

# The *luadraw* package (3.1)

Patrick Fradin

May 29, 2026

## 2D and 3D drawing with Lua and TikZ.

<https://github.com/pfradin/luadraw>

### Abstract

The *luadraw* package defines the environment of the same name, which lets you create mathematical graphs using the Lua language. These graphs are ultimately drawn by TikZ (and automatically saved), so why make them in Lua? Because Lua brings all the power of a simple, efficient programming language with good computational capabilities.

# Contents

<b>I</b>	<b>2D Drawing</b>	<b>9</b>		
I	Introduction	9		
1)	What's New Since Version 3.0	9		
2)	Prerequisites	10		
3)	<i>luadraw</i> Environment Options	11		
4)	The <i>cpx</i> (complex numbers) class	12		
5)	Displaying a Variable in the Terminal	13		
6)	Creating a Graph	13		
7)	Can we use TikZ directly in the <i>luadraw</i> environment?	14		
II	Graphics Methods	14		
1)	Polygonal Lines	15		
	<i>Dpolyline</i>	15		
	Options for drawing a polygonal line	15		
2)	Segments and Lines	17		
	<i>Dangle</i>	17		
	<i>Dbissec</i>	18		
	<i>Dhline</i>	18		
	<i>Dline</i>	18		
	<i>DlineEq</i>	18		
	<i>Dmarkarc</i>	18		
	<i>Dmarkseg</i>	18		
	<i>Dmed</i>	19		
	<i>Dparallel</i>	19		
	<i>Dperp</i>	19		
	<i>Dseg</i>	19		
	<i>Dtangent</i>	19		
	<i>DtangentC</i>	20		
	<i>DtangentI</i>	20		
	<i>Dtangent_from</i>	20		
	<i>Dnormal</i>	21		
	<i>DnormalC</i>	21		
	<i>DnormalI</i>	21		
3)	Geometric Figures	22		
	<i>Darc</i>	22		
	<i>Dcircle</i>	23		
	<i>Dellipse</i>	23		
	<i>Dellipticarc</i>	23		
	<i>Dparallelogram</i>	23		
	<i>Dpolyreg</i>	23		
	<i>Drectangle</i>	24		
	<i>Dsequence</i>	24		
	<i>Dsquare</i>	25		
	<i>Dwedge</i>	25		
4)	Curves	25		
	Parametric: <i>Dparametric</i>	25		
	Polars: <i>Dpolar</i>	26		
	Cartesian: <i>Dcartesian</i>	26		
	Periodic Functions: <i>Dperiodic</i>	27		
	Step Functions: <i>Dstepfunction</i>	27		
	Piecewise Affine Functions:			
	<i>Daffinebypiece</i>	28		
	Differential Equations: <i>Dodesolve</i>	28		
	Implicit Curves: <i>Dimplicit</i>	30		
	Contour Lines: <i>Dcontour</i>	30		
	Parameterization of a Polygonal			
	Line: <i>curvilinear_param</i>	31		
5)	Domains related to Cartesian curves	32		
	<i>Ddomain1</i>	32		
	<i>Ddomain2</i>	32		
	<i>Ddomain3</i>	33		
	<i>Dinequalities</i>	34		
	<i>Dimplicit_inequalities</i>	34		
6)	Points ( <i>Ddots</i> ) and Labels ( <i>Dlabel</i> )	36		
7)	Paths: <i>Dpath</i> , <i>Dspline</i> , and <i>Dtcurve</i>	37		
	What is a path	37		
	Draw a Path	39		
8)	Paths and Clipping: <i>Beginclip()</i> and <i>Endclip()</i>	40		
9)	Axes and Grids	41		
	<i>Daxes</i>	42		
	<i>DaxeX</i> and <i>DaxeY</i>	44		
	<i>Dgradline</i>	46		
	<i>Dgrid</i>	48		
	<i>Dgradbox</i>	49		
10)	Set Drawings (Venn Diagrams)	50		
	Drawing a Set	50		
	Operations on Sets	51		
11)	Importing an Image	52		
	Placing an Image in the Graphic	52		
	Mapping an Image onto a			
	Parallelogram	53		
12)	Colors	54		
	Color Calculations	54		
III	Geometric Constructions	56		
1)	<i>circumcircle()</i> , <i>incircle()</i>	56		
2)	<i>cvx_hull2d()</i>	56		
3)	<i>delaunay()</i>	56		
4)	<i>voronoi()</i>	57		
5)	<i>line2strip()</i>	58		

6)	<code>parallel_polyline()</code>	59		<code>mtransform</code> and <code>mLtransform</code>	71
7)	<code>sss_triangle()</code>	59	2)	Matrix associated with the graph	71
8)	<code>sas_triangle()</code>	59		<code>g:Composematrix()</code>	71
9)	<code>asa_triangle()</code>	59		<code>g:Det2d()</code>	71
IV	Computations on Lists	60		<code>g:IDmatrix()</code>	71
1)	<code>concat</code>	60		<code>g:Mtransform()</code>	71
2)	<code>cut</code>	60		<code>g:MLtransform()</code>	71
3)	<code>cutpolyline</code>	60		<code>g:Rotate()</code>	72
4)	<code>cutpolyline2</code>	61		<code>g:Scale()</code>	72
5)	<code>getbounds</code>	62		<code>g:Savematrix()</code> and <code>g:Restorematrix()</code>	72
6)	<code>getdot</code>	62		<code>g:Setmatrix()</code>	72
7)	<code>insert</code>	62		<code>g:Shift()</code>	72
8)	<code>interCC</code>	63	3)	View change. Change of coordinate system	73
9)	<code>interD</code>	64	VII	Adding Your Own Methods	74
10)	<code>interDC</code>	64	1)	An Example	75
11)	<code>interDL</code>	64	2)	How to import the file	76
12)	<code>interL</code>	64	3)	Modifying an existing method	77
13)	<code>interP</code>	64			
14)	<code>linspace</code>	64	2	3D Drawing	79
15)	<code>map</code>	64	I	Introduction	79
16)	<code>merge</code>	64	1)	Prerequisites	79
17)	<code>polyline2path</code>	64	2)	Some reminders	80
18)	<code>range</code>	64	3)	Creating a 3D Graph	80
19)	<code>read_csv_file</code>	65	4)	Affine Projection Modes	81
20)	Clipping Functions	65	5)	Central Projection	82
21)	Adding Mathematical Functions	65	II	The <code>pt3d</code> Class	83
	Protected Evaluation: <code>evalf</code>	65	1)	Representation of Points and Vectors	83
	<code>int</code>	66	2)	Operations on 3D Points	83
	<code>gcd</code>	66	3)	Methods of the class <i>pt3d</i>	84
	<code>lcm</code>	66	4)	Displaying a Variable in the Terminal	84
	<code>nth_root</code>	66	III	Graphics Methods	84
	<code>solve</code>	66	1)	Line Drawing	85
V	Transformations	68		Polygonal Line: <code>Dpolyline3d</code>	85
1)	<code>affin</code>	68		Right Angle: <code>Dangle3d</code>	85
2)	<code>ftransform</code>	68		Segment: <code>Dseg3d</code>	85
3)	<code>hom</code>	68		Line: <code>Dline3d</code>	85
4)	<code>inv</code>	68		Circular arc: <code>Darc3d</code>	85
5)	<code>proj</code>	68		Circle: <code>Dcircle3d</code>	86
6)	<code>projO</code>	69		3D Path: <code>Dpath3d</code>	86
7)	<code>rotate</code>	69		Plane: <code>Dplane</code>	87
8)	<code>shift</code>	69		Parametric curve: <code>Dparametric3d</code>	88
9)	<code>simil</code>	69		Parameterization of a Polygonal Line: <i>curvilinear_param3d</i>	89
10)	<code>sym</code>	69		The reference: <code>Dboxaxes3d</code>	89
11)	<code>symG</code>	69	2)	Points and Labels	90
12)	<code>symO</code>	69		3D Points: <code>Ddots3d</code> , <code>Dballdots3d</code> , <code>Dcrossdots3d</code>	90
VI	Matrix Calculus	70		3D Labels: <code>Dlabel3d</code>	91
1)	Matrix Calculations	70	3)	Basic Solids (Without Facets)	91
	<code>applymatrix</code> and <code>applyLmatrix</code>	70		Cylinder: <code>Dcylinder</code>	91
	<code>composematrix</code>	70		Cone: <code>Dcone</code>	92
	<code>invmatrix</code>	70			
	<code>matrixof</code>	70			

	Frustum: Dfrustum . . . . .	93		3) Planes: plane(), planeEq(), orthoframe(), plane2ABC() . . . . .	127
	Sphere: Dsphere . . . . .	94		4) Circumscribed Sphere, Inscribed Sphere: circumsphere(), insphere() . . . . .	128
IV	Faceted Solids . . . . .	95		5) Fixed-length tetrahedron: tetra_len() . . . . .	129
1)	Definition of a Solid . . . . .	95		6) Triangles: sss_triangle3d(), sas_triangle3d(), asa_triangle3d() . . . . .	129
2)	Drawing a Polyhedron: Dpoly . . . . .	96			
3)	Displaying the Face and/or Vertex Numbers of a Polyhedron . . . . .	97	VII	Matrix Calculus Transformations and Some Mathematical Functions . . . . .	130
4)	Polyhedron Construction Functions . . . . .	98	1)	3D Transformations . . . . .	130
5)	Reading from an obj file . . . . .	102		Apply a transformation function: ftransform3d . . . . .	130
6)	Surface in obj format . . . . .	103		Projections: proj3d, proj3dO, dproj3d . . . . .	130
7)	Drawing a List of Facets: Dfacet and Dmixfacet . . . . .	103		Projections onto axes or planes related to axes . . . . .	130
8)	Functions for Constructing Facet Lists surface() . . . . .	105		Symmetries: sym3d, sym3dO, dsym3d, psym3d . . . . .	131
	cartesian3d() . . . . .	105		Rotation: rotate3d, rotateaxe3d . . . . .	131
	cylindrical_surface() . . . . .	106		Scaling: scale3d . . . . .	131
	curve2cone() . . . . .	107		Inversion: inv3d . . . . .	131
	curve2cylinder() . . . . .	108		Stereography: projstereo and inv_projstereo . . . . .	131
	line2tube(); section2tube() . . . . .	109		Translation: shift3d . . . . .	131
	rotcurve() . . . . .	110	2)	Matrix Calculus . . . . .	131
	rotline() . . . . .	111		applymatrix3d and applyLmatrix3d . . . . .	132
9)	Edges of a solid . . . . .	112		composematrix3d . . . . .	132
10)	Methods and functions applying to facets or polyhedra . . . . .	113		invmatrix3d . . . . .	132
11)	Cutting a solid: cutpoly and cutfacet . . . . .	114		matrix3dof . . . . .	132
12)	Clipping Facets with a Convex Polyhedron: clip3d . . . . .	115		mtransform3d and mLtransform3d . . . . .	132
13)	Clip a plane with a convex polyhedron: clipplane . . . . .	116	3)	Matrix associated with the 3D graph g:Composematrix3d() . . . . .	132
V	The Dscene3d Method . . . . .	116		g:Det3d() . . . . .	132
1)	The Principle, the Limitations . . . . .	116		g:IDmatrix3d() . . . . .	132
2)	Construction of a 3D scene . . . . .	117		g:Mtransform3d() . . . . .	132
3)	Methods for adding an object to the 3D scene . . . . .	118		g:MLtransform3d() . . . . .	133
	Adding facets: g:addFacet and g:addPoly . . . . .	118		g:Rotate3d() . . . . .	133
	Adding a plane: g:addPlane and g:addPlaneEq . . . . .	119		g:Scale3d() . . . . .	133
	Add a polygonal line: g:addPolyline . . . . .	119		g:Setmatrix3d() . . . . .	133
	Add Axes: g:addAxes . . . . .	120	4)	Additional Mathematical Functions . . . . .	133
	Add a line: g:addLine . . . . .	120		clippolyline3d() . . . . .	133
	Adding a "right" angle: g:addAngle . . . . .	120		clipline3d() . . . . .	133
	Add a circular arc: g:addArc . . . . .	120		cutpolyline3d() . . . . .	133
	Add a circle: g:addCircle . . . . .	120		getbounds3d() . . . . .	134
	Adding points: g:addDots . . . . .	122		interDP() . . . . .	134
	Adding Labels: g:addLabels . . . . .	122		interPP() . . . . .	134
	Adding dividing walls: g:addWall . . . . .	123		interDD() . . . . .	134
VI	Geometric Constructions . . . . .	126		interCS() . . . . .	134
1)	Circumscribed circle, incircle: circumcircle3d(), incircle3d() . . . . .	126		interDS() . . . . .	134
2)	Convex Hull: cvx_hull3d() . . . . .	127		interPS() . . . . .	134
				interSS() . . . . .	134

	interSSS()	134		6)	The <i>luadraw_fields</i> module	166
	merge3d()	135		7)	The <i>luadraw_shadedforms</i> module	166
	split_points_by_visibility()	135			Dshadedpolyline	167
VIII	More Advanced Examples	136			Dcolorbar	168
1)	The Box of Sugars	136			Dshadedrectangle	169
2)	Stack of Cubes	138			Dshadedregion	170
3)	Illustration of Dandelin's Theorem	139	8)	The <i>luadraw_povray</i> Module	171	
4)	Volume defined by a double integral	141		Prerequisites and introduction	171	
5)	Volume defined on something other than a block	142		Default Settings	171	
				Before Creating Objects	172	
				Creation of Objects	172	
				List of Predefined Objects	173	
				Save, execution, inclusion	176	
				Examples	176	
3	Appendices	144	9)	The <i>luadraw_log_axes</i> module	179	
I	Extensions	144		Initialization: g:Beginlogview()	179	
1)	The <i>luadraw_polyhedrons</i> module	144		Drawing and Conversion Methods	181	
2)	The <i>luadraw_spherical</i> Module	147		End of grid usage: g:Endlogview()	182	
	Global Module Functions	147		Examples	182	
	Sphere Definition	147	10)	The <i>luadraw_decorations</i> module	183	
	Add a circle: g:DScircle	148		2D Polygonal Lines: g:Dpolyline()	184	
	Add a great circle: g:DSbigcircle	148		2D Paths: g:Dpath()	184	
	Add a great circle arc: g:DSarc	148		2D Lines: g:Dline()	184	
	Add an angle: g:DSangle	149		2D Segments: g:Dseg()	185	
	Add a spherical facet: g:DSfacet	149		2D Arc: g:Darc()	185	
	Add a spherical curve: g:DScurve	149		3D Arc: g:Darc3d()	186	
	Add a segment: g:DSeg	150	11)	The <i>luadraw_coils_chains</i> module	190	
	Add a line: g:DSline	150		Springs	190	
	Add a polygonal line: g:DSpolyline	150		Chains	193	
	Add a plane: g:DSplane	151	II	History	195	
	Add a label: g:DLabel	151	1)	Version 3.1	195	
	Add points: g:DSdots and g:DStars	151	2)	Version 3.0	196	
	Inverse Stereography: g:DSinvstereo_curve and g:DSinvstereo_polyline	152	3)	Version 2.8	197	
	Examples	152	4)	Version 2.7	197	
3)	The <i>luadraw_palettes</i> Module	157	5)	Version 2.6	198	
4)	The <i>luadraw_compile_tex</i> module	157	6)	Version 2.5	198	
	Part One: Compilation and Reading	158	7)	Version 2.4	199	
	Part Two: Using the Result	158	8)	Version 2.3	199	
5)	The <i>luadraw_cvx_polyhedra_nets</i> module	161	9)	Version 2.2	200	
	Basic Function	161	10)	Version 2.1	200	
	The Drawing Method	162	11)	Version 2.0	201	
	Examples	163	12)	Version 1.0	201	
	The <i>unfold_tree()</i> function	164				

# List of Figures

1	A first example: three sub-figures in the same graph	9
2	Vector field, integral curve of $y' = 1 - xy^2$	17
3	Tangents from a point	21
4	Symmetric of the orthocenter	22
5	Sequence $u_{n+1} = \cos(u_n)$	25
6	A Lokta-Volterra differential system	29
7	Example with Dcontour	31
8	Points distributed on a polygonal line	32
9	Floor function, Ddomain1 and Ddomain3 functions	33
10	Dinequalities	34
11	The <b>g:Dimplicit_inequalities</b> method	35
12	Path example	38
13	Path and Spline	39
14	Interpolation curve with imposed tangent vectors	40
15	Clipping Example	41
16	Example with axes with grid	44
17	Examples of numbered lines	48
18	Example of a non-orthogonal coordinate system	49
19	Using Dgradbox	50
20	Drawing a Set	51
21	Set Operations	52
22	Importing an Image	53
23	Mapping an Image	54
24	The five default palettes	56
25	Delaunay Triangulation	57
26	Voronoi diagram	58
27	Example with <i>line2strip</i>	59
28	sss_triangle, sas_triangle and asa_triangle	60
29	Illustrate a linear programming exercise	61
30	cutpolyline2	62
31	Tangents to a circle O,2 and to an ellipse O,3,2 from a point	63
32	Function $f$ defined by $\int_x^{f(x)} \exp(t^2)dt = 1$ .	68
33	Using Transformations	70
34	Using the Graph Matrix	72
35	Using Shift, Rotate and Scale	73
36	Classification of the points of a parametric curve	74
37	Using the new methods	77
38	Modifying an existing method	78

1	Col point at $M(0, 0, 0)$ ( $z = x^2 - y^2$ )	79
2	Viewing Angles	81
3	Affine Projection Modes	82
4	Central projection	83
5	Dplane, example with mode = left+bottom	87
6	A curve and its projections onto three planes	89
7	A tetrahedron and the centers of gravity of each face	91
8	Cylinders, Cones, and Spheres	95
9	Section of a tetrahedron by a plane	97
10	Visualize the faces and vertices of a polyhedron	98
11	Truncated cone, truncated pyramid, oblique cylinder	100
12	Hyperbola: cone-plane intersection	100
13	Cone section with multiple views	102
14	Mask of Nefertiti	103
15	Example of contour lines on a surface	105
16	Surfaces using the <i>addwall</i> option	107
17	Elliptical Cone Example	108
18	Section of a non-circular cylinder	109
19	Example with <i>line2tube</i> and <i>section2tube</i>	110
20	Example with <i>rotcurve</i>	111
21	Example with <i>rotline</i>	112
22	Sphere inscribed in an octahedron with the center projected onto the faces	114
23	Cube cut by a plane ( <i>cutpoly</i> ), with <i>close=false</i> and with <i>close=true</i>	115
24	Example with <i>clip3d</i> : constructing a die from a cube and a sphere	116
25	First example with <i>Dscene3d</i> : intersection of two planes	118
26	Solid cylinder immersed in water	121
27	Construction of an icosahedron	123
28	Example with <i>addWall</i> (the two transparent pink facets are normally invisible)	124
29	Torus and lemniscate	125
30	Section of a sphere without <i>Dscene3d()</i>	126
31	Using <i>cvx_hull3d()</i>	127
32	Faces of a cube with holes in it and a regular hexagon	128
33	A tetrahedron with fixed edge lengths	129
34	A curve on a cylinder	136
35	Box of Sugar Cubes	137
36	Stack of Cubes	139
37	Illustration of Dandelin's Theorem	141
38	Volume correspondent à $\int_{x_1}^{x_2} \int_{y_1}^{y_2} f(x, y) dx dy$	142
39	Volume : $0 \leq x \leq 1$ ; $0 \leq y \leq x^2$ ; $0 \leq z \leq y^2$	143
1	Polyhedra from the <i>luadraw_polyhedrons</i> module	146
2	Cube in a Sphere	153
3	Viviani window	154
4	A spherical tiling	155
5	Tangents to the sphere from a point	156
6	<i>DSinvstereo_curve</i> and <i>DSinvstereo_polyline</i> methods	157
7	Example with <i>compile_tex</i> in 2D	159
8	Write on a cylinder	161
9	Net of a parallelepiped	163
10	Imposed pattern of a parallelepiped	163
11	Half unfolded truncated parallelepiped	164
12	Unfolding a dodecahedron	166

13	Shaded polyline . . . . .	168
14	Color bars . . . . .	169
15	Shaded rectangle . . . . .	170
16	Shaded region . . . . .	171
17	Intersection of two implicit surfaces . . . . .	177
18	Villarceau circles . . . . .	178
19	Holes in a hemisphere . . . . .	179
20	Logarithmic x-axis . . . . .	182
21	Logarithmic x-axis and logarithmic xy-axis . . . . .	183
22	<code>g:Darc()</code> Method . . . . .	186
23	Method <code>g:Ddecoratedarc3d()</code> . . . . .	188
24	2D Decorations . . . . .	189
25	3D Decorations . . . . .	190
26	The <code>g:Dcoil()</code> Method . . . . .	191
27	The <code>g:Dcoil2()</code> Method . . . . .	193
28	The <code>g:Dchain()</code> Method . . . . .	194
29	The <code>g:Dchain2()</code> Method . . . . .	195



## 2D Drawing

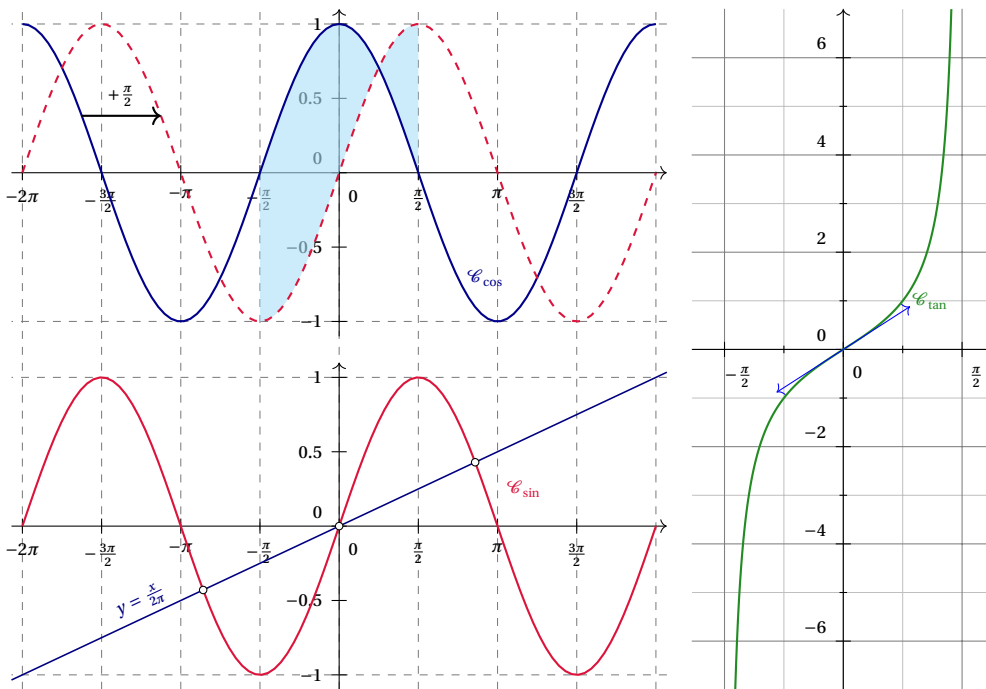


Figure 1: A first example: three sub-figures in the same graph

### I Introduction

#### 1) What's New Since Version 3.0

Since version 3.0, **all** data relating to the *luadraw* package is encapsulated in a single namespace, called **luadraw** (it's a table). This means that the data is only accessible using dot notation, specifically:

1. the class *graph* becomes *luadraw.graph*,
2. the class *cpx* (complex numbers) becomes *luadraw.cpx*,
3. the class *graph3d* becomes *luadraw.graph3d*,
4. the class *pt3d* (3D points) becomes *luadraw.pt3d*,
5. all global variables and all non-graphical functions are in the *luadraw* namespace, except for a few exceptions specific to complex numbers which are passed to the *cpx* class, and a few exceptions specific to 3D points which are passed to the *pt3d* class,

6. however, there is no change for the graphical methods since they were already encapsulated in the *graph* and *graph3d* classes.

