\_polyhedra\_nets*

### 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 :

**unfold\_polyhedron(P, options)**

L'argument *P* doit être un polyèdre convexe. L'argument *options* est une table permettant d'ajuster certains paramètres. Ceux-ci sont les suivants (avec leur valeur par défaut entre parenthèses) :

- **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 :

$$\{ \{ \text{ancestor}, n1, n2, \text{angle}, \text{vertices} \}, \dots \}$$

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  $\{ \{ \{ a1, b1 \}, \{ a2, b2 \} \}, \dots \}$  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 *Pfacets*) ainsi que le numéro des arêtes afin de voir celles qui doivent être collées ensemble :

```

1 \begin{luadraw}{name=parallelep_net}
2 local g = graph3d:new{viewdir={30,60},window={-8.5,8,-5,5},bbox=false, size={10,10}}
3 require 'luadraw_cvx_polyhedra_nets'
4 P = parallelep(Origin, 4*vecI, 5*vecJ, 3*vecK)

```

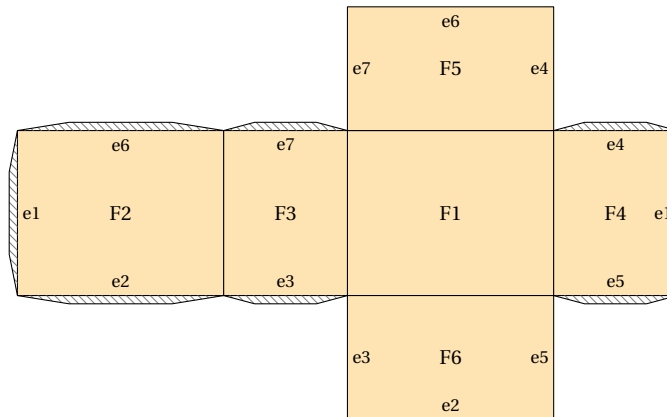


```

5 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})
6 g:Show()
7 \end{luadraw}

```

FIGURE 9 : Patron d'un parallélépipède



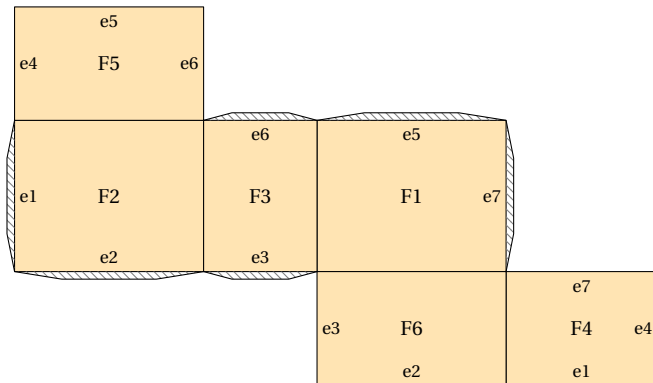
Le patron par défaut ici correspondrait à l'option  $model=\{\{1,3\},\{1,4\},\{1,5\},\{1,6\},\{3,2\}\}^2$ , mais on peut vouloir imposer un autre modèle, par exemple, avec le même parallélépipède :

```

1 \begin{luadraw}{name=parallelep_net2}
2 local g = graph3d:new{viewdir={30,60},window={-9,9,-5,5},bbox=false,size={10,10}}
3 require 'luadraw_cvx_polyhedra_nets'
4 P = parallelep(Origin, 4*vecI,5*vecJ,3*vecK)
5 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})
6 g:Show()
7 \end{luadraw}

```

FIGURE 10 : Patron imposé d'un parallélépipède



Voici un exemple avec un parallélépipède tronqué qui l'on déplie à moitié :

```

1 \begin{luadraw}{name=parallelep_net3}
2 local g = graph3d:new{window={-9,15,-9,9,0.6,0.6},bg="lightgray", viewdir={30,60}, margin={0,0,0,0}}
3 require 'luadraw_cvx_polyhedra_nets'
4 P = parallelep(Origin, 4*vecI,5*vecJ,3*vecK)
5 local A, B, C = M(4,2.5,3), M(2,5,3), M(4,5,1.5)
6 P = cutpoly(P, plane(A,B,C), true) -- P is truncated with a plane
7 g:Shift3d(M(0,-4,5))
8 g:Dpolynames(P,"facet") -- this function shows facet numbers of P
9 -- half unfolded P
10 g:Shift3d(M(0,0,-11))
11 g:Dpolyhedron_net(P,{opening=0.5, facet_options={color="Crimson", opacity=0.7, edgecolor="Gold", edgewidth=8}})
12 -- 2d net
13 g:Shift(10)
14 g:Dpolyhedron_net(P,{tabs=true, tabs_options="pattern=north west lines, pattern color=gray", facet_name=true,
  ↳ rotate=90})