The Lua language does not allow namespace factorization, so you really have to use dot notation, but fortunately you can create shortcuts (or aliases), for example:

```
local ld = luadraw -- alias on the namespace
local cpx, pt3d = ld.cpx, ld.pt3d -- shortcuts for the cpx and pt3d classes
local Z, M, i = cpx.Z, pt3d.M, cpx.I -- shortcuts to the Z and M functions and the complex number i
local Origin, vecI, vecJ, vecK = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK
local g = ld.graph3d:new{}
...
```

Due to this major change, scripts from previous versions **can no longer be compiled directly**; they must be adapted to this new version. However, with shortcuts, the changes are minimal.

Another new feature: the *luadraw* environment now uses a *luacode* environment (and no longer *luacode\**), which allows TeX macros to be inserted into Lua code. Therefore, the most frequently used shortcuts can be put into a macro and used in *luadraw* code, for example:

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

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

\begin{document}

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

The starred environment *luadraw\**, on the other hand, uses a *luacode\** environment; in this case, there is no interference from TeX in the Lua code.

**NB** : Throughout this documentation, we will use the following shortcuts:

```
local ld = luadraw -- alias on the namespace
local cpx = ld.cpx -- shortcut for the cpx class
local pt3d = ld.pt3d -- shortcut for the pt3d class
```

## 2) Prerequisites

- In the preamble, you must declare the *luadraw* package: `\usepackage[global options]{luadraw}`
- Compilation is done with LuaLatex **exclusively**.
- The colors in the *luadraw* environment are strings that must correspond to colors known to TikZ. It is strongly recommended to use the *xcolor* package with the *svgnames* option.

Regardless of the global options chosen, this package loads the *luadraw\_graph2d.lua* module, which defines the *ld.graph* class, and provides the *luadraw* environment for creating graphs in Lua. To create 3D graphics, you must also load the *ld.graph3d* class using the global option *3d*:

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

**Global package options** : `noexec`, `3d`, and `cachedir=`.

- `noexec`: When this global option is specified, the default value of the `exec` option for the `luadraw` environment will be false (and no longer true).
- `3d`: When this global option is specified, the `luadraw_graph3d.lua` module is also loaded. This module also defines the `ld.graph3d` class (which relies on the `ld.graph` class) for 3D drawings.
- `cachedir=<folder>`: By default, the created files are saved in the `_luadraw` folder, which is a subfolder of the current folder (containing the master document). This folder can be changed with the `cachedir` option, for example `cachedir={tikz}`.

**NB:** In this chapter, we will not discuss the `3d` option. This is the subject of the next chapter. We will therefore only discuss the 2D version.

When a graph is finished, it is exported in TikZ format, so this package also loads the TikZ package and the libraries:

- `patterns`
- `plotmarks`
- `arrows.meta`
- `decorations.markings`

Graphs are created in a `luadraw` environment, which calls `luacode`, so the **lua language** must be used in this environment:

```
\begin{luadraw}{ name=<filename>, exec=true/false, auto=true/false }
local ld = luadraw
-- create a new graph with a local name (this line is a comment)
local g = ld.graph:new{ window={x1,x2,y1,y2 [,xscale,yscale]}, margin={top,right,bottom,left},
size={width,height,ratio}, bg="color", border=true/false }
-- build the g chart
graphic instructions in Lua language ...
-- display the g chart and save it in the <filename>.tkz file
g:Show()
-- or save it only in the <filename>.tkz file
g:Save()
\end{luadraw}
```

**Saving the \*.tkz file** : the chart is exported in TikZ format to a file (with the `tkz` extension). By default, it is saved in the `_luadraw` folder, which is a subfolder of the current folder (containing the master document), but it is possible to specify a path to another subfolder. with the global option `cachedir=...`

### 3) `luadraw` Environment Options

These are:

- `name=...`: Allows you to name the produced TikZ file. It is given a name without an extension (this will be automatically added; it is `.tkz`). If this option is omitted, then a default name is the name of the master file followed by a number.
- `exec=true/false`: Allows you to execute or not the Lua code included in the environment. By default, this option is true, **UNLESS** if the global option `noexec` was mentioned in the preamble with the package declaration. When a complex graph that requires a lot of calculations is ready, it may be useful to add the `exec=false` option. This will prevent recalculations of the same graph for future compilations.
- `auto=true/false`: Allows you to automatically include or exclude the TikZ file in place of the `luadraw` environment when the `exec` option is set to `false`. By default, the `auto` option is `true`.

#### 4) The `cpx` (complex numbers) class

Reminder: we use the alias `local cpx = ld.cpx`.

It is automatically loaded by the `luadraw_graph2d` module and therefore when the `luadraw` package is loaded. This class allows you to manipulate complex numbers and perform common calculations. We create a complex number with the function `cpx.Z(a, b)` for  $a + i \times b$ , or with the function `cpx.Zp(r, theta)` for  $r \times e^{i\theta}$  in polar coordinates.

- Example: `local z = cpx.Z(a, b)` will create the complex number corresponding to  $a + i \times b$  in the variable `z`. We then access the real and imaginary parts of `z` like this: `z.re` and `z.im`. Of course, you can create a shortcut to this function with the instruction `local Z = cpx.Z` at the beginning of the code.
- **Warning:** A real number  $x$  is not considered complex by Lua. However, the functions provided for graphical constructions perform the verification and conversion from real to complex. However, we can use `Z(x, 0)` instead of `x`.
- The usual operators have been overloaded, allowing the use of the usual symbols, namely: `+`, `x`, `-`, `/`, as well as the equality test with `=` (this equality test is performed to within `ld.epsilon`, where `ld.epsilon` is a global variable that defaults to  $1e - 16$ ). When a calculation fails, the returned result should normally be equal to `nil`. In addition, the following functions are added:
  - modulus: `cpx.abs(z)`,
  - modulus squared: `cpx.abs2(z)`,
  - normalization: `cpx.normalize(z)` (returns `nil` if `z` is null),
  - norm 1: `cpx.N1(z)`,
  - main argument: `cpx.arg(z)`,
  - conjugate: `cpx.bar(z)`,
  - perfect equality: `cpx.equal(z1, z2)`,
  - complex exponential: `cpx.exp(z)`,
  - the complex hyperbolic cosine and sine: `cpx.cosh(z)` and `cpx.sinh(z)`,
  - the power function  $a$ : `cmdcpx.pow(z, a)` (the calculation uses the main argument of `z`),
  - scalar product: `cpx.dot(z1, z2)`, where the complex numbers represent vector affixes,
  - determinant: `cpx.det(z1, z2)`,
  - the oriented angle (in radians) between two non-zero vectors: `cpx.angle(z1, z2)`
  - rounding: `cpx.round(z, number of decimals)`,
  - the function: `cpx.isNul(z)` tests whether the real and imaginary parts of `z` are in absolute value less than the global variable `ld.epsilon` which is equal to  $1e - 16$  by default,
  - the function `cpx.isComplex(u)` tests whether  $\langle u \rangle$  is complex or not and returns a boolean (note: if  $\langle u \rangle$  is real, the function returns `false`),
  - the function `cpx.toComplex(x)` tests if  $\langle x \rangle$  is a real number, if so, it returns the complex number `cpx.Z(x, 0)`, if  $\langle x \rangle$  is a complex number the function returns  $\langle x \rangle$ , and in other cases, it returns `nil`,
  - the function `cpx.isobar(L)` where  $\langle L \rangle$  is a list of complex numbers, returns the isobarycenter of the points of  $\langle L \rangle$ .

We also have the constant `cpx.I` which represents the pure imaginary  $i$ .

Example:

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

The multiplication symbol is required.

## 5) Displaying a Variable in the Terminal

The instruction `ld.whatis(variable [, msg])` displays the type of the  $\langle variable \rangle$  and its contents in the terminal during compilation. Recognized types include the predefined types plus: *complex number*, *list of (complex) numbers*, and *list of lists of (complex) numbers*. The argument *msg* is an optional string (empty by default) which is displayed with the type to locate the variable in the terminal.

## 6) Creating a Graph

As seen above, creation is done in a *luadraw* environment. This creation is done by naming the graph:

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

The *ld.graph* class is defined in the *luadraw* package. This class is instantiated by invoking its constructor and giving it a name (here it's *g*). This is done locally so that the graph *g* thus created will no longer exist once it leaves the environment (otherwise *g* would remain in memory until the end of the document). Here are the options for the constructor **ld.graph:new**:

- **window={x1,x2,y1,y2 [,xscale,yscale]}**. This option (optional) defines the  $\mathbf{R}^2$  tile in which the graph is plotted (it is  $[x_1;x_2] \times [y_1;y_2]$ ), as well as the axis scales, which are the parameters  $\langle xscale \rangle$  and  $\langle yscale \rangle$ . These last two parameters are optional and default to 1; they represent the scale (cm per unit) on the axes. By default, we have **window = {-5,5,-5,5,1,1}**.
- **margin={left,right,top,bottom}**. This option (optional) sets the margins around the graph in cm. By default, we have **margin={0.5,0.5,0.5,0.5}**.
- **size={width,height [,ratio]}**. This option (optional) allows you to impose a size (in cm, including margins) for the graph. The  $\langle ratio \rangle$  argument is optional and corresponds to the desired scale ratio ( $\langle xscale \rangle / \langle yscale \rangle$ ). A ratio of 1 will result in an orthonormal coordinate system, and if the ratio is not specified, the default ratio (defined with the **window** option) is retained. A ratio of 0 determines the scales so that the graph is exactly the requested size. Using this parameter will modify the values of  $\langle xscale \rangle$  and  $\langle yscale \rangle$ . By default, the size is  $11 \times 11$  (in cm) with margins ( $10 \times 10$  without margins).
- **bg="color"**. This option (optional) allows you to define a background color for the graph. This color is a string representing a color for TikZ. By default, this string is empty, meaning the background will not be painted.
- **border=true/false**. This option (optional) indicates whether or not a frame should be drawn around the graph (including the margins). By default, this parameter is set to **false**.
- **bbox=true/false**. This option (optional) indicates whether a bounding box should be added to the graph so that it has the desired size. Everything outside of it is clipped by TikZ. By default, this parameter is set to **true**. With the value **false**, no bounding box is added, but everything outside the 2D window, except for the paths, is clipped by *luadraw*. The graph size can be smaller than the requested size.
- **pictureoptions=""**. This option (optional) is a string containing options that will be passed to the *tikzpicture* environment like this:

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

### Graph construction.

- The instantiated object (*g* in the example) has several methods for drawing (segments, lines, curves, etc.). Drawing instructions are not sent directly to T<sub>E</sub>X; they are stored as strings in a table that is a property of the *g* object. The **g:Show()** method will send these instructions to T<sub>E</sub>X while saving them in a text file.<sup>1</sup> The **g:Save()** method saves the graph in the file designated by the (environment) option **name** but without sending the instructions to T<sub>E</sub>X.

<sup>1</sup>This file will contain a *tikzpicture* environment.

- The current graph can be saved to another file with the **g:Savetofile(<filename with extension>)** method.
- A current graph can be reset, i.e., delete all elements already created, with the **g:Cleargraph()** method.
- The *luadraw* package also provides a number of mathematical functions, as well as functions for calculating lists (tables) of complex numbers, geometric transformations, etc. These functions are in the namespace *luadraw*.

### Coordinate system. Location

- The instantiated object (*g* in the example) has:
  1. An original view: this is the  $\mathbf{R}^2$  block defined by the **window** option at creation. This **must not be modified** subsequently.
  2. A current view: this is a  $\mathbf{R}^2$  block that must be included in the original view; anything outside this block will be clipped. By default, the current view is the original view. To retrieve the current view, you can use the **g:Getview()** method, which returns a table  $\{x1, x2, y1, y2\}$ , representing the block  $[x_1; x_2] \times [y_1; y_2]$ .
  3. A transformation matrix: this is initialized to the identity matrix. During a drawing instruction, the points are automatically transformed by this matrix before being sent to TikZ.
  4. A coordinate system (Cartesian coordinate system) linked to the current view; this is the user's coordinate system. By default, this is the canonical coordinate system of  $\mathbf{R}^2$ , but it is possible to change it. Let's say the current view is the  $[-5; 5] \times [-5; 5]$  block. It is possible, for example, to decide that this block represents the  $[-1; 12]$  interval for the abscissas and the  $[0; 8]$  interval for the ordinates. The method that makes this change will modify the graph's transformation matrix, so that for the user, everything happens as if they were in the  $[-1; 12] \times [0; 8]$  block. The intervals of the user's coordinate system can be retrieved using the methods:

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

- Complex numbers are used to represent points or vectors in the user's Cartesian coordinate system.
- In the TikZ export, the coordinates will be different because the lower left corner (excluding margins) will have coordinates (0,0), and the upper right corner (excluding margins) will have coordinates corresponding to the size (excluding margins) of the graph, and with 1 cm per unit on both axes. This means that normally, TikZ should only handle « small » numbers.
- The conversion is done automatically with the **g:strCoord(x, y)** method, which returns a string of the form  $(a,b)$ , where  $a$  and  $b$  are the coordinates for TikZ, or with the **g:Coord(z)** method, which also returns a string of the form  $(a,b)$  representing the TikZ coordinates of the point with affix  $z$  in the user's coordinate system.

## 7) Can we use TikZ directly in the *luadraw* environment?

Suppose we are creating a graph named *g* in a *luadraw* environment. It is possible to write a TikZ instruction during this creation, but not using `tex.sprint("<tikz instruction>")`, because then this instruction would not be part of the graph *g*. To do this, you must use the method **g:Writeln("<tikz instruction>")**, with the constraint that **the backslashes must be doubled**, and without forgetting that the graphic coordinates of a point in *g* are not the same for TikZ. For example:

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

Or to change styles:

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

In a Beamer presentation, this can also be used to insert pauses in a graph:

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

## II Graphics Methods

Polygonal lines, curves, paths, points, and labels can be created.

## 1) Polygonal Lines

### Dpolyline

A polygonal line is a list (table) of connected components, and a connected component is a list (table) of complex numbers that represent the affixes of the points. For example, the instruction:

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

creates a polygonal line with two components in a variable  $L$ .

**Drawing a polygonal line.** This is the

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

method (where  $g$  denotes the graphic being created), the argument  $\langle L \rangle$  is a polygonal line,  $\langle close \rangle$  is an optional argument equal to `true` or `false` indicating whether the line should be closed or not (`false` by default), and  $\langle draw\_options \rangle$  is a string that will be passed directly to the `\draw` instruction in the export (empty by default). The argument  $\langle clip \rangle$  must contain either `nil` (default value) or a table of the form  $\{x_1, x_2, y_1, y_2\}$ , in the first case the line is clipped by the current 2D window **after** its transformation by the 2D matrix of the graph, in the second case the line is clipped by the window  $[x_1; x_2] \times [y_1; y_2]$  **before** being transformed by the graph matrix.

### Options for drawing a polygonal line

Drawing options can be passed directly to the `\draw` instruction in the export, but they will have only a local effect. These options can be modified globally using the method

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

when one of the arguments is `nil`, its default value is used.

- $\langle color \rangle$  is a string representing a color known to TikZ (default is black),
- $\langle style \rangle$  is a string representing the type of line to draw. This style can be:
  - `"noline"`: undrawn line,
  - `"solid"`: solid line (default value),
  - `"dotted"`: dotted line,
  - `"dashed"`: dashed line,
  - custom style: the  $\langle style \rangle$  argument can be a string of the form (example): `"{2.5pt}{2pt}"` which means: a 2.5pt line followed by a 2pt space, the number of values can be greater than 2, e.g.: `"{2.5pt}{2pt}{1pt}{2pt}"` (succession of on, off).
- $\langle width \rangle$  is a number representing the line thickness expressed in tenths of a point, for example 8 for an actual thickness of 0.8pt (default value of 4),
- $\langle arrows \rangle$  is a string that specifies the type of arrow that will be drawn, this can be:
  - `"-"` which means no arrow (default value),
  - `"->"` which means an arrow at the end,
  - `"<-"` which means an arrow at the beginning,
  - `"<->"` which means an arrow at each end.

**WARNING:** The arrow is not drawn when the argument  $\langle close \rangle$  is `true`.

The options can be modified individually using the following methods:

- `g:Linecolor(color)`,
- `g:Linestyle(style)`,
- `g:Linewidth(width)`,



- **g:Arrows(arrows)**,
- plus the following methods:
  - **g:Lineopacity(opacity)**, which sets the opacity of the line drawing. The  $\langle opacity \rangle$  argument must be a value between 0 (fully transparent) and 1 (fully opaque). The default value is 1.
  - **g:Linecap(style)**, to adjust the line ends, the  $\langle style \rangle$  argument is a string that can be:
    - \* "butt" (straight end at the breakpoint, default value),
    - \* "round" (rounded semicircle end),
    - \* "square" (rounded square end).
  - **g:Linejoin(style)**, to adjust the join between segments, the  $\langle style \rangle$  argument is a string that can be:
    - \* "miter" (pointed corner, default value),
    - \* "round" (or rounded corner),
    - \* "bevel" (cut corner).

**Polygonal line fill options.** This is the

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

method (which uses the TikZ *patterns* library, which is loaded with the package). When one of the arguments is `nil`, its default value is used:

- $\langle color \rangle$  is a string representing a color known to TikZ (default is "black").
- $\langle style \rangle$  is a string representing the fill type. This style can be:
  - "none": no fill, this is the default value,
  - "full": full fill,
  - "bdiag": descending hatching from left to right,
  - "fdiag": ascending hatching from left to right,
  - "horizontal": horizontal hatching,
  - "\vertical": vertical hatching,
  - "hvcross": horizontal and vertical hatching,
  - "diagcross": descending and ascending diagonals,
  - "gradient": in this case, the fill is done with a gradient defined with the method **g:Gradstyle(string)**, this style is passed as is to the `\draw` instruction. By default, the string defining the gradient style is:
 

```
"left color=white,right color=red"
```
  - Any other style known from the *patterns* library is also possible.
  - $\langle opacity \rangle$ : a number between 0 and 1 (1 by default).
  - $\langle evenodd \rangle$ : a boolean indicating whether the *even odd rule* option should be used for fill (`false` by default).