```

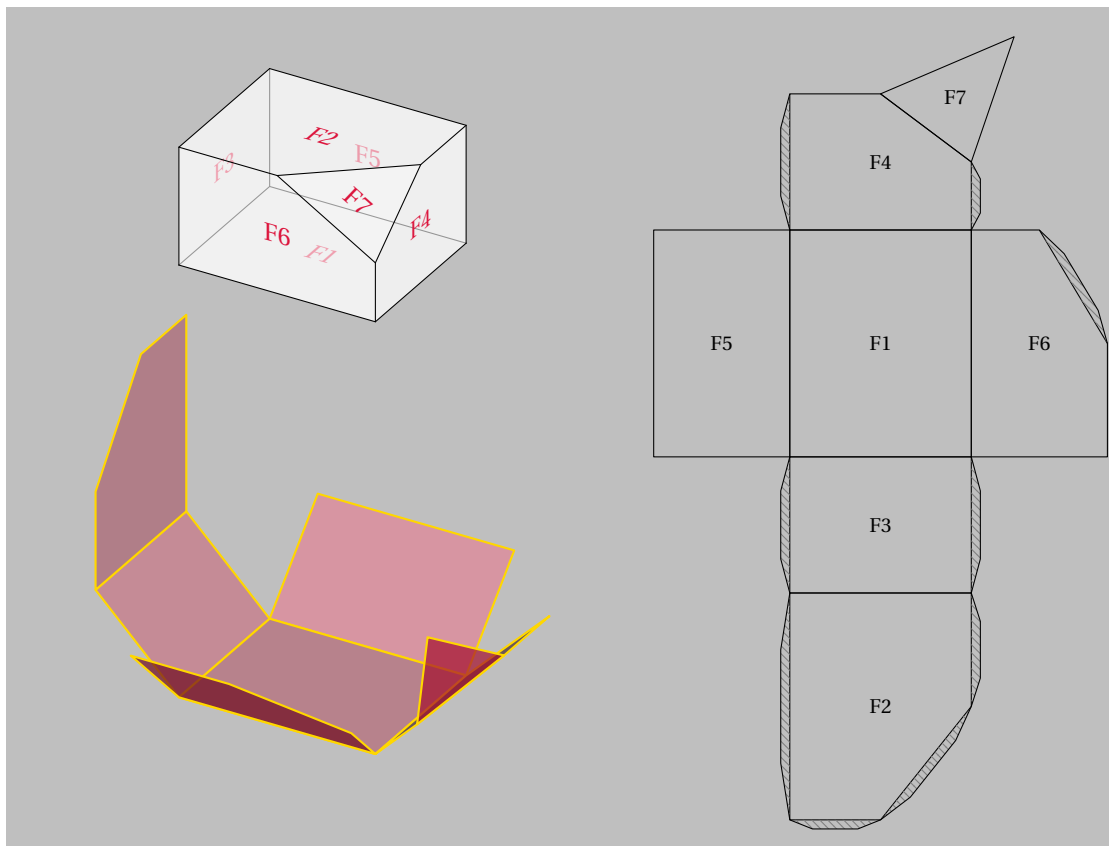
2. L'algorithme prend la première facette, puis cherche ses voisines, puis les voisines de la première voisine, etc.

```

15 g:Show()
16 \end{luadraw}

```

FIGURE 11 : Parallélépipède tronqué à demi déplié



**NB :** Les fonctions **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 **unfold\_polyhedron** afin d'éviter de recalculer celui-ci plusieurs fois, lors d'une animation par exemple. La fonction **unfold\_tree(tree,opening,num)** permet aussi de déplier le polyèdre. L'argument *tree* est l'arbre fourni par la fonction *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 :

```

1 \begin{luacode*}
2 nbimages = 70 -- must be global
3 -- images creation
4 local g = graph3d:new{ viewdir=perspective("central",30,60), bg="gray", size={10,10}, margin={0,0,0,0} }
5 -- declarations
6 require 'luadraw_polyhedrons'
7 require 'luadraw_cvx_polyhedra_nets'
8 local p = linspace(0,1,36)
9 local T = linspace(0,360,nbimages+1)
10 local P = dodecahedron(Origin, -2*vecI)
11 local net = unfold_polyhedron(P)
12 local tree = net.tree
13 -- create the image number k, this function must be global
14 function makeframe(k)
15     local r = k
16     if k > 36 then r = 72-k end

```

```

17   local P1 = rotate3d( unfold_tree(tree,p[r]), T[k], {Origin,vecK})
18   g:Dfacet(P1, {color="Crimson", edgecolor="Gold", edgewidth=8})
19   -- send image number k
20   g:Sendtotex() -- send the tikzpicture to TeX
21   g:Cleargraph()
22 end
23 \end{luacode*}

```

Le code  $\text{\TeX}$  (avec le paquet *animate*) :

```

\def\nb{\directlua{tex.print(nbimages)}}
\def\makeframe#1{\directlua{makeframe(#1)}}%

\begin{animateinline}[poster=first,controls,loop]{8}
\multiframe{\nb}{ik=1+1}{%
\makeframe{ik}%
}%
\end{animateinline}

```

Le résultat :

FIGURE 12 : Dépliage d'un dodécaèdre

## 6) Le module *luadraw\_fields*

Ce module contient les fonctions et 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 **field(f,x1,x2,y1,y2,grid,length)** qui renvoie le champ de vecteurs (liste de segments). L'argument  $f$  est une fonction  $f: (x,y) \mapsto f(x,y) \in \mathbf{R}^2$  ( $f(x,y)$  est une liste de deux réels). Le champ est calculé sur le pavé  $[x_1; x_2] \times [y_1; y_2]$ . L'argument *grid* 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 *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.
- La méthode **g:Dvectorfield(f,options)** fait le dessin du champ de vecteurs associé à la fonction  $f$ . L'argument *options* est une table dont les champs définissent les paramètres, qui sont (avec leur valeur par défaut) :
  - **view = fenêtre par défaut**, liste de la forme  $\{x1,x2,y1,y2\}$  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 =` , 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**  $f: (x,y) \mapsto f(x,y) \in \mathbf{R}$ . L'argument *options* est identique à celui de la méthode *Dvectorfield*.
- Un exemple de ces deux méthodes a déjà été donné page 67.

## 7) Le module *luadraw\_shadedforms*

Ce module permet de dessiner des lignes polygonales ou de remplir une forme en utilisant un gradient de couleurs. Ce module charge l'extension *luadraw\_palette*.

### Dshadedpolyline

La méthode **g:Dshadedpolyline(L, palette, options)** permet de dessiner une ligne polygonale 2d (*L*) avec un dégradé de couleurs en fonction de la méthode de calcul et de la *palette* choisies. *L* est une liste de nombres complexes ou une liste de listes de nombres complexes, *palette* 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 *options* est une table dont les champs définissent les paramètres, qui sont (avec leur valeur par défaut) :

- `values = "x"`, pour chaque point de *L* 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 *L* la valeur sera l'abscisse.
  - "y", dans ce cas pour chaque point de *L* 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 *L* la valeur sera donnée par  $f(x,y)$ .
- `width = current line width`, permet de définir l'épaisseur de ligne en dixième de point (épaisseur courante par défaut).
- `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 *L* en une succession de trapèzes qui sont ensuite remplis avec un dégradé de couleur.

```

1 \begin{luadraw}{name=shading_polyline}
2 local i = cpx.I
3 local g = graph:new{size={10,10},bg="lightgray", margin={0,0,0,0}}
4 require 'luadraw_shadedforms'
5 -- first example diff. equation y' = x^2+y^2-1 (=f(x,y))
6 local x1,x2,y1,y2 = -3,3,-3,3
7 local A = Z(0,1/2) -- initial condition
8 local f = function(x,y)
9     return x^2+y^2-1
10 end
11 local S = odesolve(f, A.re, A.im, x1, x2, 150) -- S is a matrix {X,Y}
12 local L = {} -- to convert {X,Y} into the complex numbers list L
13 for k = 1, #S[1] do table.insert(L, Z(S[1][k],S[2][k])) end
14 L = clippolyline(L,-2.5,2.4,y1,y2)[1] -- L is the solution curve
15 g:Dshadedpolyline(L, palRainbow, {values=f, width=12}) -- solution drawn with rainbow color map using function f
16 -- second example
17 L = polar(function(t) return 2*math.cos(3*t) end, -math.pi, math.pi)
18 local f = function(x,y) return cpx.abs(Z(x,y)) end -- here the value will be the modulus
19 g:Shift(2-2.5*i)
20 g:Dshadedpolyline(L, palAutumn, {values=f, width=12})
21 -- third example
22 g:Shift(-4.5+5*i)
23 g:Dshadedpolyline( polyreg(0,2,8), palGasFlame, {values="y", width=24, close=true})
24 g:Show()
25 \end{luadraw}

```

FIGURE 13 : 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, avec 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 paramètres, 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 :

$$A, A+\text{length}*\text{dir}, A+\text{length}*\text{dir}+\text{width}*cpx.I*\text{dir}, A+\text{width}*cpx.I*\text{dir}$$