Some options can be modified individually with the methods:

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

```
\begin{luadraw}{name=champ}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z
local g = ld.graph:new{window={-5,5,-5,5},bg="Cyan!30",size={10,10}}
local f = function(x,y) -- ode y' = 1-x*y^2=f(x,y)
  return 1-x*y^2
end
```

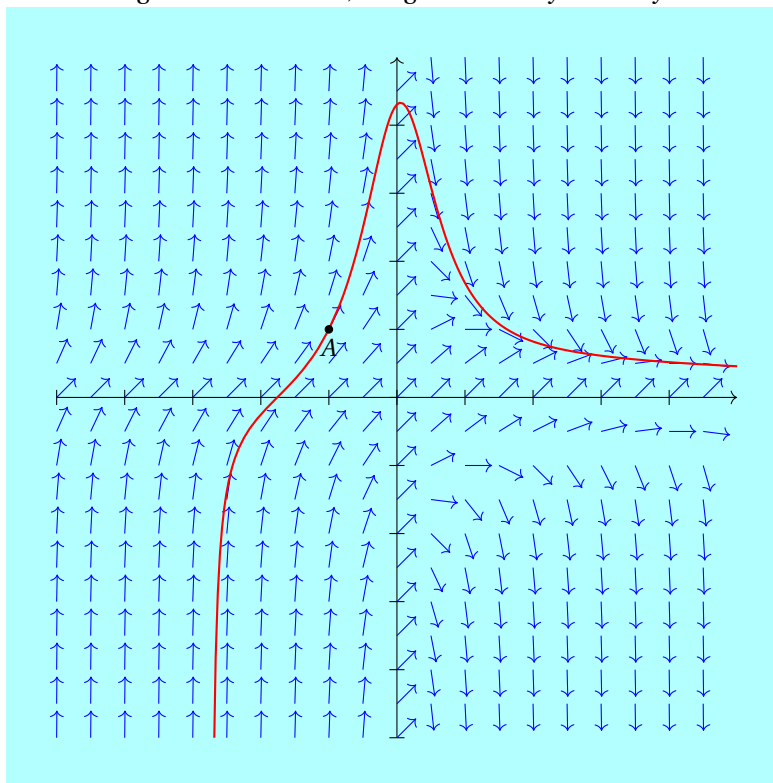


```

local A = Z(-1,1) -- A = -1+i
local deltaX, deltaY, long = 0.5, 0.5, 0.4
local champ = function(f)
  local vecteurs, v = {}
  for y = g:Yinf(), g:Ysup(), deltaY do
    for x = g:Xinf(), g:Xsup(), deltaX do
      v = Z(1,f(x,y)) -- coordinates 1 and f(x,y)
      v = v/cpx.abs(v)*long -- normalization of v
      table.insert(vecteurs, {Z(x,y), Z(x,y)+v} )
    end
  end
  return vecteurs -- vecteurs is a polygonal line
end

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

```

Figure 2: Vector field, integral curve of  $y' = 1 - xy^2$ 

## 2) Segments and Lines

In this section we will use the alias `local ld = luadraw`.

A segment is a list (table) of two complex numbers representing the endpoints. A line is a list (table) of two complex numbers, the first representing a point on the line, and the second a direction vector of the line (this must be non-zero).

### Dangle

- The `g:Dangle(B, A, C [, r, draw_options])` method draws the angle  $BAC$  with a parallelogram (only two sides are drawn). The optional argument  $\langle r \rangle$  specifies the length of one side (0.25 by default). The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the `\draw` instruction.
- The function `ld.angleD(B, A, C, r)` returns the list of points of this angle.

**Dbissec**

- The method **g:Dbissec(B, A, C, interior, draw\_options, scale)** draws a bisector of the geometric angle BAC, interior if the optional argument *interior* is **true** (default value), exterior otherwise. The argument *draw\_options* is a string (empty by default) that will be passed as is to the instruction `\draw`. The argument *scale* (which defaults to 1) is either a number (percentage) to vary the length of the displayed segment (from its midpoint), or an array of two numbers (percentages) {scaleA, scaleB} to vary the length of the displayed segment at each of its endpoints.
- The function **ld.bissec(B, A, C, interior)** returns this bisector as a list  $\{A, u\}$  where  $u$  is a direction vector of the line.

**Dhline**

The method **g:Dhline(d [, draw\_options, scale])** draws a half-line. The argument *d* must be a list of two complex numbers  $\{A, B\}$  variables. The half-line  $[A, B)$  is drawn.

Variant: **g:Dhline(A, B [, draw\_options, scale])**. The argument *draw\_options* is a string (empty by default) that will be passed as is to the `\draw` instruction. The argument *scale* (which defaults to 1) is either a number (percentage) to vary the length of the displayed segment (from its midpoint), or an array of two numbers (percentages) {scaleA, scaleB} to vary the length of the displayed segment at each of its endpoints.

**Dline**

The **g:Dline(d [, draw\_options, scale])** method draws the line *d*, which is a list of type  $\{A, u\}$  where  $A$  represents a point on the line (a complex number) and  $u$  a direction vector (a non-zero complex number).

Variant: the **g:Dline(A, B [, draw\_options, scale])** method draws the line passing through the points *A* and *B* (two complex numbers). The argument *draw\_options* is a string (empty by default) that will be passed as is to the `\draw` instruction. The argument *scale* (which defaults to 1) is either a number (percentage) to vary the length of the displayed segment (from its midpoint), or an array of two numbers (percentages) {scaleA, scaleB} to vary the length of the displayed segment at each of its endpoints.

**DlineEq**

- The **g:DlineEq(a, b, c [, draw\_options, scale])** method draws the line with the equation  $ax + by + c = 0$ . The *draw\_options* argument is a string (empty by default) that will be passed as is to the `\draw` instruction. The argument *scale* (which defaults to 1) is either a number (percentage) to vary the length of the displayed segment (from its midpoint), or an array of two numbers (percentages) {scaleA, scaleB} to vary the length of the displayed segment at each of its endpoints.
- The **ld.lineEq(a, b, c)** function returns the line with the equation  $ax + by + c = 0$  as a list  $\{A, u\}$  where  $A$  is a point on the line and  $u$  is a direction vector.

**Dmarkarc**

The method **g:Dmarkarc(b, a, c, r, n [, length, space, draw\_options])** marks the arc of a circle with center *a*, radius *r*, extending from *b* to *c*, with *n* small segments. By default, the *length* is 0.25, and the spacing (argument *space*) is 0.0625. The argument *draw\_options* is a string (empty by default) that will be passed as is to the instruction `\draw`.

**Dmarkseg**

The method **g:Dmarkseg(a, b, n [, length, space, angle, draw\_options])** marks the segment defined by *a* and *b* with *n* small segments inclined at *angle* degrees (45° by default). By default, the *length* is 0.25, and the spacing (argument *space*) is 0.125. The argument *draw\_options* is a string (empty by default) that will be passed as is to the instruction `\draw`.

**Dmed**

- The **g:Dmed(A, B [, draw\_options, scale])** method draws the perpendicular bisector of the segment  $[A; B]$ .

Variant: **g:Dmed(seg [, draw\_options, scale])** where *segment* is a list of two points representing the segment. The  $\langle draw\_options \rangle$  argument is a string (empty by default) that will be passed as is to the `\draw` instruction. The argument  $\langle scale \rangle$  (which defaults to 1) is either a number (percentage) to vary the length of the displayed segment (from its midpoint), or an array of two numbers (percentages)  $\{scaleA, scaleB\}$  to vary the length of the displayed segment at each of its endpoints.

- The function **ld.med(A, B)** (or **ld.med(seg)**) returns the perpendicular bisector of the segment  $[A; B]$  as a list  $\{C, u\}$  where  $C$  is a point on the line and  $u$  is a direction vector.

**Dparallel**

- The method **g:Dparallel(d, A [, draw\_options, scale])** draws the parallel to  $\langle d \rangle$  passing through  $\langle A \rangle$ . The argument  $\langle d \rangle$  can be either a line (a list consisting of a point and a direction vector) or a non-zero vector. The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the instruction `\draw`. The argument  $\langle scale \rangle$  (which defaults to 1) is either a number (percentage) to vary the length of the displayed segment (from its midpoint), or an array of two numbers (percentages)  $\{scaleA, scaleB\}$  to vary the length of the displayed segment at each of its endpoints.
- The function **ld.parallel(d, A)** returns the parallel to  $\langle d \rangle$  passing through  $\langle A \rangle$  as a list  $\{B, u\}$  where  $B$  is a point on the line and  $u$  is a direction vector.

**Dperp**

- The method **g:Dperp(d, A [, draw\_options, scale])** draws the perpendicular to  $\langle d \rangle$  passing through  $\langle A \rangle$ . The argument  $\langle d \rangle$  can be either a line (a list consisting of a point and a direction vector) or a non-zero vector. The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the instruction `\draw`. The argument  $\langle scale \rangle$  (which defaults to 1) is either a number (percentage) to vary the length of the displayed segment (from its midpoint), or an array of two numbers (percentages)  $\{scaleA, scaleB\}$  to vary the length of the displayed segment at each of its endpoints.
- The function **ld.perp(d, A)** returns the perpendicular to  $\langle d \rangle$  passing through  $\langle A \rangle$  as a list  $\{B, u\}$  where  $B$  is a point on the line and  $u$  is a direction vector.

**Dseg**

The **g:Dseg(seg [, scale, draw\_options])** method draws the segment defined by the  $\langle seg \rangle$  argument, which must be a list of two complex numbers. The  $\langle draw\_options \rangle$  argument is a string (empty by default) that will be passed as is to the `\draw` instruction. The argument  $\langle scale \rangle$  (which defaults to 1) is either a number (percentage) to vary the length of the displayed segment (from its midpoint), or an array of two numbers (percentages)  $\{scaleA, scaleB\}$  to vary the length of the displayed segment at each of its endpoints.

**Dtangent**

- The **g:Dtangent(p, t0 [, length, draw\_options])** method draws the tangent to the curve parameterized by  $\langle p \rangle: t \mapsto p(t)$  (with complex values), at the parameter point  $\langle t0 \rangle$ . If the argument  $\langle long \rangle$  is `nil` (the default value), then the entire line is drawn; otherwise, it is a segment of  $\langle length \rangle$ . The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the `\draw` instruction.
- The function **ld.tangent(p, t0 [, length])** returns either the line or a segment (depending on whether  $\langle long \rangle$  is `nil` or not).

**DtangentC**

- The method **g:DtangentC(f, x0 [, length, draw\_options])** draws the tangent to the Cartesian curve with equation  $y = f(x)$ , at the abscissa point  $\langle x0 \rangle$ . If the argument  $\langle length \rangle$  is **nil** (the default value), then the entire line is drawn; otherwise, it is a segment of  $\langle length \rangle$ . The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the instruction `\draw`.
- The function **ld.tangentC(f, x0 [, length])** returns either the line or a segment (depending on whether  $\langle long \rangle$  is **nil** or not).

**DtangentI**

- The method **g:DtangentI(f, x0, y0 [, length, draw\_options])** draws the tangent to the implicit curve with equation  $f(x, y) = 0$ , at the assumed point  $\langle (x0, y0) \rangle$  on the curve. If the argument  $\langle length \rangle$  is **nil** (the default value), then the entire line is drawn; otherwise, it is a segment of  $\langle length \rangle$ . The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the instruction `\draw`.
- The function **ld.tangentI(f, x0, y0 [, length])** returns either the line or a segment (depending on whether  $\langle long \rangle$  is **nil** or not).

**Dtangent\_from**

- The method **g:Dtangent\_from(A, p, t1, t2 [, dp, draw\_options, out, scale])** draws the tangent(s) to the curve parameterized by  $\langle p \rangle$ :  $t \mapsto p(t)$  (with complex values) originating from the point  $\langle A \rangle$  (a complex number). The arguments  $\langle t1 \rangle$  and  $\langle t2 \rangle$  represent the bounds of the search interval. The optional argument  $\langle dp \rangle$  is a function representing the derivative of the function  $\langle p \rangle$ ; by default, this argument is **nil**, and the derivative of  $\langle p \rangle$  is replaced by an approximation. The optional argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the `\draw` instruction. The optional argument  $\langle out \rangle$ , if used, must be the name of a variable. This variable must be an array, and at the end of execution, it will contain the points on the curve for which the tangent passes through  $\langle A \rangle$ . The argument  $\langle scale \rangle$  (which defaults to 1) is either a number (percentage) to vary the length of the displayed segment (from its midpoint), or an array of two numbers (percentages) {scaleA, scaleB} to vary the length of the displayed segment at each of its endpoints.
- The function **ld.tangent\_from(A, p, t1, t2 [, dp])** returns the list of points on the curve parameterized by  $\langle p \rangle$ , on the interval  $[t_1, t_2]$ , for which the tangent passes through  $\langle A \rangle$  (a complex number). If there are no points, the function returns an empty list.

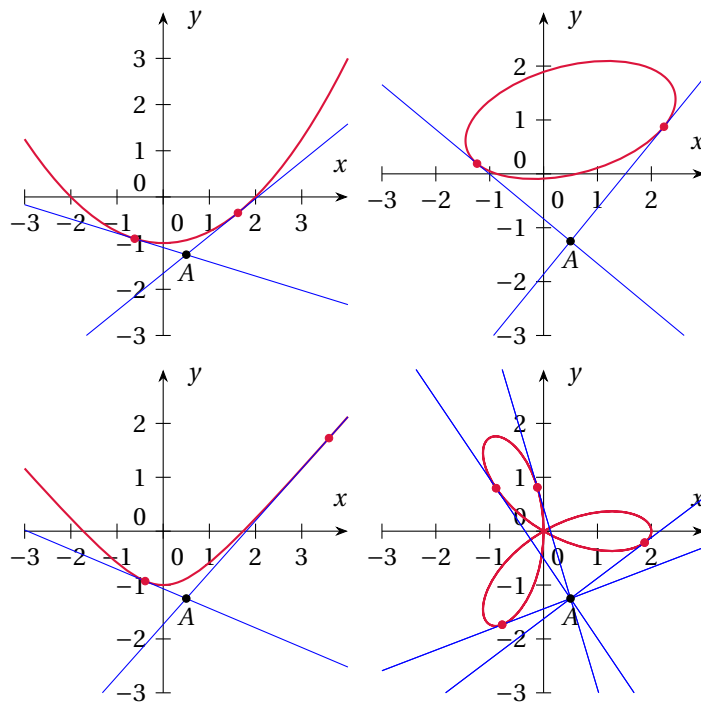
```

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

```

```
\end{luadraw}
```

Figure 3: Tangents from a point



### Dnormal

- The method **g:Dnormal(p, t0 [, long, draw\_options])** draws the normal to the curve parameterized by  $\langle p \rangle: t \mapsto p(t)$  (with complex values), at the point parameter  $\langle t0 \rangle$ . If the argument  $\langle long \rangle$  is **nil** (the default value), then the entire line is drawn; otherwise, it is a segment of length  $\langle long \rangle$ . The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the instruction `\draw`.
- The function **ld.normal(p, t0 [, long])** returns either the line or a segment (depending on whether  $\langle long \rangle$  is **nil** or not).

### DnormalC

- The **g:DnormalC(f, x0 [, long, draw\_options])** method draws the normal to the Cartesian curve with equation  $y = f(x)$ , at the abscissa point  $\langle x0 \rangle$ . If the argument  $\langle long \rangle$  is **nil** (the default value), then the entire line is drawn; otherwise, it is a segment of length  $\langle long \rangle$ . The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the `\draw` instruction.
- The function **ld.normalC(f, x0 [, long])** returns either the line or a segment (depending on whether  $\langle long \rangle$  is **nil** or not).

### DnormalI

- The **g:DnormalI(f, x0, y0 [, long, draw\_options])** method draws the normal to the implicit curve with equation  $f(x, y) = 0$ , at the assumed point  $\langle (x0, y0) \rangle$  on the curve. If the  $\langle long \rangle$  argument is **nil** (the default value), then the entire line is drawn; otherwise, it is a segment of length  $\langle long \rangle$ . The  $\langle draw\_options \rangle$  argument is a string (empty by default) that will be passed as is to the `\draw` instruction.
- The function **ld.normalI(f, x0, y0, long)** returns either a line or a segment (depending on whether  $\langle long \rangle$  is **nil** or not).

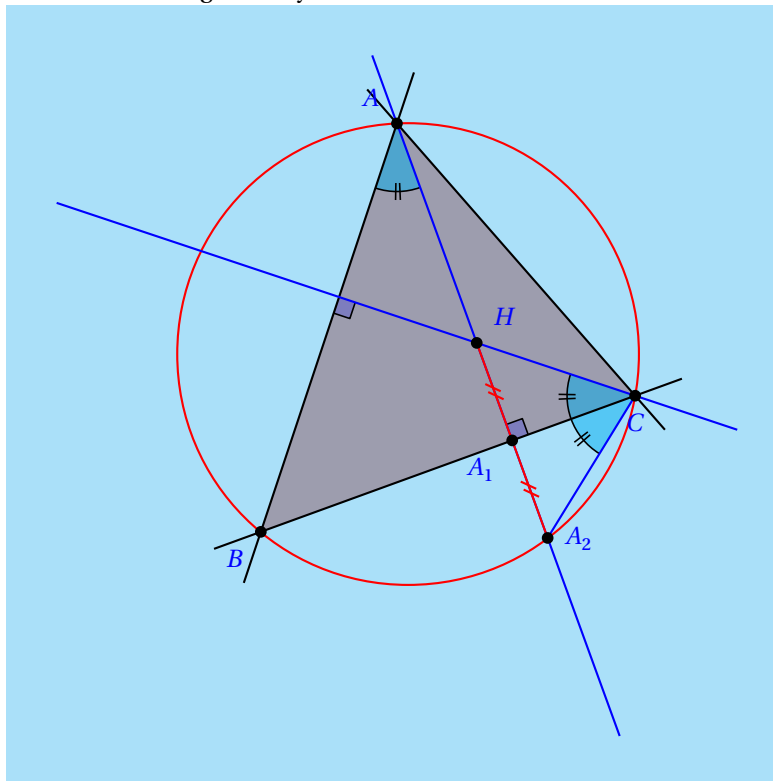
```
\begin{luadraw}{name=orthocentre}
local ld = luadraw
```

```

local cpx = ld.cpx
local g = ld.graph:new{window={-5,5,-5,5},bg="cyan!30",size={10,10}}
local i = cpx.I
local A, B, C = 4*i, -2-2*i, 3.5
local h1, h2 = ld.perp({B,C-B},A), ld.perp({A,B-A},C) -- altitudes of the triangle
local A1, F = ld.proj(A,{B,C-B}), ld.proj(C,{A,B-A}) -- projected
local H = ld.interD(h1,h2) -- orthocenter
local A2 = 2*A1-H -- symmetric to H with respect to BC
g:Dpolyline({A,B,C},true, "draw=none,fill=Maroon,fill opacity=0.3") -- background of the triangle
g:Linewidth(6); g:Filloptions("full", "blue", 0.2)
g:Dangle(C,A1,A,0.25); g:Dangle(B,F,C,0.25) -- right angles
g:Linecolor("black"); g:Filloptions("full","cyan",0.5)
g:Darc(H,C,A2,1); g:Darc(B,A,A1,1) -- arcs
g:Filloptions("none","black",1) -- The opacity is restored to 1
g:Dmarkarc(H,C,A1,1,2); g:Dmarkarc(A1,C,A2,1,2) -- markers
g:Dmarkarc(B,A,H,1,2)
g:Linewidth(8); g:Linecolor("black")
g:Dseg({A,B},1.25); g:Dseg({C,B},1.25); g:Dseg({A,C},1.25) -- sides
g:Linecolor("red"); g:Dcircle(A,B,C) -- circle
g:Linecolor("blue"); g:Dline(h1); g:Dline(h2) -- altitudes of the triangle
g:Dseg({A2,C}); g:Linecolor("red"); g:Dseg({H,A2}) -- segments
g:Dmarkseg(H,A1,2); g:Dmarkseg(A1,A2,2) -- markers
g:Labelcolor("blue") -- for the label
g:Dlabel("$A$",A, {pos="NW",dist=0.1}, "$B$",B, {pos="SW"}, "$A_2$",A2,{pos="E"},
"$C$", C, {pos="S"}, "$H$", H, {pos="NE"}, "$A_1$", A1, {pos="SW"})
g:Linecolor("black"); g:Filloptions("full"); g:Ddots({A,B,C,H,A1,A2})
g:Show()
\end{luadraw}

```

Figure 4: Symmetric of the orthocenter



### 3) Geometric Figures

#### Darc

- The **g:Darc(B, A, C, r [, sens, draw\_options])** method draws an arc of a circle with center  $\langle A \rangle$  (complex number), radius  $\langle r \rangle$ , going from  $\langle B \rangle$  (complex number) to  $\langle C \rangle$  (complex) counterclockwise if the argument  $\langle sens \rangle$  is 1 (default value), and counterclockwise otherwise. The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the `\draw` instruction.

- The **ld.arc(B, A, C, r, sens)** function returns the list of points on this arc (polygonal line).
- The function **ld.arcb(B, A, C, r, sens)** returns this arc as a path (see *Dpath* page 37) using Bézier curves.

### Dcircle

- The method **g:Dcircle(c, r [, d, draw\_options])** draws a circle. When the argument  $\langle d \rangle$  is `nil`, it is the circle with center  $\langle c \rangle$  (complex number) and radius  $\langle r \rangle$ ; when  $\langle d \rangle$  is specified (complex number), it is the circle passing through the affix points  $\langle c \rangle$ ,  $\langle r \rangle$ , and  $\langle d \rangle$ . The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the instruction `\draw`. Another possible syntax: **g:Dcircle(C [, draw\_options])** where  $\langle C \rangle = \{c, r[, d]\}$ .
- The **ld.circle(c, r [, d])** function returns the list of points on this circle (polygonal line).
- The function **ld.circle({c, r [, d]}, nbdots)** returns the list of points of this circle (polygonal line) with the  $\langle nbdots \rangle$  points.
- The **ld.circleb(c, r [, d])** function returns this circle as a path (see *Dpath* page 37) using Bézier curves.