- **values** = 0, cette option permet de définir soit le nombre de valeurs numériques affichées équiréparties dans l'intervalle *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*).

```

1 \begin{luadraw}{name=Dcolorbar}
2 local g = graph:new{size={10,10}, bbox=false}
3 g:Labelsize("small")
4 require 'luadraw_shadedforms'
5 g:Dcolorbar(Z(-4,-4), palRainbow)
6 g:Dcolorbar(Z(-2,-4), palGasFlame, {minmax={0,7},values=8})
7 g:Dcolorbar(Z(-1,1), palAutumn, {dir=1,length=6,width=1,addvalues={0.25,0.5,0.75},labelpos="S"})
8 g:Dcolorbar(Z(-1,-1), palViridis, {dir=1,length=6,width=-1,addvalues={0.25,0.5,0.75},labelpos="N"})
9 g:Dlabel("No value", Z(-4.25,-4),{pos="S"},
10 "\parbox{1.95cm}{minmax={0,7}\\values=8}", Z(-2.25,-4), {},

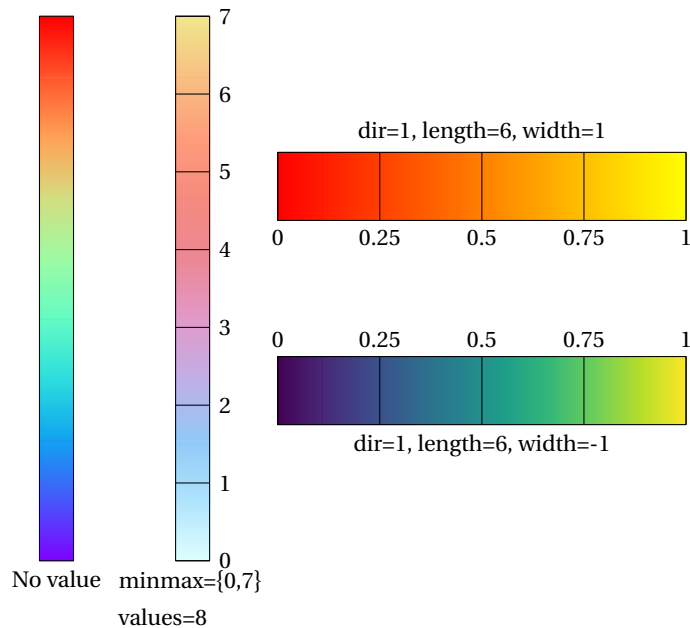
```

```

11 "dir=1, length=6, width=1", Z(2,2), {pos="N"},
12 "dir=1, length=6, width=-1", Z(2,-2), {pos="S"})
13 g:Show()
14 \end{luadraw}

```

FIGURE 14 : Color bars



### Dshadedrectangle

**NB :** l'utilisation de cette méthode nécessite la librairie *shadings*.

La méthode **g:Dshadedrectangle(x1,x2,y1,y2,pal,options)** permet de remplir le rectangle  $[x1;x2] \times [y1;y2]$  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 paramètres, qui sont (avec leur valeur par défaut) :

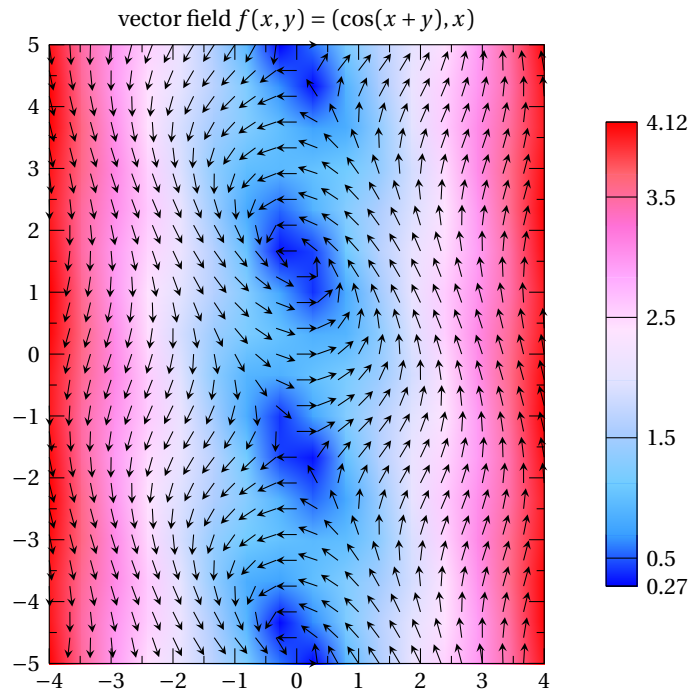
- **values** = `function(x,y) return cpx.abs(Z(x,y)) end`, cette option définit la fonction  $f$  utilisée pour calculer la couleur de chaque point. La valeur maximale de  $f$  sur le rectangle correspondra à la dernière couleur de la palette, et la valeur minimale de  $f$  correspondra à la première couleur de la palette.
- **grid** = `{15,15}`, cette option définit le nombre de subdivisions pour l'intervalle  $[x1;x2]$  et pour l'intervalle  $[y1;y2]$ . Plus la subdivision est fine plus l'affichage sera long...
- **bar** = `"none"`, permet d'ajouter ou non une légende en faisant appel à la méthode *Dcolorbar*, cette option peut valoir : `"none"`, ou `"right"`, ou `"bottom"`, ou `"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).

```

1 \begin{luadraw}{name=Dshadedrectangle}
2 local g = graph:new{window={-4,6,-5.5,5.5}, size={10,10}, bbox=false}
3 require 'luadraw_shadedforms'
4 require 'luadraw_fields'
5 g:Labelsize("small")
6 local f = function(x,y) return {math.cos(x+y), x} end -- vector field
7 local Nf = function(x,y) local A = f(x,y); return cpx.abs(Z(A[1],A[2])) end -- modulus of f(x,y)
8 g:Dshadedrectangle(-4,4,-5,5, palPicnic, {values=Nf, bar="right", baroptions={addvalues={0.5,1.5,2.5,3.5}}})
9 g:Dvectorfield(f,{view={-4,4,-5,5},draw_options="-stealth"})
10 g:Dgradbox({Z(-4,-5),Z(4,5),1,1}, {title="vector field $f(x,y)=(\\cos(x+y),x)$"})
11 g:Show()
12 \end{luadraw}

```

FIGURE 15 : Shaded rectangle



### Dshadedregion

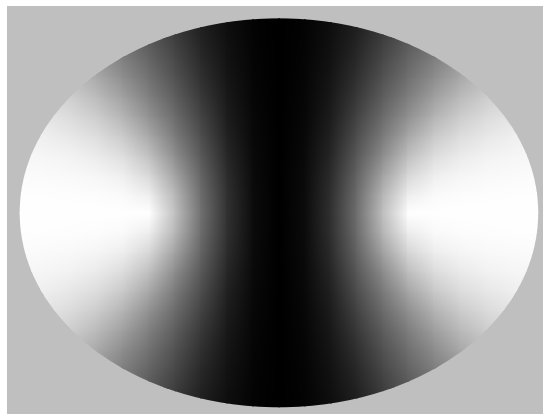
La méthode **g:Dshadedregion(*apath*,*pal*,*options*)** permet de remplir la région définie par le chemin *apath* avec un gradient de couleurs extraites de la palette *pal*. Cette méthode utilise la précédente (*Dshadedrectangle*) sur le rectangle définie par la boîte englobante du chemin. Chaque point  $(x,y)$  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*.

```

1 \begin{luadraw}{name=Dshadedregion}
2 local g = graph:new{window={-4.5,6,-4.5,5}, size={10,10},bg="lightgray", bbox=false}
3 g:Labelsize("small")
4 require 'luadraw_shadedforms'
5 local L1, L2 = -2, 2
6 local f = function(x,y)
7     local z = Z(x,y)
8     return (cpx.abs(z-L1)-cpx.abs(z-L2))^2
9 end
10 g:Dshadedregion({4,0,4,3,"e"}, getPal("Grays",{reverse=true}), {values=f, grid={20,20}})
11 g:Show()
12 \end{luadraw}

```

FIGURE 16 : Shaded region



## 8) Le module *luadraw\_povray*

Ce module permet de créer des fichiers sources (relativement basiques) pour le logiciel *Pov-Ray* et de les compiler à la volée si ce logiciel est bien installé sur votre système et que son chemin d'accès est connu. 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.

### Avant la création des objets

Il faut **obligatoirement initialiser** le dessin *Pov-Ray* avec la méthode :

**g:Pov\_new(options).**

L'argument *options* est une table dont les champs 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 au 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).
- **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 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*".
- **win\_param\_ext** = "/RENDER /EXIT", paramètres supplémentaires qui ne concernent que windows.

On peut ensuite demander à *Pov-Ray* d'inclure certains fichiers *\*.inc* avec la méthode :

**g:Pov\_include("file1.inc","file2.inc",...),**

par exemple : **g:Pov\_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.

### 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 paramètres de l'objet, voici les **paramètres communs 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** = **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*).
  - **opacity** = 1, nombre entre 0 et 1 définissant l'opacité de l'objet.
  - **ambient** = 0.35, **diffuse** = 0.8 et **phong** = 0.5 : 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.
- `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 boîte, 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 *povfunction* 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 =`, 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 :

```
1 local f = function(x,y,z) return x^2+y^2+z^2 - 1 end
2 local r = 1.1
3 g:Pov_implicit("x*x+y*y+z*z-1", f, {color=SteelBlue, containedby={M(-r,-r,-r), M(r,r,r)}})
```

- **Surfaces paramétrées** par  $(u, v) \mapsto (x(u, v), y(u, v), z(u, v))$ . C'est la méthode :

**g:Pov\_surface( xfunc, yfunc, zfunc, u1, u2, v1, v2, options)**

Les arguments *xfunc*, *yfunc* et *zfunc* sont trois chaînes contenant respectivement les expressions  $x(u, v)$ , de  $y(u, v)$  et de  $z(u, v)$ . Les arguments *u1* et *u2*, respectivement *v1* et *v2*, 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 =`, 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 :

```
1 local pi = math.pi
2 local r = 1.1
3 g:Pov_surface("cos(u)*sin(v)", "sin(u)*sin(v)", "cos(v)", -pi, pi, 0, pi, {color=SteelBlue,
  containedby={M(-r,-r,-r), M(r,r,r)}})
```

Remarque : on peut mettre "*pi*" à la place de *pi* car cette constante est définie dans Pov-Ray. D'autre part, sur cet exemple, on aurait pu ajouter l'option `grad_max=1` compte tenu du paramétrage.