### Dellipse

- The **g:Dellipse(c, rx, ry [, inclin, draw\_options])** method draws the ellipse centered at  $\langle c \rangle$  (complex number). The arguments  $\langle rx \rangle$  and  $\langle ry \rangle$  specify the two radii (on  $x$  and  $y$ ). The optional argument  $\langle inclin \rangle$  is an angle in degrees that indicates the inclination of the ellipse relative to the  $Ox$  axis (zero by default). The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the `\draw` instruction.
- The **ld.ellipse(c, rx, ry [, inclin])** function returns the list of points on this ellipse (polygonal line).
- The function **ld.ellipse({c, rx, ry [, inclin]}, nbdots)** returns the list of points of this ellipse (polygonal line) with the  $\langle nbdots \rangle$  points.
- The function **ld.ellipseb(c, rx, ry [, inclination])** returns this ellipse as a path (see *Dpath* page 37) using Bézier curves.

### Dellipticarc

- The method **g:Dellipticarc(B, A, C, rx, ry [, sens, inclination, draw\_options])** draws an arc of an ellipse centered at  $\langle A \rangle$  (complex number) and radii at  $\langle rx \rangle$  and  $\langle ry \rangle$ , making an angle equal to  $\langle inclination \rangle$  with respect to the  $Ox$  axis (zero by default), going from  $\langle B \rangle$  (complex number) to  $\langle C \rangle$  (complex number) in the counterclockwise direction if the argument  $\langle sens \rangle$  is 1 (default value), and the opposite direction otherwise. The  $\langle draw\_options \rangle$  argument is a string (empty by default) that will be passed as is to the `\draw` instruction.
- The **ld.ellipticarc(B, A, C, rx, ry [, sens, inclination])** function returns the list of points of this arc (polygonal line).
- The **ld.ellipticarcb(B, A, C, rx, ry [, sens, inclination])** function returns this arc as a path (see *Dpath* page 37) using Bézier curves.

### Dparallelogram

- The method **g:Dparallelogram(a, u, v [, draw\_options])** where  $\langle a \rangle$ ,  $\langle u \rangle$ ,  $\langle v \rangle$  represent three complex numbers denoting a vertex and two vectors, draws the parallelogram with vertices  $\{a, a + u, a + u + v, a + v\}$ . The argument  $\langle draw\_options \rangle$  is a string (empty by default) which will be passed as is to the `\draw` instruction.
- The function **ld.parallelogram(a, u, v)** returns the list of vertices of this parallelogram.

### Dpolyreg

- The **g:Dpolyreg(vertex1, vertex2, nbides, direction [, draw\_options])** or **g:Dpolyreg(center, vertex, nbides [, draw\_options])** method draws a regular polygon. The  $\langle draw\_options \rangle$  argument is a string (empty by default) that will be passed as is to the `\draw` instruction.
- The **ld.polyreg(vertex1, vertex2, nbides, direction)** function and the **ld.polyreg(center, vertex, nbides)** function return the list of vertices of this regular polygon.



## Drectangle

- The **g:Drectangle(a, b, c [, draw\_options])** method draws a rectangle with consecutive vertices  $\langle a \rangle$  and  $\langle b \rangle$ , and whose opposite side passes through  $\langle c \rangle$  (complex numbers). The  $\langle draw\_options \rangle$  argument is a string (empty by default) that will be passed as is to the  $\backslash draw$  instruction.

The **ld.rectangle(a, b, c)** function returns the list of vertices of this rectangle.

- The method **g:Drectangle(a, b [, draw\_options])** draws the rectangle whose sides are parallel to the axes and whose opposite vertices are  $\langle a \rangle$  and  $\langle b \rangle$ . The argument  $\langle draw\_options \rangle$  is a string (empty by default) which is passed as is to the  $\backslash draw$  instruction.

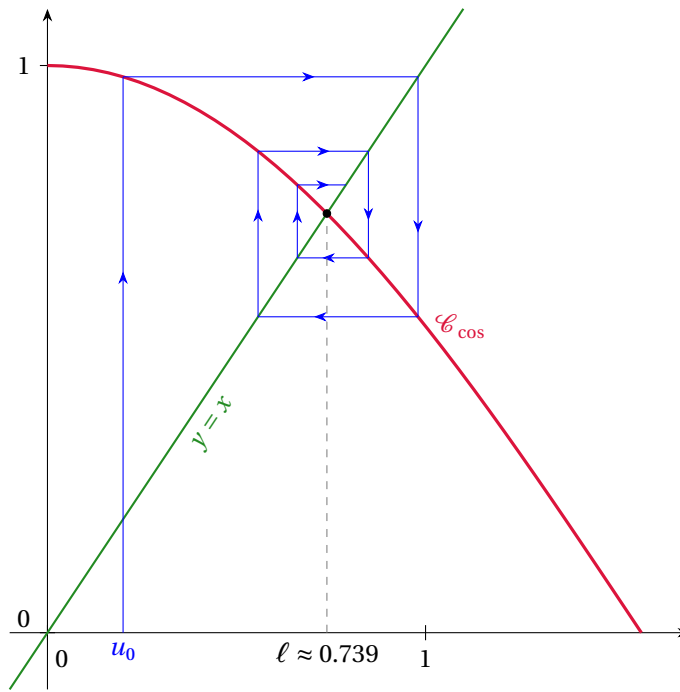
The function **ld.rectangle(a, b)** returns the list of vertices of this rectangle.

## Dsequence

- The **g:Dsequence(f, u0, n, draw\_options)** method draws the "staircases" of the recurrent sequence defined by its first term  $\langle u_0 \rangle$  and the relation  $u_{k+1} = f(u_k)$ . The argument  $\langle f \rangle$  must be a function of a real-valued variable, and the argument  $\langle n \rangle$  is the number of terms calculated. The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the  $\backslash draw$  instruction.
- The **ld.sequence(f, u0, n)** function returns the list of points constituting these "staircases".

```
\begin{luadraw}{name=sequence}
local ld = luadraw
local cpx = ld.cpx
local g = ld.graph:new{window={-0.1,1.7,-0.1,1.1},size={10,10,0}}
local i, pi, cos = cpx.I, math.pi, math.cos
local f = function(x) return cos(x)-x end
local ell = ld.solve(f,0,pi/2)[1]
local L = ld.sequence(cos,0.2,5) -- u_{n+1}=cos(u_n), u_0=0.2
local seg, z = {}, L[1]
for k = 2, #L do
    table.insert(seg,{z,L[k]})
    z = L[k]
end -- seg is a list of segments (staircases)
local styleA = "\\tikzset{->/.style={decoration={markings, mark="
local styleB = "at position #1 with {\\arrow{Stealth}}}, postaction={decorate}}}"
g:Writeln(styleA..styleB)
g:Daxes({0,1,1}, {arrows="-Stealth"})
g:DlineEq(1,-1,0,"line width=0.8pt,ForestGreen")
g:Dcartesian(cos, {x={0,pi/2},draw_options="line width=1.2pt,Crimson"})
g:Dpolyline(seg,false,"->=0.65,blue")
g:Dlabel("$u_0$",0.2,{pos="S",node_options="blue"})
g:Dseg({ell, ell*(1+i)},1,"dashed,gray")
g:Dlabel("$\\ell\\approx"..ld.round(ell,3).."$", ell,{pos="S"})
g:Ddots(ell*(1+i)); g:Labelcolor("Crimson")
g:Dlabel("$\\mathcal{C}_{\\cos}$",1+i*cos(1),{pos="E"})
g:Labelcolor("ForestGreen"); g:Labelangle(g:Arg(1+i)*180/pi)
g:Dlabel("$y=x$",0.4+i*0.4,{pos="S",dist=0.1})
g:Show()
\\end{luadraw}
```



Figure 5: Sequence  $u_{n+1} = \cos(u_n)$ 

The **g:Arg(z)** method calculates and returns the *real* argument of the complex  $z$ , that is, its argument (in radians) when exported to the TikZ coordinate system (to do this, you must apply the graph's transformation matrix to  $z$ , then convert the coordinate system to that of TikZ). If the graph coordinate system is orthonormal and the transformation matrix is the identity matrix, then the result is identical to that of `cpx.arg(z)` (this is not the case in the example above).

Similarly, the **g:Abs(z)** method calculates and returns the *real* modulus of the complex  $z$ , that is, its modulus when exported to the TikZ coordinate system; it is therefore a length in centimeters. If the graph coordinate system is orthonormal with 1 cm per unit on each axis, and if the transformation matrix is an isometry, then the result is identical to that of `cpx.abs(z)`.

### Dsquare

- The **g:Dsquare(a, b [, sens, draw\_options])** method draws the square with consecutive vertices  $\langle a \rangle$  and  $\langle b \rangle$  (complex numbers), in the counterclockwise direction when  $\langle sens \rangle$  is 1 (the default value). The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the `\draw` instruction.
- The **ld.square(a, b, sens)** function returns the list of vertices of this square.

### Dwedge

The method **g:Dwedge(B, A, C, r [, sens, draw\_options])** draws an angular sector with center  $\langle A \rangle$  (complex number), radius  $\langle r \rangle$ , going from  $\langle B \rangle$  (complex number) to  $\langle C \rangle$  (complex number) counterclockwise if the argument  $\langle sens \rangle$  is 1, and counterclockwise otherwise. The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the instruction `\draw`.

## 4) Curves

### Parametric: Dparametric

- The function **ld.parametric(p, t1, t2 [, nbdots, discount, nbdiv])** calculates the points of the curve and returns a polygonal line (a list of lists of complex numbers, no drawing).
  - The argument  $\langle p \rangle$  is the parameterization; it must be a function of a real variable  $t$  and complex-valued variables, for example: `local p = function(t) return cpx.exp(t*cpx.I) end`
  - The arguments  $\langle t1 \rangle$  and  $\langle t2 \rangle$  are required with  $t_1 < t_2$ ; they form the bounds of the interval for the parameter.

- The argument  $\langle nbdots \rangle$  is optional; it is the (minimum) number of points to calculate; it is 40 by default.
  - The argument  $\langle discount \rangle$  is an optional Boolean that indicates whether there are discontinuities or not; it is `false` by default.
  - The argument  $\langle nbdiv \rangle$  is a positive integer that is 5 by default and indicates the number of times the interval between two consecutive values of the parameter can be cut in two (dichotomized) when the corresponding points are too far apart.
- The method **g:Dparametric(p, options)** calculates the points and draws the curve parametrized by  $\langle p \rangle$ . The parameter  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:
    - `t= {g:Xinf(),g:Xsup()}`, this is the interval for the parameter  $t$  (by default, it's the entire range of the window's x-coordinates),
    - `nbdots=40`,
    - `discount=false`,
    - `nbdiv=5`,
    - `draw_options=""`, string which will be passed as is to the instruction `\draw`,
    - `clip=nil`, this option is either `nil` (default value) or a table `{x1,x2,y1,y2}`. In the first case, the line is clipped by the current 2D window **after** its transformation by the graph's 2D matrix. In the second case, the line is clipped by the window  $[x_1; x_2] \times [y_1; y_2]$  **before** being transformed by the graph's matrix.

### Polars: Dpolar

- The function **ld.polar(rho, t1, t2 [, nbdots, discount, nbdiv])** calculates the points of the curve and returns a polygonal line (no drawing). The argument  $\langle rho \rangle$  is the polar parameterization of the curve; it must be a function of a real variable  $t$  and with real values, for example:
 

```
local rho = function(t) return 4*math.cos(2*t) end
```

 The other arguments are identical to those for parameterized curves.
- The method **g:Dpolar(rho, options)** calculates the points and draws the polar curve parameterized by  $\rho$ . The parameter  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:
  - `t= {-pi,pi}`, this is the interval for the parameter  $t$ ,
  - `nbdots=40`,
  - `discount=false`,
  - `nbdiv=5`,
  - `draw_options=""`, string which will be passed as is to the instruction `\draw`,
  - `clip=nil`, this option is either `nil` (default value) or a table `{x1,x2,y1,y2}`. In the first case, the line is clipped by the current 2D window **after** its transformation by the graph's 2D matrix. In the second case, the line is clipped by the window  $[x_1; x_2] \times [y_1; y_2]$  **before** being transformed by the graph's matrix.

### Cartesian: Dcartesian

- The function **ld.cartesian(f, x1, x2 [, nbdots, discount, nbdiv])** calculates the points and returns a polygonal line (no drawing). The argument  $\langle f \rangle$  is the function whose curve we want to obtain. It must be a function of a real variable  $x$  and with real values, for example:
 

```
local f = function(x) return 1+3*math.sin(x)*x end
```

 The arguments  $\langle x1 \rangle$  and  $\langle x2 \rangle$  are required and form the bounds of the interval for the variable. The other arguments are identical to those for parametric curves.
- The method **g:Dcartesian(f, options)** calculates the points and draws the curve of  $f$ . The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:

- `x= {g:Xinf(),g:Xsup()}`, this is the interval for the parameter  $t$  (by default, it's the entire range of the window's x-coordinates),
- `nbdots=40`,
- `discont=false`,
- `nbdiv=5`,
- `draw_options=""`, string which will be passed as is to the instruction `\draw`,
- `clip=nil`, this option is either `nil` (default value) or a table `{x1,x2,y1,y2}`. In the first case, the line is clipped by the current 2D window **after** its transformation by the graph's 2D matrix. In the second case, the line is clipped by the window  $[x_1;x_2] \times [y_1;y_2]$  **before** being transformed by the graph's matrix.

### Periodic Functions: Dperiodic

- The function `ld.periodic(f, period, x1, x2 [, nbdots, discont, nbdiv])` calculates the points of the curve and returns a polygonal line (no drawing).
  - The argument  $\langle f \rangle$  is the function whose curve we want; it must be a function of a real variable  $x$  and with real values.
  - The argument  $\langle period \rangle$  is a table of the type  $\{a, b\}$ , with  $a < b$  representing a period of the function  $\langle f \rangle$ .
  - The arguments  $\langle x1 \rangle$  and  $\langle x2 \rangle$  are required and form the bounds of the interval for the variable.
  - The other arguments are identical to those for parametric curves.
- The method `g:Dperiodic(f, period, options)` calculates the points and draws the curve of  $\langle f \rangle$ . The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:
  - `x= {g:Xinf(),g:Xsup()}`, this is the interval for the parameter  $t$  (by default, it's the entire range of the window's x-coordinates),
  - `nbdots=40`,
  - `discont=false`,
  - `nbdiv=5`,
  - `draw_options=""`, string which will be passed as is to the instruction `\draw`,
  - `clip=nil`, this option is either `nil` (default value) or a table `{x1,x2,y1,y2}`. In the first case, the line is clipped by the current 2D window **after** its transformation by the graph's 2D matrix. In the second case, the line is clipped by the window  $[x_1;x_2] \times [y_1;y_2]$  **before** being transformed by the graph's matrix.

### Step Functions: Dstepfunction

- The `ld.stepfunction(def, discont)` function calculates the points of the curve and returns a polygonal line (no drawing).
  - The  $\langle def \rangle$  argument defines the step function; it is a table with two elements:
 

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

The first element  $\{x_1, x_2, x_3, \dots, x_n\}$  must be a subdivision of the segment  $[x_1; x_n]$ .

The second element  $\{c_1, c_2, \dots\}$  is the list of constants, with  $c_1$  for the segment  $[x_1; x_2]$ ,  $c_2$  for the segment  $[x_2; x_3]$ , etc.
  - The argument  $\langle discont \rangle$  is a Boolean that defaults to `true`.
- The method `g:Dstepfunction(def, options)` calculates the points and draws the curve of the step function.
  - The argument  $\langle def \rangle$  is the same as the one described above (definition of the step function).
  - The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:
    - \* `discont=true`,

- \* `draw_options=""`, string which will be passed as is to the instruction `\draw`,
- \* `clip=nil`, this option is either `nil` (default value) or a table `{x1,x2,y1,y2}`. In the first case, the line is clipped by the current 2D window **after** its transformation by the graph's 2D matrix. In the second case, the line is clipped by the window  $[x_1;x_2] \times [y_1;y_2]$  **before** being transformed by the graph's matrix.

### Piecewise Affine Functions: Daffinebypiece

- The function **ld.affinebypiece(def, discount)** calculates the points of the curve and returns a polygonal line (no drawing).
  - The argument  $\langle def \rangle$  defines the step function; it is a two-field table:  

```
{ {x1,x2,x3,...,xn}, { {a1,b1}, {a2,b2},...} }
```

The first element  $\{x_1, x_2, x_3, \dots, x_n\}$  must be a subdivision of the segment  $[x_1; x_n]$ .  
The second element  $\{ \{a_1, b_1\}, \{a_2, b_2\}, \dots \}$  means that on  $[x_1; x_2]$  the function is  $x \mapsto a_1x + b_1$ , on  $[x_2; x_3]$  the function is  $x \mapsto a_2x + b_2$ , etc.
  - The argument  $\langle discount \rangle$  is a boolean that defaults to `true`.
- The method **g:Daffinebypiece(def, options)** calculates the points and draws the curve of the piecewise affine function.
  - The argument  $\langle def \rangle$  is the same as the one described above (definition of the piecewise affine function).
  - The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:
    - \* `discount=true`,
    - \* `draw_options=""`, string which will be passed as is to the instruction `\draw`,
    - \* `clip=nil`, this option is either `nil` (default value) or a table `{x1,x2,y1,y2}`. In the first case, the line is clipped by the current 2D window **after** its transformation by the graph's 2D matrix. In the second case, the line is clipped by the window  $[x_1; x_2] \times [y_1; y_2]$  **before** being transformed by the graph's matrix.

### Differential Equations: Dodesolve

- The function **ld.odesolve(f, t0, Y0, tmin, tmax, nbdots [, method])** allows an approximate solution of the differential equation  $Y'(t) = f(t, Y(t))$  in the interval  $[t_{\min}; t_{\max}]$  which must contain  $\langle t0 \rangle$ , with the initial condition  $Y(t_0) = Y_0$ .
  - The argument  $\langle f \rangle$  is a function  $f : (t, Y) \rightarrow f(t, Y)$  with values in  $\mathbf{R}^n$  and where  $Y$  is also in  $\mathbf{R}^n$ :  $Y = \{y_1, y_2, \dots, y_n\}$ , but when  $n = 1$ ,  $Y$  is a real number.
  - The arguments  $\langle t0 \rangle$  and  $\langle Y0 \rangle$  give the initial conditions with  $Y_0 = \{y_1(t_0), \dots, y_n(t_0)\}$  (the  $y_i$  numbers are real), or  $Y_0 = y_1(t_0)$  when  $n = 1$ .
  - The arguments  $\langle tmin \rangle$  and  $\langle tmax \rangle$  define the resolution interval; this must contain  $\langle t0 \rangle$ .
  - The argument  $\langle nbdots \rangle$  indicates the number of points calculated on either side of  $\langle t0 \rangle$ .
  - The optional argu *method* is a string that can be `"rkf45"` (default), or `"rk4"`. In the first case, we use the Runge Kutta-Fehlberg method (with variable step size), in the second case, it is the classic Runge-Kutta method of order 4.
  - As output, the function returns the following matrix (list of lists of real numbers):  

```
{ {tmin,...,tmax}, {y1(tmin),...,y1(tmax)}, {y2(tmin),...,y2(tmax)},...}
```

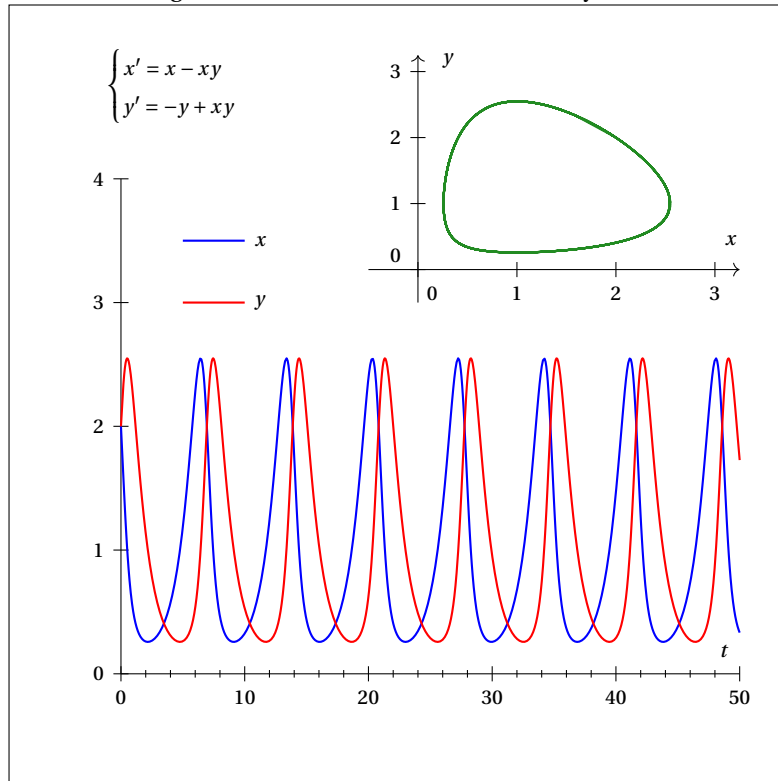
The first component is the list of values of  $t$  (in ascending order), the second is the list of (approximate) values of the component  $y_1$  corresponding to these values of  $t$ , etc.
- The method **g:DplotXY(X, Y [, draw\_options, clip])**, where the arguments  $\langle X \rangle$  and  $\langle Y \rangle$  are two lists of real numbers of the same length, draws the polygonal line consisting of the points  $(X[k], Y[k])$ . The argument  $\langle draw\_options \rangle$  is a string (empty by default) that will be passed as is to the `\draw` instruction. The The argument  $\langle clip \rangle$  is either `nil` (default value) or a table `{x1,x2,y1,y2}`. In the first case, the line is clipped by the current 2D window **after** its transformation by the graph's 2D matrix. In the second case, the line is clipped by the window  $[x_1; x_2] \times [y_1; y_2]$  **before** being transformed by the graph's matrix.

```

\begin{luadraw}{name=lokta_volterra}
local ld = luadraw
local cpx = ld.cpx
local g = ld.graph:new{window={-5,50,-0.5,5},size={10,10,0}, border=true}
local i = cpx.I
local f = function(t,y) return {y[1]-y[1]*y[2],-y[2]+y[1]*y[2]} end
g:Labelsize("footnotesize")
g:Daxes({0,10,1},{limits={0,50},{0,4}}, nsubdiv={4,0}, legendsep={0.1,0},
originpos={"center","center"}, legend={"$t$",""})
local y0 = {2,2}
local M = ld.odesolve(f,0,y0,0,50,250) -- approximate resolution
-- M est une table à 3 éléments: t, x et y
g:Lineoptions("solid","blue",8)
g:Dseg({5+3.5*i,10+3.5*i}); g:Dlabel("$x$",10+3.5*i,{pos="E"})
g:DplotXY(M[1],M[2]) -- points (t,x(t))
g:Linecolor("red"); g:Dseg({5+3*i,10+3*i}); g:Dlabel("$y$",10+3*i,{pos="E"})
g:DplotXY(M[1],M[3]) -- points (t,y(t))
g:Lineoptions(nil,"black",4)
g:Saveattr(); g:Viewport(20,50,3,5) -- change of view
g:Coordsystem(-0.5,3.25,-0.5,3.25) -- new associated landmark
g:Daxes({0,1,1},{legend={"$x$","$y$"},arrows="->"})
g:Lineoptions(nil,"ForestGreen",8); g:DplotXY(M[2],M[3]) -- points (x(t),y(t))
g:Restoreattr() -- back to the old view
g:Dlabel("$\\begin{cases}x'=x-xy\\\\y'=-y+xy\\end{cases}$", 5+4.75*i,{})
g:Show()
\end{luadraw}

```

Figure 6: A Lokta-Volterra differential system



- The method **g:Dodesolve(f, t0, Y0, options)** allows the drawing of a solution to the equation  $Y'(t) = f(t, Y(t))$ .
  - The required argument  $\langle f \rangle$  is a function  $f : (t, Y) \rightarrow f(t, Y)$  with values in  $\mathbf{R}^n$  and where  $Y$  is also in  $\mathbf{R}^n$ :  $Y = \{y_1, y_2, \dots, y_n\}$ , but when  $n = 1$ ,  $Y$  is a real number.
  - The arguments  $\langle t0 \rangle$  and  $\langle Y0 \rangle$  give the initial conditions with  $Y_0 = \{y_1(t_0), \dots, y_n(t_0)\}$  (the  $y_i$  are real), or  $Y_0 = y_1(t_0)$  when  $n = 1$ .
  - The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:
    - \* **t={g:Xinf(), g:Xsup()}**, determines the interval for the variable  $t$ ,

- \* `out={1,2}`, table of two integers  $\{i_1, i_2\}$ . If  $M$  denotes the matrix returned by the `odesolve` function, the points drawn will have the  $M[i_1]$  as abscissas and the  $M[i_2]$  as ordinates. By default, we have  $i_1 = 1$  and  $i_2 = 2$ , which corresponds to the  $y_1$  function as a function of  $t$ ,
- \* `nbdots=50`, determines the number of points to calculate for the function,
- \* `method="rkf45"`, determines the method to use; possible values are `"rkf45"` (default value), or `"rk4"`,
- \* `draw_options=""`, is a string (empty by default) that will be passed as is to the `\draw` instruction,
- \* `clip=nil`, this option is either `nil` (default value) or a table  $\{x_1, x_2, y_1, y_2\}$ . In the first case, the line is clipped by the current 2D window **after** its transformation by the graph's 2D matrix. In the second case, the line is clipped by the window  $[x_1; x_2] \times [y_1; y_2]$  **before** being transformed by the graph's matrix.

### Implicit Curves: Dimplicit

- The function `ld.implicit(f, x1, x2, y1, y2, grid)` calculates and returns a polygonal line constituting the implicit curve with equation  $f(x, y) = 0$  in the box  $[x_1, x_2] \times [y_1, y_2]$ . This box is split according to the argument  $\langle grid \rangle$ .
  - The argument  $\langle f \rangle$  is a function  $f : (x, y) \rightarrow f(x, y)$  with values in  $\mathbf{R}$ .
  - The arguments  $\langle x1 \rangle, \langle x2 \rangle, \langle y1 \rangle, \langle y2 \rangle$  define the plot window, which will be the  $[x_1, x_2] \times [y_1, y_2]$  box. We must have  $x_1 < x_2$  and  $y_1 < y_2$ .
  - The argument  $\langle grid \rangle$  is a table containing two positive integers:  $\{n_1, n_2\}$ . The first integer indicates the number of subdivisions following the  $x$ -axis, and the second the number of subdivisions following the  $y$ -axis.
- The `g:Dimplicit(f, options)` method draws the implicit curve of equations  $f(x, y) = 0$ .
  - The argument  $\langle f \rangle$  is a function  $f : (x, y) \rightarrow f(x, y)$  with values in  $\mathbf{R}$ .
  - The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:
    - \* `view={g:Xinf(), g:Xsup(), g:Yinf(), g:Ysup()}`, table of the form  $\{x_1, x_2, y_1, y_2\}$  which determines the drawing area  $[x_1, x_2] \times [y_1, y_2]$ ,
    - \* `grid={50,50}`, determines the subdivisions,
    - \* `draw_options=""`, string (empty by default) that will be passed as is to the `\draw` instruction.

### Contour Lines: Dcontour

The `g:Dcontour(f, z, options)` method draws **contour lines** of the function  $f : (x, y) \rightarrow f(x, y)$  with real values.

- The argument  $\langle f \rangle$  is the function.
- The argument  $\langle z \rangle$  is the list of different levels to plot.
- The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:
  - `view={g:Xinf(), g:Xsup(), g:Yinf(), g:Ysup()}`, table of the form  $\{x_1, x_2, y_1, y_2\}$  which determines the drawing area  $[x_1, x_2] \times [y_1, y_2]$ ,
  - `grid={50,50}`, determines the subdivisions,
  - `colors={}`, list of colors per level. By default, this list is empty and the current drawing color is used.
  - `draw_options=""`, string (empty by default) that will be passed as is to the `\draw` instruction.

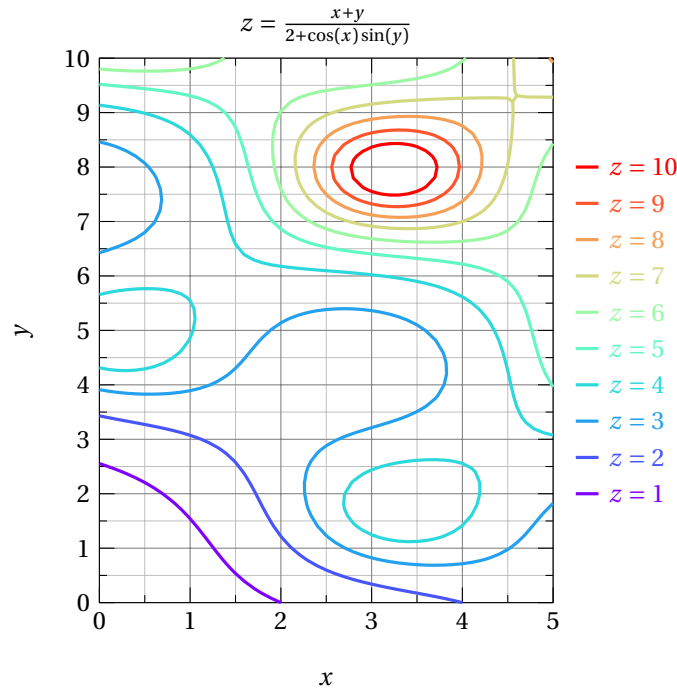
```
\begin{luadraw}{name=Dcontour}
local ld = luadraw
local cpx = ld.cpx
local g = ld.graph:new{window={-1,6.5,-1.5,11},size={10,10,0}}
local i, sin, cos = cpx.I, math.sin, math.cos
local f = function(x,y) return (x+y)/(2+cos(x)*sin(y)) end
local Lz = ld.range(1,10) -- levels to be drawn
local Colors = ld.getpalette(ld.palRainbow,10) -- 10 evenly distributed colors in the palRainbow palette
g:Dgradbox({0,5+10*i,1,1},{legend={"$x$", "$y$"}, grid=true, title="$z=\frac{x+y}{2+\cos(x)\sin(y)}$"})
```

```

g:Linewidth(12); g:Dcontour(f,Lz,{view={0,5,0,10}, colors=Colors})
for k = 1, 10 do
  local y = (2*k+4)/3*i
  g:Dseg({5.25+y,5.5+y},1,"color"..Colors[k])
  g:Labelcolor(Colors[k])
  g:Dlabel("$z="..k.."$",5.5+y,{pos="E"})
end
g:Show()
\end{luadraw}

```

Figure 7: Example with Dcontour



### Parameterization of a Polygonal Line: *curvilinear\_param*

Let  $L$  be a list of complex numbers representing a « continuous » line. It is possible to obtain a parameterization of this line based on a parameter  $t$  between 0 and 1 ( $t$  is the curvilinear abscissa divided by the total length of  $L$ ).

The function `ld.curvilinear_param(L [, close])` returns a function of one variable  $t \in [0; 1]$  and values on the line  $\langle L \rangle$  (list of complex numbers). The value at  $t = 0$  is the first point of  $\langle L \rangle$ , and the value at  $t = 1$  is the last point; this function is followed by a number representing the total length of  $\langle L \rangle$ . The optional argument  $\langle close \rangle$  indicates whether the line  $\langle L \rangle$  should be closed (`false` by default).

```

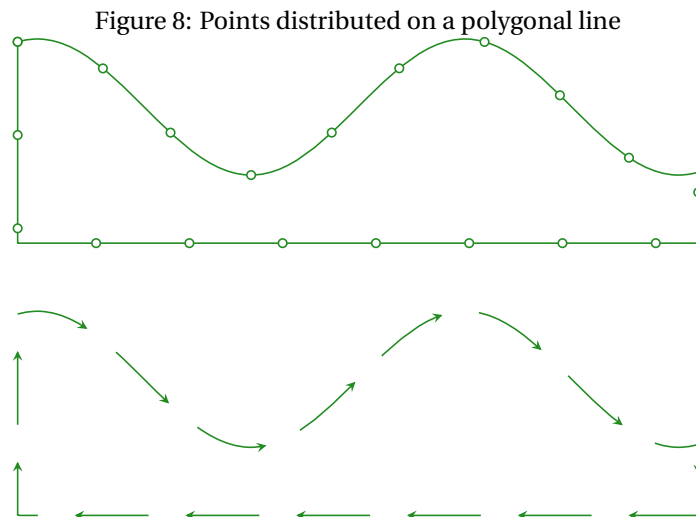
\begin{luadraw}{name=curvilinear_param}
local ld = luadraw
local cpx = ld.cpx
local g = ld.graph:new{bbox=false,size={10,10}}
local i = cpx.I; g:Linewidth(8)
local L = ld.cartesian(math.sin,-5,5)[1]
ld.insert(L, {5-2*i, -5-2*i})
local f = ld.curvilinear_param(L, true)
local I = ld.map(f,ld.linspace(0,1,20)) -- 20 points distributed over L
g:Shift(4*i)
g:Lineoptions(nil,"ForestGreen",6); g:Dpolyline(L,true)
g:Filloptions("full","white"); g:Ddots(I) -- The first and last points coincide because L is closed
-- Another example of use:
local nb = 16 -- number of arrows
local t = ld.linspace(0,1,3*nb+1)
g:Shift(-4*i)
for k = 0,nb-1 do
  g:Dparametric(f,{t={t[3*k+1],t[3*k+3]}},nbdots=10,nbdiv=2,draw_options="-stealth")
end
\end{luadraw}

```

```

end
g:Show()
\end{luadraw}

```



## 5) Domains related to Cartesian curves

### Ddomain1

- The function **ld.domain1(f, a, b [, nbdots, discont, nbdiv])** returns a list of complex numbers that represents the contour of the part of the plane bounded by the curve of the function  $\langle f \rangle$  on an interval determined by  $\langle a \rangle$  and  $\langle b \rangle$ , the  $x$ -axis, and the lines  $x = a$ ,  $x = b$ .
- The method **g:Ddomain1(f, options)** draws this contour. The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:
  - **x= {g:Xinf(),g:Xsup()}**, this is the interval for the parameter  $t$  (by default, it's the entire range of the window's  $x$ -coordinates),
  - **nbdots=40**,
  - **discont=false**,
  - **nbdiv=5**,
  - **draw\_options=""**, string which will be passed as is to the instruction `\draw`.

### Ddomain2

- The **ld.domain2(f, g, a, b [, nbdots, discont, nbdiv])** function returns a list of complex numbers that represents the contour of the part of the plane bounded by the curve of the function  $\langle f \rangle$ , the curve of the function  $\langle g \rangle$ , and the lines  $x = a$ ,  $x = b$ .
- The **g:Ddomain2(f, g, options)** method draws this contour. The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:
  - **x= {g:Xinf(),g:Xsup()}**, this is the interval for the parameter  $t$  (by default, it's the entire range of the window's  $x$ -coordinates),
  - **nbdots=40**,
  - **discont=false**,
  - **nbdiv=5**,
  - **draw\_options=""**, string which will be passed as is to the instruction `\draw`.



**Ddomain3**

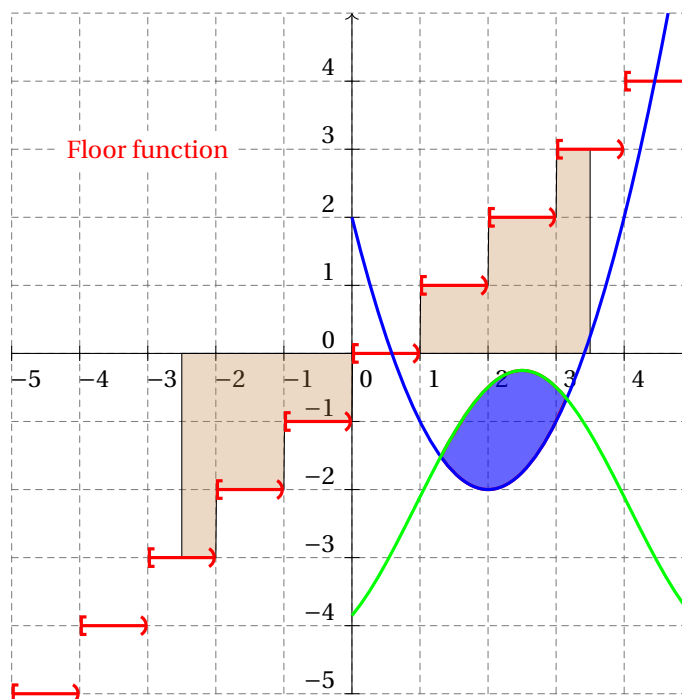
- The **ld.domain3(f, g, a, b [, nbdots, discont, nbdiv])** function returns a list of complex numbers that represents the contour of the part of the plane bounded by the curve of the function  $\langle f \rangle$  and that of the function  $\langle g \rangle$ , searching for intersection points in the interval determined by  $\langle a \rangle$  and  $\langle b \rangle$ .
- The **g:Ddomain3(f, g, options)** method draws this contour. The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:
  - **x= {g:Xinf(), g:Xsup()}**, this is the interval for the parameter  $t$  (by default, it's the entire range of the window's x-coordinates),
  - **nbdots=40**,
  - **discont=false**,
  - **nbdiv=5**,
  - **draw\_options=""**, string which will be passed as is to the instruction `\draw`.

```

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

```

Figure 9: Floor function, Ddomain1 and Ddomain3 functions

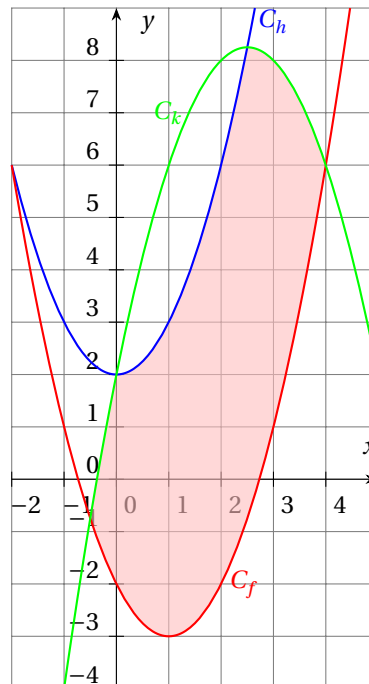


## Dinequalities

- The function `ld.inequalities(constraints, x1, x2, y1, y2)` returns a list of complex numbers representing the boundary of the portion of the plane located within the cuboid  $[x_1; x_2] \times [y_1; y_2]$  and satisfying the constraints expressed in the argument  $\langle constraints \rangle$ . This argument is a list of the form  $\{f_1, sg_1, f_2, sg_2, \dots, f_n, sg_n\}$  where the  $\langle f_i \rangle$  are functions ( $f_i: x \mapsto f_i(x) \in \mathbf{R}$ ), and the  $\langle sg_i \rangle$  are either ">" or "<". The first constraint is given by  $\langle f_1 \rangle$  and  $\langle sg_1 \rangle$ , and this constraint is either  $y > f_1(x)$ , or  $y < f_1(x)$  depending on the value of  $\langle sg_1 \rangle$ . The same applies to the following constraints.
- The `g:Dinequalities(constraints, options)` method allows you to draw this contour. The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:
  - `view={g:Xinf(),g:Xsup(),g:Yinf(),g:Ysup()}`, this is the window in which the resolution will take place. By default, it is the current 2D window.
  - `draw_options=""`, a string that will be passed as is to the `\draw` instruction.
  - `useclip=false`, with the value `false` the method uses the previous function (which calculates the contour of the solution), with the value `true` the method does not calculate the contour but uses clips (one per constraint).

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

Figure 10: Dinequalities



## Dimplicit\_inequalities

- The function `ld.implicit_inequality(f, sg, x1, x2, y1, y2, grid)` returns a **path** representing the boundary of the portion of the plane located within the tile  $[x_1; x_2] \times [y_1; y_2]$  and satisfying the condition  $f(x, y) \geq 0$  if  $\langle sg \rangle = ">"$ , or  $f(x, y) \leq 0$  if  $\langle sg \rangle = "<"$ .

if  $\langle sg \rangle = "<".$  The argument  $\langle f \rangle$  is a function of two variables  $\langle f \rangle: (x, y) \mapsto f(x, y) \in \mathbf{R}$ . The argument  $\langle grid \rangle$  is a table of two integers:  $\langle grid \rangle = \{\langle n1 \rangle, \langle n2 \rangle\}$ , the first indicates the number of subdivisions of the interval  $[x_1; x_2]$ , and the second, the number of subdivisions of the interval  $[y_1; y_2]$ .

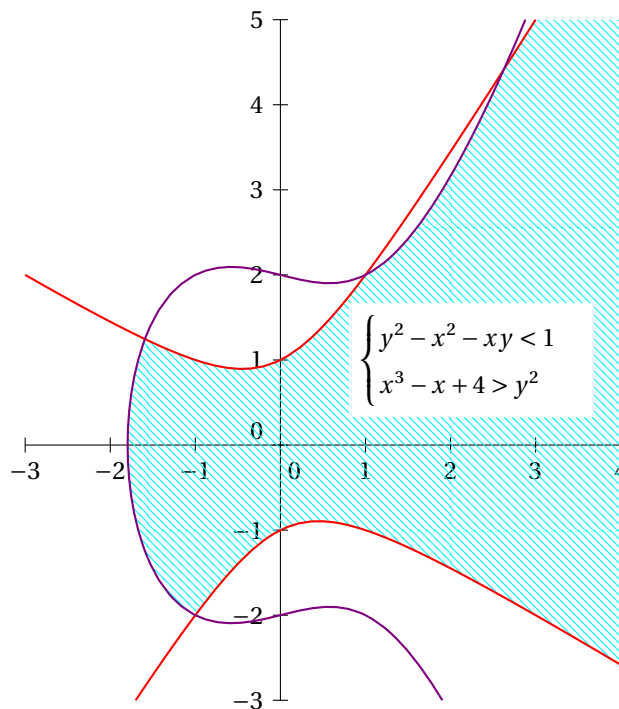
- The method **g:Dimplicit\_inequalities(constraints, options)** allows you to fill in a portion of the design that satisfies the constraints expressed in the argument  $\langle constraints \rangle$ . This argument is a list of the form  $\{f1, sg1, f2, sg2, \dots, fn, sgn\}$  where the  $\langle fi \rangle$  are functions ( $f_i: (x, y) \mapsto f_i(x, y) \in \mathbf{R}$ ), and the  $\langle sgi \rangle$  are either ">" or "<". The first constraint is given by  $\langle f1 \rangle$  and  $\langle sg1 \rangle$ , and this constraint is either  $f_1(x, y) > 0$  or  $f_1(x, y) < 0$  depending on the value of  $\langle sg1 \rangle$ . The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are, with their default values:

- **view**={g:Xinf(),g:Xsup(),g:Yinf(),g:Ysup()}, this is the window in which the resolution will take place. By default, it is the current 2D window.
- **draw\_options**="", a string that will be passed as is to the `\draw` instruction.
- **grid**={50,50}: a table of two integers, the first indicating the number of subdivisions of the interval of  $x$ , and the second, the number of subdivisions of the interval of  $y$ .