- **Polyèdre ou liste de facettes.** C'est la méthode :

#### **g:Pov\_facet(F,options)**

L'argument  $F$  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 = current line style`, style des arêtes.
- `edgecolor = Black`, couleur des arêtes.
- `edgewidth = current line width`, épaisseur des arêtes.
- `hidden = false`, booléen indiquant si les arêtes cachées doivent être dessinées.
- `hiddenstyle = Hiddenlinestyle`, style des arêtes cachées.

Exemple :

```
1 local T1 = tetra(M(-1,-1,-1), 3*vecI, 3*vecJ, M(1,1,3))
2 g:Pov_facet(T1, {color=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  $L$  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 = current line style`, style de la ligne.
- `width = current line width`, épaisseur de la ligne.
- `close = false`, booléen indiquant si la ligne doit être refermée.
- `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`, permet de jouer sur la taille des flèches.
- `hidden = false`, booléen indiquant si les parties cachées doivent être dessinées.
- `hiddenstyle = Hiddenlinestyle`, style des parties cachées.

Exemple : dessin des axes

```
1 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).
- `dotsscale = 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 *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  $O$  comme point d'intersection. Les options sont les mêmes que pour *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* (point3d) et de rayon *radius*. Ce sont les options communes.

- **g:Pov\_torus(center, R, r, N, options)**, qui dessine le tore de centre *center* (point3d) 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 **deux** 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.
- **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. 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

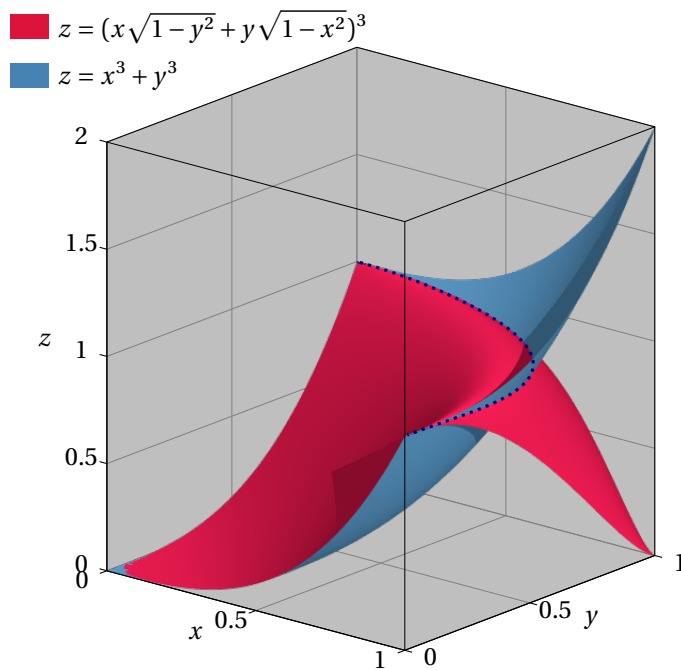
#### Intersection de deux surfaces implicites

```

1 \begin{luadraw}{name=intersection_surf}
2 local g = 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}}
3 local sqrt = math.sqrt
4 require "luadraw_povray"
5 local f = function(x,y,z) return x^3+y^3-z end
6 local h = function(x,y,z) return (x*sqrt(1-y^2)+y*sqrt(1-x^2))^3-z end
7 local L = implicit(function(x,y) return f(x,y,0)-h(x,y,0) end,0.01,1,0.01,1,{25,25})
8 L = map(function(z) return M(z.re,z.im,z.re^3+z.im^3) end, L[1]) -- intersection curve
9 -- using povray
10 g:Pov_new()
11 g:Pov_implicit("pow(x,3)+pow(y,3)-z", f, {color=SteelBlue})
12 g:Pov_implicit("z-pow(x*sqrt(1-y*y)+y*sqrt(1-x*x),3)", h, {color=Crimson, containedby={M(0,0,0),M(0.999,0.999,2)}})
13 g:Pov_exec()
14 -- drawing
15 g:Dboxaxes3d({grid=true, gridcolor="gray",fillcolor="lightgray",xyzstep=0.5, drawbox=true})
16 g:Pov_show() --Pov-Ray image
17 g:Dpolyline3d(L, "line width=1.2pt,dotted,Navy") --we draw the curve over the image
18 local d = 0.1
19 g:Dsquare(Z(-0.25,2.25), Z(-0.25,2.25-d),1,"draw=none,fill=Crimson")
20 g:Dsquare(Z(-0.25,2.25-2*d), Z(-0.25,2.25-3*d),1,"draw=none,fill=SteelBlue")
21 g:Dlabel("$z=(x\\sqrt{1-y^2}+y\\sqrt{1-x^2})^3$", Z(-0.25+d,2.25-d/2) , {pos="E"},
22         "$z=x^3+y^3$", Z(-0.25+d,2.25-5*d/2),{f})
23 g:Show()
24 \end{luadraw}