**NB:** This method does not calculate the contour but uses clipping (one per constraint).

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

Figure 11: The **g:Dimplicit\_inequalities** method



## 6) Points (Ddots) and Labels (Dlabel)

- The method for drawing one or more points is: **g:Ddots(dots [, mark\_options])**.
  - The argument  $\langle dots \rangle$  can be either a single point (i.e., a complex number), a list (a table) of complex numbers, or a list of lists of complex numbers. The points are drawn in the current color of the line plot.
  - The  $\langle mark\_options \rangle$  argument is a string (empty by default) that will be passed as is to the `\draw` instruction (local modifications), for example:
 

```
mark_options = "color=green, line width=1.2, scale=0.25"
```
  - Two methods to globally modify the appearance of points:
    - \* The **g:Dotstyle(style)** method defines the point style. The  $\langle style \rangle$  argument is a string that defaults to `"*"`. The possible styles are those of the `plotmarks` library.
    - \* The **g:Dotscale(scale)** method allows you to adjust the dot size. The  $\langle scale \rangle$  argument is a positive integer that defaults to 1. It is used to multiply the default dot size. The current line width also affects the dot size. For "solid" dot styles (e.g., the `"triangle*"` style), the current fill style and color are used by the library.
- The method for placing a label is:

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

- The arguments  $\langle text1 \rangle$ ,  $\langle text2 \rangle$ , ..., are strings; they are the labels.
- The arguments  $\langle anchor1 \rangle$ ,  $\langle anchor2 \rangle$ , ..., are complex numbers representing the anchor points of the labels.
- The arguments  $\langle options1 \rangle$ ,  $\langle options2 \rangle$ , ..., allow you to locally define the label parameters; they are tables whose fields are the possible options. These are, with their default values:
  - \* **pos="center"**, indicates the position of the label relative to the anchor point. `"center"` (centered), `"N"` (north), `"NE"` (northeast), `"E"` (east), `"SE"` (southeast), `"S"` (south), `"SW"` (southwest), `"W"` (west), `"NW"` (northwest). By default, it is set to `"center"`, and in this case the label is centered on the anchor point,
  - \* **dist=0**, distance in cm between the label and its anchor point when **pos** is not equal to `"center"`,
  - \* **dir= {}**, this is a table of the form `{dirX [,dirY,dep]}` which indicates the writing direction. The three values **dirX**, **dirY**, and **dep** are three complex numbers representing three vectors. The first two indicate the writing direction, and the third indicates a translation of the label relative to the anchor point. The vector **dep** is zero by default, and the vector **dirY**, if absent, is equal to the vector **dirX** rotated 90 degrees in the clockwise direction. By default, the **dir** option is equal to the current value of the writing direction,
  - \* **node\_options=""**, string (empty by default) intended to receive options that will be passed directly to TikZ in the `node[]` instruction.
  - \* The labels are drawn in the current color of the document text, but the color can be changed with the **node\_options** argument, for example, by setting: **node\_options="color=blue"**.
- Warning:** The options chosen for a label also apply to subsequent labels if they are unchanged.

Global options for labels:

- The **g:Labelstyle(position)** method allows you to specify the position of the labels relative to the anchor points. The  $\langle position \rangle$  argument is a string that can be: `"N"` for north, `"NE"` for northeast, `"NW"` for northwest, or `"S"`, `"SE"`, `"SW"`. By default, it is set to `"center"`, and in this case the label is centered on the anchor point.
- The **g:Labelcolor(color)** method allows you to set the color of the labels. The  $\langle color \rangle$  argument is a string representing a color for TikZ. By default, the argument is an empty string, which represents the current color of the document.
- The **g:Labelangle(angle)** method allows you to specify an  $\langle angle \rangle$  (in degrees) for rotating the labels around the anchor point. This angle is zero by default.
- The **g:Labelsize(size)** method allows you to manage the size of the labels. The  $\langle size \rangle$  argument is a string that can be: `"tiny"`, or `"scriptsize"`, or `"footnotesize"`, etc. By default, the argument is an empty string, which represents the `"normalsize"` size.

- The **g:Labeldir(dir)** method allows you to manage the writing direction. The argument  $\langle dir \rangle$  is an table of the form  $\{\text{dirX } [, \text{dirY}, \text{dep}]\}$  which indicates the writing direction. The three values **dirX**, **dirY**, and **dep** are three complex numbers representing three vectors. The first two indicate the writing direction, and the third indicates a translation of the label relative to the anchor point. The vector **dep** is zero by default, and the vector **dirY**, if absent, is equal to the vector **dirX** rotated 90 degrees in the clockwise direction. When  $\langle dir \rangle$  is an empty list, this represents the usual writing direction.
- The **g:Dlabeldot(text, anchor, options)** method allows you to place a label and draw the anchor point at the same time.
  - The  $\langle text \rangle$  argument is a string; it is the label.
  - The  $\langle anchor \rangle$  argument is a complex representing the label's anchor point.
  - The argument  $\langle options \rangle$  is a table whose fields are the possible options. These are identical to those of the **Dlabel** method, plus the `mark_options=""` option, which is a string that will be passed as is to the `\draw` instruction when drawing the anchor point.

## 7) Paths: Dpath, Dspline, and Dtcurve

### What is a path

A path is a table of complex numbers and instructions (in the form of strings). This table represents a succession of different "pieces", each piece being a sequence of data (2D points and sometimes numeric values) and ending with a string of characters that represents an instruction. The path is governed by the following rule:

**the last point of one piece is the first point of the next piece (it is therefore not repeated)**

### Example:

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

The path *L* is composed of five pieces, which are:

1.  $\{Z(-3,2), "m"\}$ : there is a data element and the instruction "m" which means *moveto*. This instruction does not actually draw the path, but it allows a new component to begin, starting at the first point  $Z(-3,2)$  (the last point of the previous piece, if there is one, is not taken into account by this instruction; this is the only exception).
2.  $\{Z(-3,2), -3, -2, "l"\}$ : the first point of the second piece is indeed  $Z(-3,2)$  and not  $-3$ , because  $Z(-3,2)$  is the last point of the previous piece. Therefore, there are three data points, and the instruction "l" which means *lineto*, it is like executing the instruction `g:Dpolyline(Z(-3,2), -3, -2)`, so these three points are connected by a segment. The last point of this piece is  $-2$ .
3.  $\{-2, 0, 2, 2, -1, "ca"\}$ : the first point of the third piece is indeed  $-2$  and not  $0$ , because  $-2$  is the last point of the previous piece. There are five data points, and the instruction "ca" stands for *circle arc*. It's as if we were executing the instruction `g:Darc(-2, 0, 2, 2, -1)`, so the center is  $0$ , the arc goes from  $-2$  to  $2$  with a radius equal to  $2$  and in a clockwise direction (last value  $-1$ ). The last point of this piece is  $2$ .
4.  $\{2, 3, Z(3,3), 0.5, "la"\}$ : the first point of the fourth piece is  $2$  (not  $3$ ). There are four data points, and the instruction "la" stands for *line arc*. This is a polygonal line with rounded corners and a circular arc of radius  $0.5$  (the value preceding the instruction). The points of this line are  $\{2, 3, Z(3,3)\}$ , so there will be a rounding to  $3$ . The last point of this piece is  $Z(3,3)$ .
5.  $\{Z(3,3), 1, Z(-1,5), Z(-3,2), "b"\}$ : the first point of the fifth piece is  $Z(3,3)$  (not  $1$ ). The instruction "b" stands for *bezier*, so we draw a Bézier curve from  $Z(3,3)$  to  $Z(-3,2)$ . The other two points,  $1$  and  $Z(-1,5)$ , are the first and second control points of the curve.

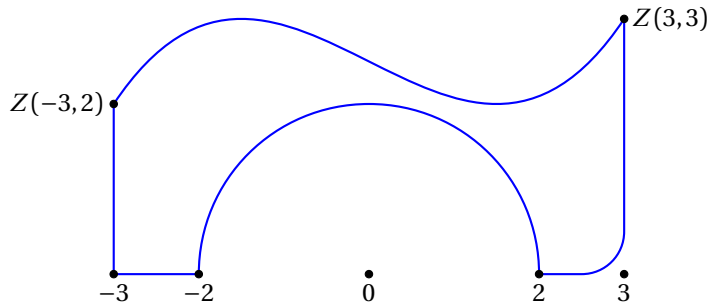
Here is what this path gives:

```

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

```

Figure 12: Path example



**Note:** In the example above, you can replace the part:  $Z(-3,2)$ , "m", -3, -2, "l", with:  $Z(-3,2)$ , -3, -2, "l" because there is no other part before the *moveto*.

**Available commands and their syntax**, the word *last* denotes the last point of the previous piece:

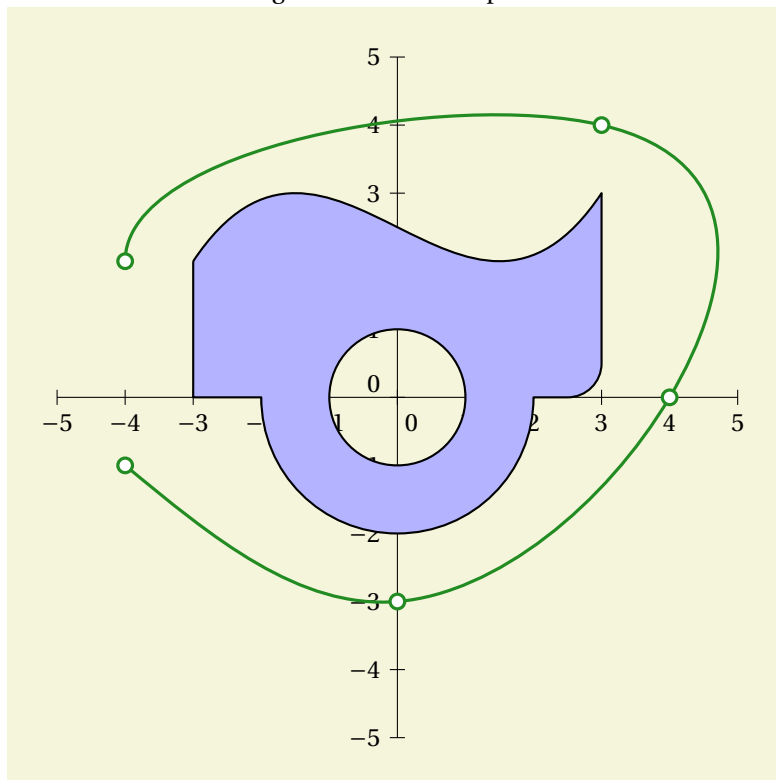
- $z_1$ , "m" (moveto) starts a new component of the path at the point with affix  $z_1$ .
- $z_1, \dots, z_n$ , "l" (lineto) draws the polyline  $\{last, z_1, \dots, z_n\}$ .
- $c_1, c_2, z_2$ , "b" (bézier) draws the Bézier curve  $\{last, c_1, c_2, z_2\}$ , where  $c_1$  and  $c_2$  are the two control points.
- $z_1, \dots, z_n$ , "s" (spline) draws the natural cubic spline through the points  $\{last, z_1, \dots, z_n\}$ .
- $z_1$ , "c" (circle) draws the circle with center  $z_1$  passing through the point *last*. There is another possible syntax for the circle:  $z_1, z_2$ , "c", in which case the circle passing through the points *last*,  $z_1$ , and  $z_2$  is drawn.
- $z_1, z_2, r, sens$ , "ca" (circular arc) draws a circular arc with center  $z_1$  and radius  $r$ , going from *last* to  $z_2$ , counterclockwise when  $sens=1$  (and therefore clockwise when  $sens=-1$ ).
- $z_1, z_2, rx, ry, sens, inclinaison$ , "ea" (elliptic arc) draws an elliptic arc with center  $z_1$  going from *last* to  $z_2$ ;  $rx$  and  $ry$  are the two radii along the two axes of the ellipse; *inclinaison* is the angle in degrees between the first axis of the ellipse (the one corresponding to  $rx$ ) and the horizontal. The parameter *inclinaison* is optional and defaults to 0.
- $z_1, rx, ry, inclinaison$ , "e" (ellipse) draws the ellipse with center  $z_1$  passing through *last*;  $rx$  and  $ry$  are the two radii along the two axes of the ellipse; *inclinaison* is the angle in degrees between the first axis of the ellipse (the one corresponding to  $rx$ ) and the horizontal. The parameter *inclinaison* is optional and defaults to 0. When the point *last* is not on this ellipse, a segment is drawn between this point and a point on the ellipse.
- $z_1, \dots, z_n, r$ , "la" (line arc) draws the polyline  $\{last, z_1, \dots, z_n\}$  while replacing each "corner" by a circular arc of radius  $r$ .
- $z_1, \dots, z_n, r$ , "cla" (closed line arc) same as the previous command, except that the polyline is closed.
- "cl" (closepath) this command is used alone; it closes the current component by drawing a segment joining the last point to the first point of the current component.

## Draw a Path

- The function **ld.path(path [, nbdots])** returns a polygonal line containing the points that make up the  $\langle path \rangle$ . The optional argument,  $\langle nbdots \rangle$ , is the minimum number of points calculated for each Bézier curve; its default value is the global variable `ld.bezier_nbdots`, which is initialized to 12.
- The **g:Dpath(path [, draw\_options])** method draws the  $\langle path \rangle$  (which was described above) using Bézier curves as much as possible, including arcs, ellipses, etc. The  $\langle draw\_options \rangle$  argument is a string that will be passed directly to the `\draw` instruction.
- The function **ld.spline(points, v1, v2)** returns a path (to be drawn with **Dpath**) of the cubic spline passing through the points of the argument  $\langle points \rangle$  (which must be a list of complex numbers). The arguments  $\langle v1 \rangle$  and  $\langle v2 \rangle$  are tangent vectors imposed at the ends (constraints); when these are equal to `nil` (default value), a natural cubic spline (i.e., unconstrained) is calculated.
- The method **g:Dspline(points [, v1, v2, draw\_options])** draws the spline described above. The argument  $\langle draw\_options \rangle$  is a string that will be passed directly to the instruction `\draw`.

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

Figure 13: Path and Spline



- The function **ld.tcurve(L)** returns a curve passing through given points as a path, with tangent vectors (left and right) imposed at each point. The argument  $\langle L \rangle$  is a table of the form:

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

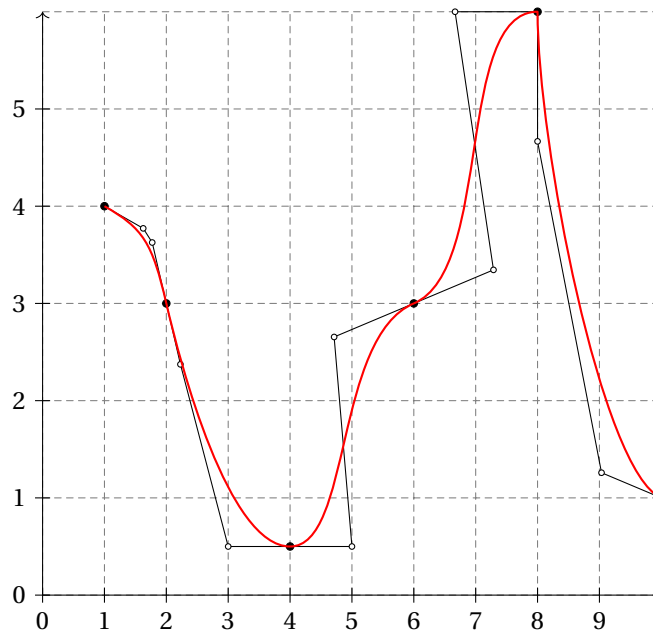
$\langle point1 \rangle, \dots, \langle pointN \rangle$  are the interpolation points of the curve (complex numbers), and each of them is followed by a table of the form  $\{t1, a1, t2, a2\}$  which specifies the tangent vectors to the curve to the left of the point (with  $\langle t1 \rangle$  and  $\langle a1 \rangle$ ) and to the right of the point (with  $\langle t2 \rangle$  and  $\langle a2 \rangle$ ). The left tangent vector is given by the formula  $V_g = t_1 \times e^{ia_1\pi/180}$ , so  $\langle t1 \rangle$  represents the modulus and  $\langle a1 \rangle$  is an argument **in degrees** of this vector. The same is true for  $\langle t2 \rangle$  and  $\langle a2 \rangle$  for the right tangent vector, **but these are optional**, and if not specified, they take the same values as  $\langle t1 \rangle$  and  $\langle a1 \rangle$ .

Two consecutive points will be connected by a Bézier curve; the function calculates the control points to obtain the desired tangent vectors.

- The method **g:Dtcurve(L, options)** draws the path obtained by  $\langle tcurve \rangle$  described above. The argument *options* is a table whose fields are the possible options. These are, with their default values:
  - **showdots=false**, this option allows you to draw or not the given interpolation points as well as the calculated control points, allowing for visualization of the constraints.
  - **draw\_options=""**, this is a string that will be passed directly to the `\draw` instruction.

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

Figure 14: Interpolation curve with imposed tangent vectors



## 8) Paths and Clipping: Beginclip() and Endclip()

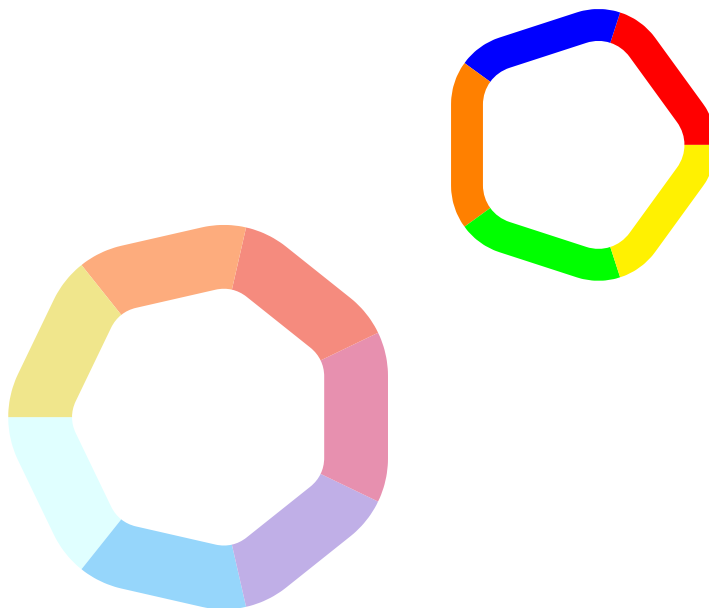
A path can be used for clipping using two functions: **g:Beginclip(path [, reverse])** and **g:Endclip()**. The first opens a *scope* group and passes the  $\langle path \rangle$  as an argument to TikZ's `\clip` function. The second closes the *scope* group; it is essential



(otherwise there will be a compilation error). The `<reverse>` argument is a Boolean that defaults to `false`. When it has the value `true`, the clipping is reversed, meaning that only what is outside the `<path>` will be drawn, but for this to happen, the path must be counterclockwise.

```
\begin{luadraw}{name=polygon_with_different_line_color_and_rounded_corners}
local ld = luadraw
local g = ld.graph:new{window={-5,5,-5,5},size={10,10}}
local i = ld.cpx.I
local Dcolored_polyreg = function(c,a,nb,r,wd,colors)
-- c=center, a=vertice, nb=number of sides, r=radius, wd=width in point, colors=list of colors
  local L = ld.polyreg(c,a,nb)
  ld.insert(L,{r,"cla"}) --polygon width rounded corners (radius=r)
  local angle = 360/nb
  local b = a
  for k = 1, nb do
    a = b; b = ld.rotate(a,angle,c)
    g:Beginclip({2*a-c,c,2*b-c,"1"}) -- definition of an angular sector for clipping
    g:Dpath(L,"line width"..wd.."pt,color"..colors[k])
    g:Endclip()
  end
end
end
Dcolored_polyreg(3+2*i,5+2*i,5,0.8,12,{"red","blue","orange","green","yellow"}) -- pentagon
Dcolored_polyreg(-2.5-2*i,-5-2*i,7,1,24,ld.getpalette(ld.palGasFlame,7)) -- heptagon
g:Show()
\end{luadraw}
```

Figure 15: Clipping Example



## 9) Axes and Grids

Global variables used for axes and grids:

- `ld.maxGrad = 100`: Maximum number of tick marks on an axis.
- `ld.defaultlabelshift = 0.1875`: When a grid is drawn with the axes (option `grid=true`), the labels are automatically shifted along the axis using this variable.
- `ld.defaulttxylabsep = 0`: Sets the default distance between labels and tick marks.
- `ld.defaultlegendsep = 0.2`: Sets the default distance between the legend and the axis.

- `ld.digits = 4`: Default number of decimal places in string conversions; terminal 0s are removed.
- `ld.dollar = true`: to add dollars around the tick labels.
- `ld.siunitx = false`: with the value `true`, the labels are formatted with the macro `\num{. .}` of the *siunitx* package, which allows you to use certain options of this package, such as replacing the decimal point with a comma by doing:

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

or by doing:

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

For axes, in both 2D and 3D, all labels are formatted as strings with the `ld.num(x)` function. This transforms the number  $x$  into a string *str* with the number of decimal places set by the global variable `ld.digits`. When the `ld.siunitx` variable has the value `true`, the function returns `"\num{str}"`, otherwise it simply returns *str*. This also applies to 3D axes. Here is the code for this function:

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

## Daxes

The axes are plotted using the method `g:Daxes([A, xstep, ystep], options)`.

- The first argument specifies the intersection point of the two axes (this is the complex  $\langle A \rangle$ ), the graduation spacing on the  $x$ -axis (this is  $\langle xstep \rangle$ ), and the graduation spacing on  $y$ -axis (this is  $\langle ystep \rangle$ ). By default, the point  $\langle A \rangle$  is the origin  $Z(0,0)$ , and both steps are equal to 1.
- The argument  $\langle options \rangle$  is a table whose fields are the possible options. Here are these options with their default values:
  - `showaxe=1,1`. This option specifies whether or not the axes should be plotted (1 or 0). The first value is for the  $x$ -axis and the second for the  $y$ -axis.
  - `arrows="--"`. This option allows you to add an arrow to the axes (no arrow by default; enter `"->"` for example to add an arrow).
  - `limits="auto","auto"`. This option specifies the extent of the two axes (first value for  $x$ -axis, second value for  $y$ -axis). The value `"auto"` means that it is the entire line, but you can specify the extreme abscissas, for example: `limits=-4,4,"auto"`.
  - `gradlimits={"auto","auto"}`. This option allows you to specify the range of the graduations on both axes (first value for  $x$ -axis, second value for  $y$ -axis). The value `"auto"` means that it is the entire line, but you can specify the extreme graduations, for example: `gradlimits={{-4.4},{-2.3}}`.
  - `unit={"",""}`. This option allows you to specify the range of the graduations on the axes. The default value means that the step value should be taken ( $\langle xstep \rangle$  on  $x$ -axis, or  $\langle ystep \rangle$  on  $y$ -axis), EXCEPT when the option `labeltext` is not the empty string, in which case `unit` takes the value 1.
  - `nbsubdiv={0,0}`. This option specifies the number of subdivisions between two main ticks on the axis.
  - `tickpos={0.5,0.5}`. This option specifies the position of the ticks relative to each axis. These are two numbers between 0 and 1. The default value of 0.5 means they are centered on the axis. (0 and 1 represent the ends).
  - `tickdir={"auto","auto"}`. This option specifies the direction of the ticks on the axis. This direction is a non-zero vector (complex number). The default value `"auto"` means the ticks are orthogonal to the axis.
  - `xyticks={0.2,0.2}`. This option specifies the length of the ticks on the axis.
  - `xylabelsep={0,0}`. This option specifies the distance between the labels and the tick marks on the axis.

- `originpos={"right","top"}`. This option specifies the position of the label at the origin on the axis. Possible values are: `"none"`, `"center"`, `"left"`, `"right"` for  $x$ -axis, and `"none"`, `"center"`, `"bottom"`, `"top"` for  $y$ -axis.
- `originnum={A.re,A.im}`. This option specifies the value at the origin point of the graduations (graduation number 0).

The formula that defines the label at tick mark number  $n$  is:

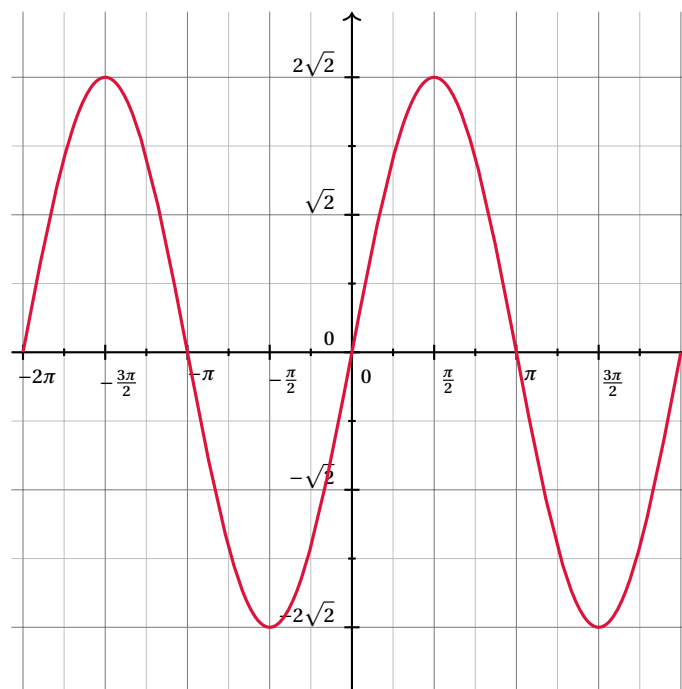
$$(\text{originnum} + \text{unit} \cdot n) \text{labeltext} / \text{labelden}.$$

- `originloc=A`. This option specifies the origin point of the graduations.
- `legend={"", ""}`. This option allows you to specify a legend for the axis.
- `legendpos={0.975,0.975}`. This option specifies the position (between 0 and 1) of the legend relative to each axis.
- `legendsep={ld.defaultlegendsep,ld.defaultlegendsep}`. This option specifies the distance between the legend and the axis. The legend is on the other side of the axis from the graduations.
- `legendangle={"auto","auto"}`. This option specifies the angle (in degrees) that the legend should make for the axis. The default value `"auto"` means that the legend must be parallel to the axis if the `labelstyle` option is also set to `"auto"`, otherwise the legend is horizontal.
- `legendstyle={"auto","auto"}`. Specifies the label style for the legends; with the value `"auto"`, it is determined automatically; otherwise, the following values can be used: `"N"`, `"NW"`, `"W"`, `"SW"`, `"S"`, `"SE"`, `"E"`, `"NE"`.
- `labelpos={"bottom","left"}`. This option specifies the position of the labels relative to the axis. For the  $x$ -axis, the possible values are: `"none"`, `"bottom"` ou `"top"`, for the  $y$ -axis it is: `"none"`, `"right"` ou `"left"`.
- `labelden={1,1}`. This option specifies the denominator of the labels (integer) for the axis. Remember that the formula that defines the label at graduation number  $n$  is:
 
$$(\text{originnum} + \text{unit} \cdot n) \text{labeltext} / \text{labelden}.$$
- `labeltext={"", ""}`. This option defines the text that will be added to the numerator of the labels for the axis.
- `labelstyle={"S","W"}`. This option sets the label style for each axis. Possible values are `"auto"`, `"N"`, `"NW"`, `"W"`, `"SW"`, `"S"`, `"SE"`, `"E"`, `"NE"`.
- `labelangle={0,0}`. This option sets the angle of the labels in degrees from the horizontal for each axis.
- `labelcolor={"", ""}`. This option allows you to choose a color for the labels on each axis. The empty string represents the default color.
- `labelshift={0,0}`. This option allows you to define a systematic offset for the labels on the axis (offset along the axis). With the option `grid=false` the default values are 0 and 0, but with the option `grid=true` the default is `labelshift={ld.defaultlabelshift,ld.defaultlabelshift}` where `ld.defaultlabelshift` is a global variable initialized to 0.1875.
- `xynode_options=""`. String of characters that will be passed as is to the instruction `\node{}` for all labels on both axes (but not for legends).
- `xnode_options=xynode_options`. String of characters that will be passed as is to the instruction `\node{}` for all labels on the  $x$  axis (but not for the legend).
- `ynode_options=xynode_options`. String of characters that will be passed as is to the instruction `\node{}` for all labels on the  $y$  axis (but not for the legend).
- `nbdeci={2,2}`. This option specifies the number of decimal places for numeric values on the axis.
- `use_siunitx={ld.siunitx,ld.siunitx}`. This option specifies whether numeric values should be formatted using the `siunitx` package; the default value is that of the global variable `ld.siunitx`, which is `false` by default.
- `myxlabels=""`. This option allows you to impose custom labels on the  $x$ -axis. When any are present, the value passed to the option must be a list of the type: `{pos1,"text1",pos2,"text2",...}`. The number `<pos1>` represents an abscissa in the  $(A,xstep)$  coordinate system, which corresponds to the affix point  $A+pos1 \cdot xstep$ .
- `myylabels=""`. This option allows you to impose custom labels on the  $y$ -axis. When any are present, the value passed to the option must be a list of the type: `{pos1,"text1",pos2,"text2",...}`. The number `<pos1>` represents an abscissa in the coordinate system  $(A,iystep)$ , which corresponds to the affix point  $A+pos1 \cdot ystep \cdot i$ .

- `grid=false`. This option allows you to add a grid or not.
- `showlines={true,true}`. When `grid=true`, this option allows you to display or hide the vertical lines of the grid (corresponding to the  $x$  axis) as well as the horizontal lines of the grid (corresponding to the  $y$  axis).
- `drawbox=false`. This option draws the axes as a box; in this case, the graduations are on the left and bottom sides.
- `gridstyle="solid"`. This option sets the line style for the primary grid.
- `subgridstyle="solid"`. This option sets the line style for the secondary grid. A secondary grid appears when there are subdivisions on one of the axes.
- `gridcolor="gray"`. This sets the color of the primary grid.
- `subgridcolor="lightgray"`. This sets the color of the secondary grid.
- `gridwidth=4`. Line thickness of the primary grid (which is 0.4pt).
- `subgridwidth=2`. Line thickness of the secondary grid (which is 0.2pt).

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

Figure 16: Example with axes with grid



### DaxeX and DaxeY

The methods `g:DaxeX([A, xstep], options)` and `g:DaxeY([A, ystep], options)` allow you to plot the axes separately.

- The first argument specifies the point serving as the origin (the complex number  $\langle A \rangle$ ) and the step size of the tick marks on the axis. By default, the point  $\langle A \rangle$  is the origin  $Z(0,0)$ , and the step size is equal to 1.
- The argument  $\langle options \rangle$  is a table specifying the possible options. Here are these options with their default values:

- `showaxe=1`. This option specifies whether or not the axis should be plotted (1 or 0).
- `arrows="-"`. This option allows you to add an arrow to the axis (no arrow by default; enter `"->"` to add an arrow).
- `limits="auto"`. This option allows you to specify the range of the two axes. The value "auto" means that it is the entire line, but you can specify the extreme abscissas, for example: `limits={-4.4}`.
- `gradlimits="auto"`. This option allows you to specify the range of the graduations on both axes. The value "auto" means that it is the entire line, but you can specify the extreme graduations, for example: `gradlimits={-2.3}`.
- `unit=""`. This option allows you to specify the range of the graduations on the axis. The default value ("" ) means to take the step value, EXCEPT when the `labeltext` option is not the empty string, in which case `unit` takes the value 1.
- `nbsubdiv=0`. This option specifies the number of subdivisions between two main tick marks.
- `tickpos=0.5`. This option specifies the position of the tick marks relative to the axis. These are two numbers between 0 and 1. The default value of 0.5 means they are centered on the axis. (0 and 1 represent the ends).
- `tickdir="auto"`. This option indicates the direction of the tick marks on the axis. This direction is a non-zero (complex) vector. The default value "auto" means the tick marks are orthogonal to the axis.
- `xyticks=0.2`. This option specifies the length of the tick marks.
- `xylabelsep=0`. This option specifies the distance between the labels and the tick marks.
- `originpos="center"`. This option specifies the position of the label at the origin on the axis. Possible values are: "none", "center", "left", "right" for *x*-axis, and "none", "center", "bottom", "top" for *y*-axis.
- `originnum=A.re` for *x*-axis and `originnum=A.im` for *y*-axis. This option specifies the value of the tick mark at the origin (tick mark number 0).

The formula that defines the label at tick mark number *n* is:

$$(\text{originnum} + \text{unit} * n) \text{ "labeltext" } / \text{labelden}.$$

- `legend=""`. This option allows you to specify a legend for the axis.
- `legendpos=0.975`. This option specifies the position (between 0 and 1) of the legend relative to the axis.
- `legendsep=1d.defaultlegendsep`. This option specifies the distance between the legend and the axis. The legend is on the other side of the axis from the graduations.
- `legendangle="auto"`. This option specifies the angle (in degrees) that the legend should make for the axis. The default value "auto" means that the legend must be parallel to the axis if the `labelstyle` option is also set to "auto", otherwise the legend is horizontal.
- `legendstyle="auto"`. Specifies the label style for the legend. With the value "auto", the style is determined automatically; otherwise, the following values can be used: "N", "NW", "W", "SW", "S", "SE", "E", "NE".
- `labelpos="bottom"` for *x*-axis and `labelpos="left"` for *y*-axis. This option specifies the position of the labels relative to the axis. For the *x*-axis, the possible values are: "none", "bottom" ou "top", for the *y*-axis it is: "none", "right" ou "left".
- `labelden=1`. This option specifies the denominator of the labels (integer) for the axis. The formula that defines the label at graduation number *n* is:
 
$$(\text{originnum} + \text{unit} * n) \text{ "labeltext" } / \text{labelden}.$$
- `labeltext=""`. This option defines the text that will be added to the numerator of the labels.
- `labelstyle="S"` for *x*-axis and `labelstyle="W"` for *y*-axis. This option defines the style of the labels. The possible values are: "auto", "N", "NW", "W", "SW", "S", "SE", "E", "NE".
- `labelangle=0`. This option sets the angle of the labels in degrees from the horizontal.
- `labelcolor=""`. This option allows you to choose a color for the labels. The empty string represents the current text color.
- `labelshift=0`. This option allows you to set a systematic offset for labels on the axis (along-axis offset).

- `nbdeci=2`. This option specifies the number of decimal places for numeric labels.
- `use_siunitx=ld.siunitx`. This option specifies whether numeric values should be formatted using the *siunitx* package; the default value is that of the global variable `ld.siunitx`, which is `false` by default.
- `mylabels=""`. This option allows you to impose custom labels. When there are any, the value passed to the option must be a list of the type: `{pos1, "text1", pos2, "text2", ...}`. The number  $\langle pos1 \rangle$  represents an abscissa in the coordinate system  $(A, xstep)$  for  $x$ -axis, or  $(A, ystep*i)$  for  $y$ -axis, which corresponds to the affix point  $A+pos1*xstep$  for  $x$ -axis, and  $A+pos1*ystep*i$  for  $y$ -axis.

## Dgradline

The axis plotting methods are based on the method `g:Dgradline({A, u}, options)`, where  $\langle A, u \rangle$  represents the line passing through  $\langle A \rangle$  (a complex number) and directed by the vector  $\langle u \rangle$  (a non-zero complex number). The pair  $(A, u)$  serves as a reference point on this line (and orients this line), so each point  $M$  on this line has an abscissa  $x$  such that  $M = A + xu$ . This method allows you to draw this graduated line. The argument  $\langle options \rangle$  is a table specifying the possible options, which are (with their default values):

- `showaxe=1`. This option specifies whether or not the axis should be plotted (1 or 0).
- `arrows="-"`. This option allows you to add an arrow to the axis (no arrow by default; enter `"->"` to add an arrow).
- `limits="auto"`. This option allows you to specify the range of the two axes. The value `"auto"` means that it is the entire line, but you can specify the extreme abscissas, for example: `limits={-4.4}`.
- `gradlimits="auto"`. This option allows you to specify the range of the graduations on both axes. The value `"auto"` means that it is the entire line, but you can specify the extreme graduations, for example: `gradlimits={-2.3}`.
- `unit=1`. This option allows you to specify the number of graduations on the axis.
- `nbsubdiv=0`. This option specifies the number of subdivisions between two main ticks.
- `tickpos=0.5`. This option specifies the position of the ticks relative to the axis. These are two numbers between 0 and 1. The default value of 0.5 means they are centered on the axis. (0 and 1 represent the ends).
- `tickdir="auto"`. This option specifies the direction of the ticks on the axis. This direction is a non-zero vector (complex number). The default value `"auto"` means the ticks are orthogonal to the axis.
- `xyticks=0.2`. This option specifies the length of the ticks.
- `xylabelsep=defaulttxylabelsep`. This option specifies the distance between the labels and the tick marks. `defaulttxylabelsep` is a global variable with a default value of 0.
- `originpos="center"`. This option specifies the position of the label at the origin on the axis. Possible values are: `"none"`, `"center"`, `"left"`, `"right"`.
- `originnum=0`. This option specifies the value of the tick mark at the origin  $A$  (tick mark number 0).

The formula that defines the label at graduation number  $n$  (at point  $A + nu$ ) is:

$$(\text{originnum} + \text{unit} \cdot n) \text{labeltext} / \text{labelden}.$$

- `legend=""`. This option allows you to specify a legend for the axis.
- `legendpos=0.975`. This option specifies the position (between 0 and 1) of the legend relative to the axis.
- `legendsep=ld.defaultlegendsep`. This option specifies the distance between the legend and the axis. The legend is on the other side of the axis from the graduations; `ld.defaultlegendsep` is a global variable that defaults to 0.2.
- `legendangle="auto"`. This option specifies the angle (in degrees) that the legend should form for the axis. The default value `"auto"` means that the legend must be parallel to the axis if the `labelstyle` option is also set to `"auto"`, otherwise the legend is horizontal.

- `legendstyle="auto"`. Specifies the label style for the legend. With the value *"auto"*, the style is determined automatically; otherwise, the following values can be used: *"N"*, *"NW"*, *"W"*, *"SW"*, *"S"*, *"SE"*, *"E"*, *"NE"*.
- `labelpos="bottom"`. This option specifies the position of the labels relative to the axis. The possible values are: *"none"*, *"bottom"*, or *"top"*. This position also determines the position of the legend: on the opposite side of the axis.
- `labelden=1`. This option specifies the denominator of the labels (integer) for the axis. The formula that defines the label at graduation number  $n$  is:

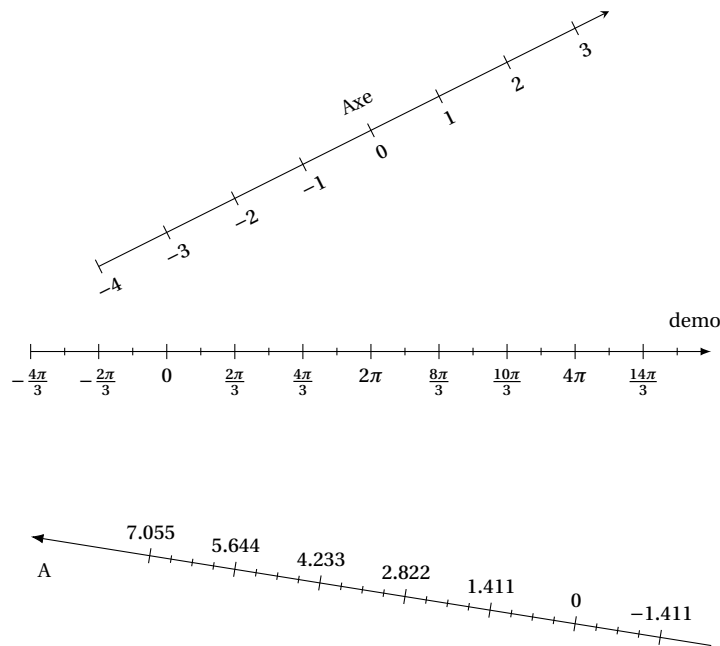
$$(\text{originnum} + \text{unit} * n) \text{"labeltext"} / \text{labelden}.$$

- `labeltext=""`. This option defines the text that will be added to the numerator of the labels.
- `labelstyle="auto"`. This option defines the label style. Possible values are: *"auto"*, *"N"*, *"NW"*, *"W"*, *"SW"*, *"S"*, *"SE"*, *"E"*, *"NE"*.
- `labelangle=0`. This option defines the angle of the labels in degrees from the horizontal.
- `labelcolor=""`. This option allows you to choose a color for the labels. The empty string represents the current text color.
- `labelshift=0`. This option allows you to define a systematic offset for the labels on the axis (along axis offset).
- `nbdec=2`. This option specifies the number of decimal places for numeric labels.
- `use_siunitx=ld.siunitx`. This option specifies whether numeric values should be formatted using the *siunitx* package; the default value is that of the global variable `ld.siunitx`, which is *false* by default.
- `mylabels=""`. This option allows you to impose custom labels. When any are present, the value passed to the option must be a list of the type:  $\{x_1, \text{"text1"}, x_2, \text{"text2"}, \dots\}$ . The numbers  $\langle x_1 \rangle, \langle x_2 \rangle, \dots$ , represent abscissas in the  $(A, u)$  coordinate system.

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



Figure 17: Examples of numbered lines



## Dgrid

The `g:Dgrid({A,B}, options)` method allows you to draw a grid.

- The first argument is mandatory; it specifies the lower-left corner (this is the  $\langle A \rangle$  complex number) and the upper-right corner (this is the  $\langle B \rangle$  complex number) of the grid.
- The  $\langle options \rangle$  argument is a table specifying the possible options. Here are these options with their default values:
  - `unit={1,1}`. This option defines the units on the axes for the main grid.
  - `showlines={true,true}`. This option allows you to display or hide the vertical lines of the grid (corresponding to the  $x$  axis) as well as the horizontal lines of the grid (corresponding to the  $y$  axis).
  - `gridwidth=4`. This option defines the line thickness of the main grid (0.4pt by default).
  - `gridcolor="gray"`. Grid color of the main grid.
  - `gridstyle="solid"`. Line style for the primary grid.
  - `nbsubdiv=0,0`. Number of subdivisions (for each axis) between two lines of the primary grid. These subdivisions determine the secondary grid.
  - `subgridcolor="lightgray"`. Color of the secondary grid.
  - `subgridwidth=2`. Line thickness of the secondary grid (0.2pt by default).
  - `subgridstyle="solid"`. Line style for the secondary grid.
  - `originloc=A`. Location of the grid origin.