```

FIGURE 17 : Intersection de deux surfaces implicites



### Cercles de Villarceau

```

1 \begin{luadraw}{name=Villarceau_circles, exec=true}
2 local g = graph3d:new{window={-6.5,6.5,-5,5},margin={0,0,0,0}, size={10,10}, viewdir={20,65}}
3 require "luadraw_povray"
4 local R, r = 3, 1
5 local N = rotate3d(vecK, math.asin(r/R)*rad, {Origin, vecJ})
6 local P = {Origin, -N}
7 -- using povray
8 g:Pov_new({bg=LightGray})
9 g:Pov_torus(Origin, R, r, vecK, {color=SteelBlue, clipplane=P})
10 g:Pov_plane(P,{color=SeaGreen, opacity=0.4, edge=true, scale=0.9})

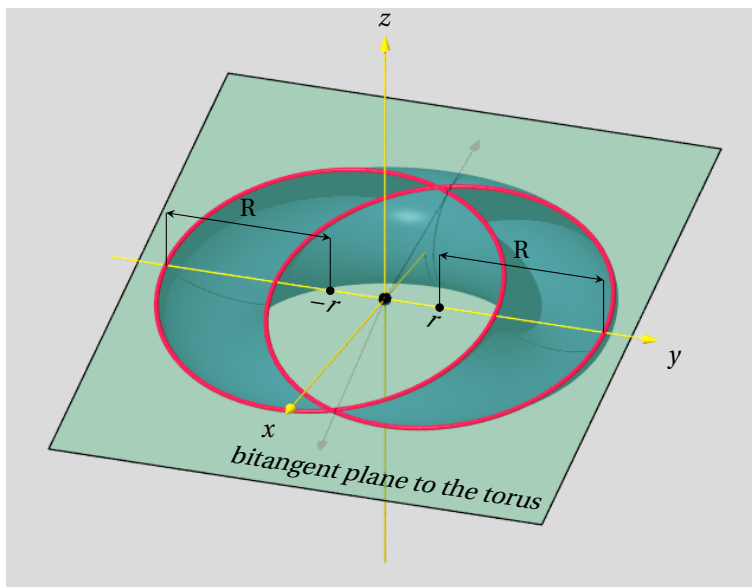
```

```

11 g:Pov_axes(Origin,{color=Gold,arrows=1})
12 g:Pov_dots(Origin, {dotscale=1.5})
13 g:Pov_circle( M(0,r,0),R,N,{color=Crimson, width=12})
14 g:Pov_circle( M(0,-r,0),R,N,{color=Crimson, width=12})
15 g:Pov_exec()
16 -- drawing
17 g:Pov_show()
18 local F = g:Plane2facet(P,0.9)
19 g:Dpolyline3d({{-r*vecJ,-r*vecJ+vecK},{-(R+r)*vecJ,-(R+r)*vecJ+vecK}})
20 g:Dpolyline3d({{r*vecJ,r*vecJ+vecK},{(R+r)*vecJ,(R+r)*vecJ+vecK}})
21 g:Dpolyline3d({{-r*vecJ+vecK,-(R+r)*vecJ+vecK},{r*vecJ+vecK,(R+r)*vecJ+vecK}}, "stealth-stealth")
22 g:Ddots3d({-r*vecJ, r*vecJ})
23 g:Dlabel3d("$x$",5*vecI,{pos="SW"},"$y$", 5*vecJ,{pos="SE"}, "$z$",5*vecK,{pos="N"},
24 "$R$", -(R/2+r)*vecJ+vecK, {}, "$R$", (R/2+r)*vecJ+vecK, {},
25 "bitangent plane to the torus", F[1], {pos="NW", dir={vecJ,pt3d.prod(N,vecJ)}},
26 "$-r$",-r*vecJ,{pos="S"}, "$r$", r*vecJ,{})
27 g:Show()
28 \end{luadraw}

```

FIGURE 18 : Cercles de Villarceau



### Trous dans une demi-sphère

```

1 \begin{luadraw}{name=holes_in_hemisphere, exec=true}
2 local g = graph3d:new{window={-5,5,-4,5}, size={10,10}, bg="lightgray"}
3 require "luadraw_povray"
4 local O, R = Origin, 4
5 local A, r = 2*R/3*vecJ, R/3
6 local P = {Origin, vecK}
7 local base = circle3d(A,r,vecK)[1] --circular base of the cylinder (list of 3d points)
8 local Cylborder = {}
9 for _, C in ipairs(base) do -- we project each point C of the base onto the half-sphere
10     table.insert(Cylborder,C + math.sqrt(R^2-pt3d.abs2(C))*vecK)
11 end -- Cylborder is now the intersection between the half-sphere and the cylinder
12 -- using povray
13 g:Pov_new()
14 g:Pov_sphere(Origin, R, {name="sph", clipplane=P, render=false}) -- only declaration
15 g:Pov_cylinder(A-vecK, r, A+R*vecK, {name="cyl1", render=false})
16 g:Pov_union({"cyl1", "cyl1 rotate 90*z"}, {name="cylext", render=false})
17 g:Pov_cylinder(A-vecK, r-0.001, A+R*vecK, {name="cyl2", render=false})
18 -- renderings
19 g:Pov_difference({"sph","cylext"}, {color=Blue, opacity=0.7})
20 g:Pov_union( {"cyl2", "cyl2 rotate 90*z"}, {color=LightBlue,opacity=0.8,clipbox="sph"})
21 g:Pov_axes(Origin,{color=Gold,arrows=1})
22 g:Pov_exec()
23 -- drawing

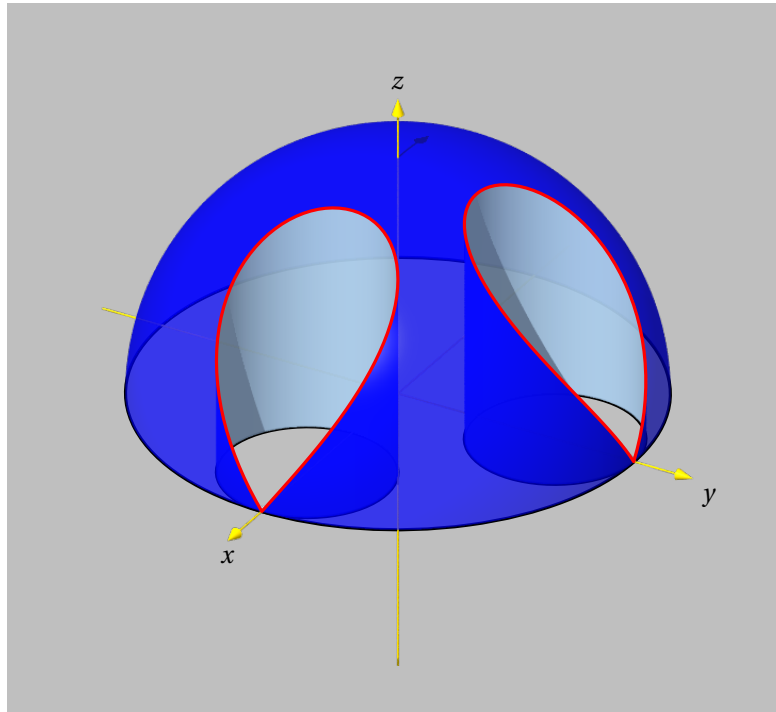
```

```

24 g:Dcircle3d(0,R,vecK,"line width=1.2pt"); g:Dcircle3d(A,r,vecK,"line width=1.2pt")
25 g:Dcircle3d(rotate3d(A,-90,{Origin,vecK}), r,vecK,"line width=1.2pt")
26 g:Pov_show()
27 g:Dpolyline3d( {Cylborder, rotate3d(Cylborder,-90,{Origin,vecK})}, "red, line width=1.2pt")
28 g:Dlabel3d("$x$",5*vecI,{pos="S"}, "$y$", 5*vecJ, {pos="SE"}, "$z$", 5*vecK, {pos="N"})
29 g:Show()
30 \end{luadraw}

```

FIGURE 19 : Trous dans une demi-sphère



## II Historique

### 1) 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...

### 2) Version 2.5

Liste non exhaustive :

- Ajout de la fonction *read\_csv\_file()*<sup>3</sup> qui permet de lire un fichier *csv* avec différentes options.

3. Sur une idée de Christophe BAL.

- L'extension *luadraw\_palettes* est passée à la version 1.3.0 du projet @prism 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...

### 3) 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...

### 4) 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 *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...

### 5) Version 2.2

Liste non exhaustive :

- Ajout de l'option *clip* pour les méthodes : *Dfacet()*, *Dmixfacet()*, *addFacet()*, *addPoly()* et *addPolyline()*, ainsi que pour les méthodes de dessin de nuages de points, et les méthodes de dessin "au trait" comme *Dpolyline3d()*, *Dparametric3d()*, *Dpath3d()*, etc.
- Ajout de l'option *xyzstep* pour la méthode *Dboxaxes3d()*, cette option définit un pas commun aux trois axes (1 par défaut).
- Ajout des méthodes *DSdots()*, *DSstars()*, *DSinvstereo\_curve()* et *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 *Dpoly*, *Dfacet*, *Dmixfacet*, *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 *Dscene3d()*.
- Ajout de la fonction *evalf(f,...)* qui permet une évaluation protégée de *f(...)*, 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...

## 6) 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 **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'argument *mode* de la méthode **g:Dplane**.
- Ajout de l'option *close* pour la méthode **g:addPolyline**.
- Correction de bug...

## 7) 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.

## 8) Version 1.0

Première version.