**Example:** It is possible to work in a non-orthogonal coordinate system. Here is an example where the  $x$ -axis is retained, but the first bisector becomes the new  $y$ -axis. We modify the graph's transformation matrix. Based on this modification, the affixes represent the coordinates in the new coordinate system.

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

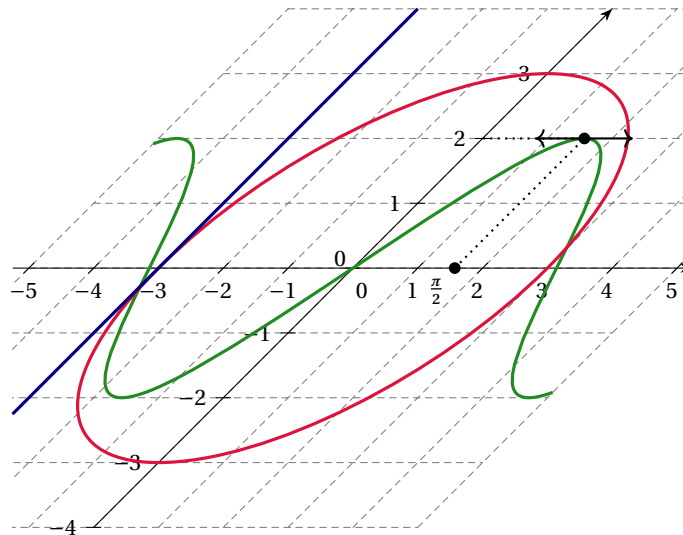


```

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

```

Figure 18: Example of a non-orthogonal coordinate system



## Dgradbox

The `g:Dgradbox({A, B, xstep, ystep}, options)` method allows you to draw a graduated box.

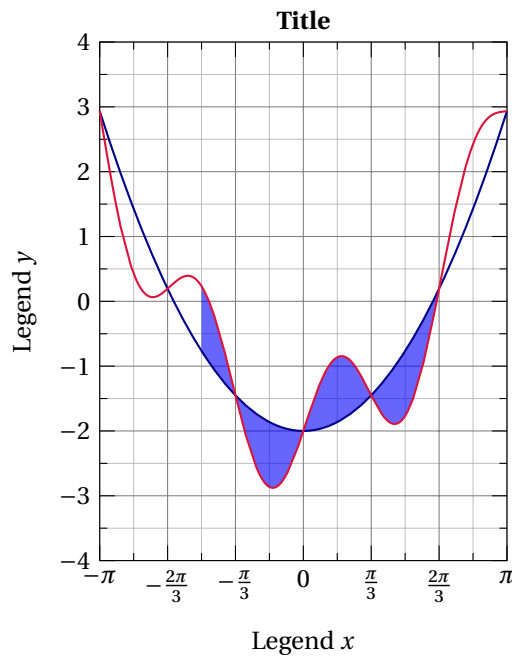
- The first argument is mandatory; it specifies the lower-left corner (this is the  $\langle A \rangle$  complex number) and the upper-right corner (this is the  $\langle B \rangle$  complex number) of the box, as well as the step on each axis.
- The  $\langle options \rangle$  argument is a table specifying the possible options. These are the same as for the axes, except for some default values. In addition, the following option is added: `title=""`, which allows you to add a title at the top of the box; however, be careful to leave enough space for this.

```

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

```

Figure 19: Using Dgradbox



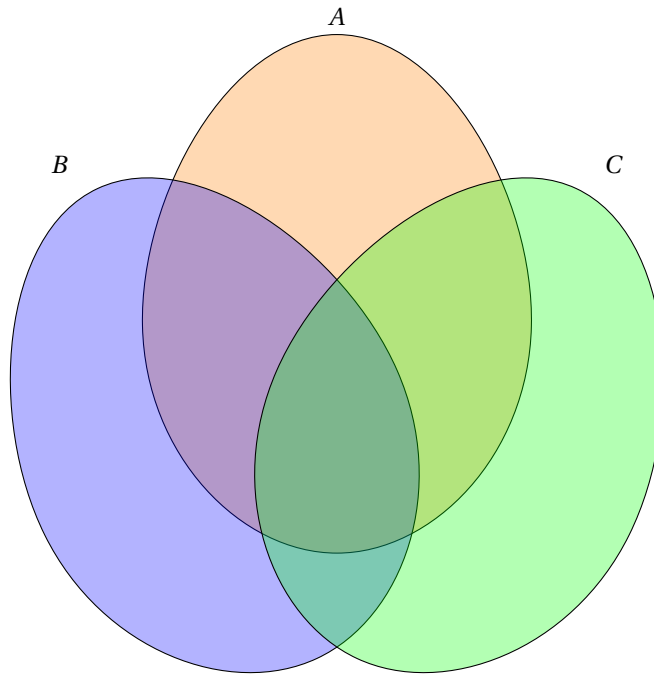
## 10) Set Drawings (Venn Diagrams)

### Drawing a Set

The function `ld.set(center [, angle, scale])` returns a path representing a set (egg-shaped), with the center being  $\langle center \rangle$  (complex number), the argument  $\langle angle \rangle$  representing the inclination (in degrees) of the set's vertical axis (0 by default), and the argument  $\langle scale \rangle$  being a scale factor to modify the size of the set (1 by default). Such a path can be drawn with the method `g:Dpath()`.

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

Figure 20: Drawing a Set



### Operations on Sets

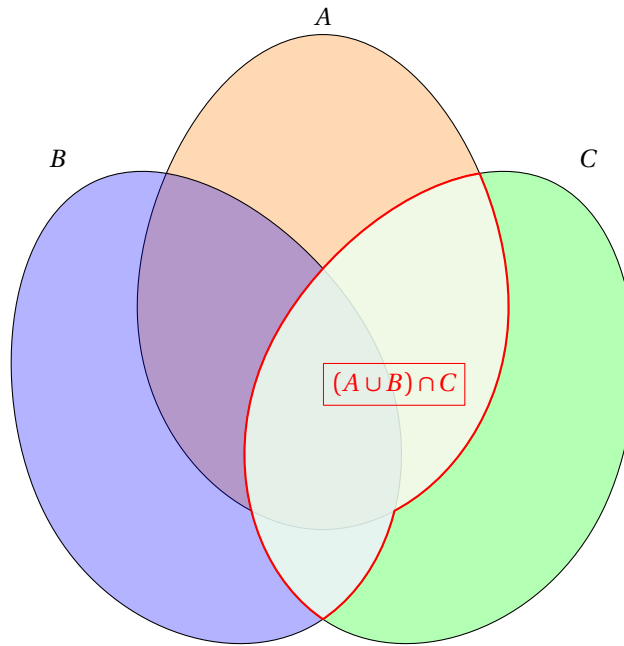
Let  $C_1$  and  $C_2$  be two lists of complex numbers representing the contour of two sets (simple closed curves, all in one piece). There are three possible operations:

- The function **ld.cap(C1, C2)** returns a list of complex numbers representing the contour of the intersection of the sets corresponding to  $C_1$  and  $C_2$ .
- The function **ld.cup(C1, C2)** returns a list of complex numbers representing the contour of the union of the sets corresponding to  $C_1$  and  $C_2$ .
- The function **ld.setminus(C1, C2)** returns a list of complex numbers representing the contour of the difference of the sets corresponding to  $C_1$  and  $C_2$  ( $C_1 \setminus C_2$ ).

The result of these operations, being a list of complex numbers, can be drawn with the **g:Dpolyline()** method.

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

Figure 21: Set Operations



**NB** : The result is not always satisfactory when the contours become too complex, or when the contours share common sections.

## 11) Importing an Image

### Placing an Image in the Graphic

To import an image into the current graphic (and possibly draw on top of it), use the method:

**g:Dimage( filename, anchor, options)**

where:

- $\langle filename \rangle$  is the full name of the image file. The image will be displayed using the `\includegraphics[]{}{}` command within a node.
- $\langle anchor \rangle$  is a complex number representing the anchor point of the image in the graphic.
- $\langle options \rangle$  is a table whose fields define the display parameters (with their default values):
  - `pos="center"` specifies the position of the image relative to the anchor point, following the same principle as for labels. Possible values are: `"center"`, `"N"`, `"NE"`, `"E"`, `"SE"`, `"S"`, `"SW"`, `"W"`, `"NW"`.
  - `name=""` optionally assigns a name to the node that will be created. This name can later be referenced in TikZ.
  - `graphics_options=""` is a string containing the options to be passed to the `\includegraphics` command.
  - `matrix=nil` is a table representing an affine transformation matrix. This matrix is applied locally to the created node (for which the origin is the anchor point). Note that the current global 2D transformation matrix of the graphic is also taken into account by this method.

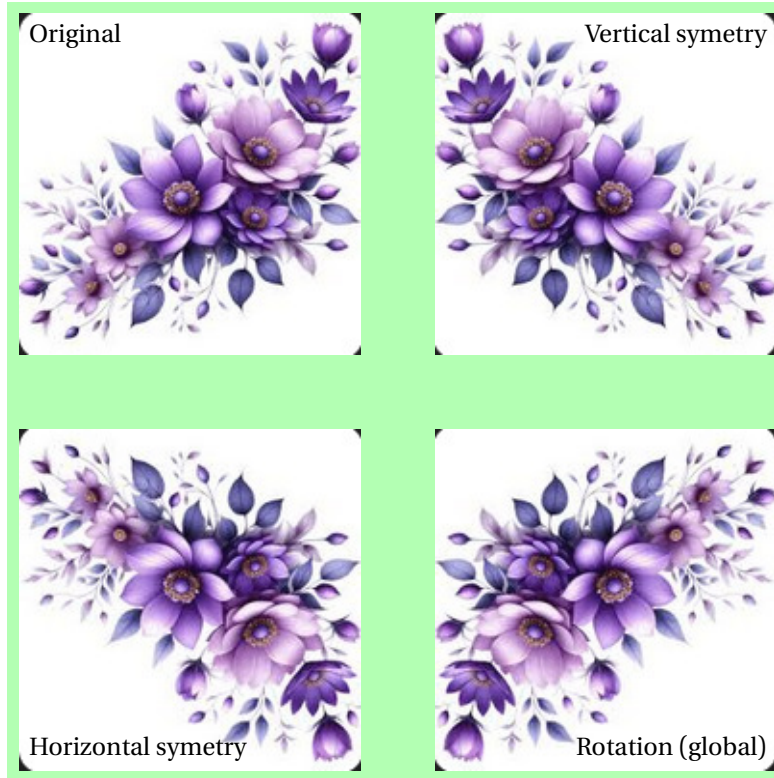
```
\begin{luadraw}{name=Dimage}
local ld = luadraw
local Z, i = ld.cpx.Z, ld.cpx.I
local g = ld.graph:new{margin={0,0,0,0}, size={10,10},bg="green!30"}
local filename = ld.cachedir.."flower.jpg"
local g_options = "width=4.5cm,height=4.5cm"
g:Dimage(filename,Z(-5,5),{pos="SE", graphics_options=g_options})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

```

g:Dimage(filename,Z(-5,-5),{pos="SE", matrix={0,1,-i}, graphics_options=g_options})
g:Dlabel( "Original", Z(-5,5),{pos="SE"},
"Vertical symetry", Z(5,5), {pos="SW"},
"Rotation (global)", Z(5,-5), {pos="NW"},
"Horizontal symetry", Z(-5,-5), {pos="NE"} )
g:Show()
\end{luadraw}

```

Figure 22: Importing an Image



### Mapping an Image onto a Parallelogram

This can be done using the method: **g:Dmapimage( filename, parallelo, options)**, where:

- $\langle filename \rangle$  is the full name of the image file.
- $\langle parallelo \rangle$  is a list of the form  $\{a, u, v\}$ , consisting of three complex numbers. This list defines the parallelogram with vertices  $\{a, a + u, a + u + v, a + v\}$ .
- $\langle options \rangle$  is a table whose fields specify the parameters (with their default values):
  - **name=""** optionally assigns a name to the node that is created. This name can later be referenced in TikZ.
  - **clip=false** is a Boolean flag indicating whether the image should be clipped to the parallelogram. In most cases, clipping is not required.
  - **border\_options=nil**, when set to a value other than **nil**, must be a string containing the drawing options for the boundary of the parallelogram. The value of **border\_options** is passed directly to the **\draw** command to render the outline.

```

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

```

```

end
g:Dmapimage(ld.cachedir.."russell.jpg", portrait, {border_options="lightgray,line width=1.5mm"})
g:Show()
\end{luadraw}

```

Figure 23: Mapping an Image



## 12) Colors

In the *luadraw* environment, colors are character strings that must correspond to colors known to TikZ. The *xcolor* package is strongly recommended so as not to be limited to basic colors.

### Color Calculations

To be able to manipulate colors, they have been defined (in the *luadraw\_colors.lua* module) as tables of three components: red, green, blue, each component being a number between 0 and 1, and with their names in the *svgnames* format of the *xcolor* package. For example, we find (among others) the declarations:

```

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

```

You can refer to the *xcolor* documentation for a list of these colors.

To use these in the *luadraw* environment, you can:

- either use them by name if you have declared them in the preamble: `\usepackage[svgnames]{xcolor}`, for example: `g:Linecolor("AliceBlue")`,
- or use them with the *luadraw* **ld.rgb()** function, for example: `g:Linecolor(ld.rgb(AliceBlue))`. However, with this **ld.rgb()** function, to change the color locally, you must do the following (example):  
`g:Dpolyline(L,"color"..ld.rgb(AliceBlue)), or g:Dpolyline(L,"fill"..ld.rgb(AliceBlue))`. Because the **rgb()** function does not return a color name, but a color definition.

**Functions for color management:**

- The **ld.rgb(*r*, *g*, *b*)** or **ld.rgb({*r*, *g*, *b*})** function returns the color as a string understandable by TikZ in the `color=...` and `fill=...` options. The values of *r*, *g*, and *b* must be between 0 and 1.
- The **ld.hsb(*h*, *s*, *b* [, *table*])** function returns the color as a string understandable by TikZ. The *h* (hue) argument must be a number between 0 and 360, the *s* (saturation) argument must be between 0 and 1, and the *b* (brightness) argument must also be between 0 and 1. The (optional) argument *table* is a boolean (false by default) that indicates whether the result should be returned as a table  $\{r, g, b\}$  or not (by default it is as a string).
- The function **ld.mixcolor(*color1*, *proportion1*, *color2*, *proportion2*, ..., *colorN*, *proportionN*)** mixes the colors  $\langle color1 \rangle$ , ...,  $\langle colorN \rangle$  in the requested proportions and returns the resulting color as a string understandable by TikZ, followed by the same color as a table  $\{r, g, b\}$ . Each color must be a table of three components  $\{r, g, b\}$ .
- The function **ld.mixpalette(*pal*, *percent*, *color*)** returns a new palette after mixing each color from the palette  $\langle pal \rangle$  with  $\langle color \rangle$ . The argument  $\langle pal \rangle$  is a list of colors, each of which is a table of three components  $\{r, g, b\}$ , the argument  $\langle percent \rangle$  is a number between 0 and 100, and the argument  $\langle color \rangle$  is a color in the form of a table  $\{r, g, b\}$ .
- The function **ld.palette(*colors*, *pos* [, *table*])**: the argument  $\langle colors \rangle$  is a list (table) of colors in the format  $\{r, g, b\}$ , the argument  $\langle pos \rangle$  is a number between 0 and 1, the value 0 corresponds to the first color in the list and the value 1 to the last. The function calculates and returns the color corresponding to the position  $\langle pos \rangle$  in the list by linear interpolation. The (optional) argument *table* is a boolean (false by default) that indicates whether the result should be returned as a table  $\{r, g, b\}$  or not (by default it is as a string).
- The **ld.getpalette(*colors*, *nb* [, *table*])** function: the  $\langle colors \rangle$  argument is a list (table) of colors in  $\{r, g, b\}$  format, the  $\langle nb \rangle$  argument indicates the desired number of colors. The function returns a list of  $\langle nb \rangle$  colors evenly distributed in  $\langle colors \rangle$ . The (optional) *table* argument is a boolean (**false** by default) that indicates whether the colors are returned as  $\{r, g, b\}$  tables or not (by default, they are as strings).
- The method **g:Newcolor(*name*, *color*)** allows you to define a new color in the TikZ export to rgb format. The name of this color will be  $\langle name \rangle$  (string). The argument  $\langle color \rangle$  can be either a table of three components: red, green, blue (between 0 and 1) defining this color, or a string representing a color. In the first case, the method uses a `\definecolor`, and in the second case, it uses a `\colorlet`.

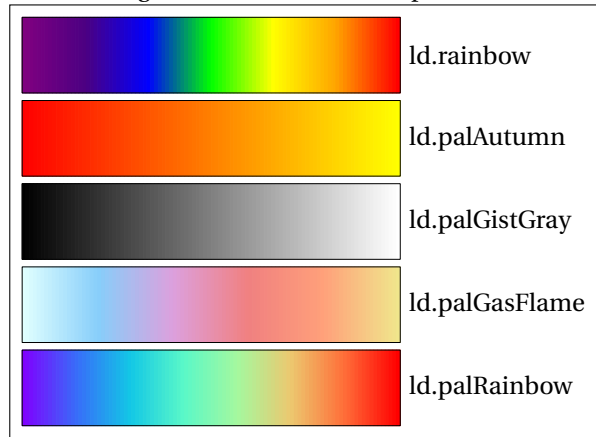
You can also use all of TikZ's usual color management features.

By default, there are five color palettes.<sup>2</sup>

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

<sup>2</sup>A palette is a table of colors; these are themselves tables of numbers between 0 and 1 representing the red, green, and blue components.

Figure 24: The five default palettes



### III Geometric Constructions

This section groups together functions that construct geometric figures without a corresponding dedicated graphical method.

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

- The function `ld.circumcircle(a, b, c)` (or `ld.circumcircle({a, b, c})`), where  $\langle a \rangle$ ,  $\langle b \rangle$ , and  $\langle c \rangle$  are three points (three complex numbers), returns the circumcircle of the triangle formed by these three points, in the form of a sequence:  $C, r$ , where  $C$  is the center of the circle (a complex number), and  $r$  is its radius.
- The function `ld.incircle3d(a, b, c)` (or `ld.incircle3d({a, b, c})`), where  $\langle a \rangle$ ,  $\langle b \rangle$ , and  $\langle c \rangle$  are three points (three complex numbers), returns the incircle of the triangle formed by these three points, as a sequence:  $C, r$ , where  $C$  is the center of the circle (a complex number), and  $r$  is its radius.

#### 2) `cvx_hull2d()`

The function `ld.cvx_hull2d(L)`, where  $\langle L \rangle$  is a list of complex numbers, computes and returns a list of complex numbers representing the convex hull of  $\langle L \rangle$ .

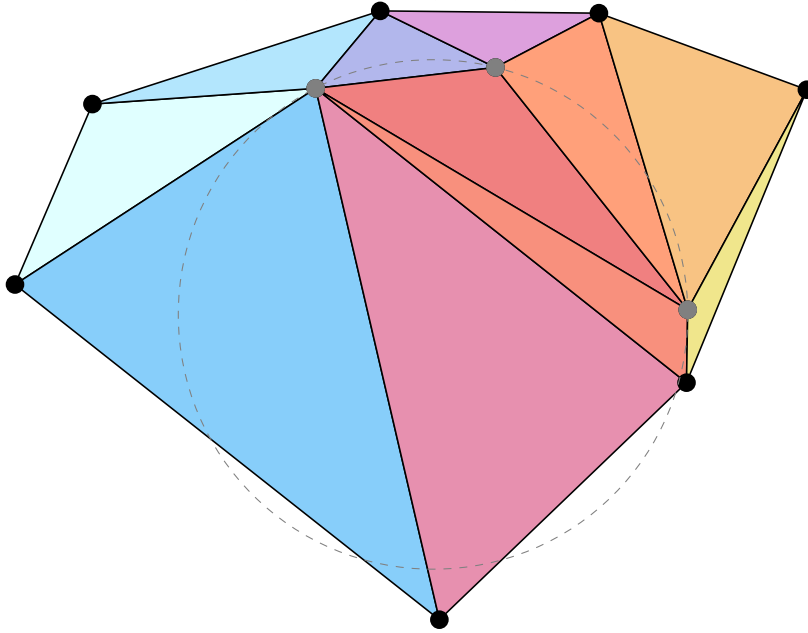
#### 3) `delaunay()`

The function `ld.delaunay(L)` where  $\langle L \rangle$  is a list of **distinct** complex numbers, returns a list of triangles (a triangle being a list of three complex numbers) obtained by Delaunay triangulation of the points of  $\langle L \rangle$  (the circumcircle of each triangle does not contain any of the other points).

```
\begin{luadraw}{name=delaunay}
local ld = luadraw
local g = ld.graph:new{bbox=false, pictureoptions="scale=2"}
local i = ld.cpx.I; g:Linewidth(6)
local L = {0.285+1.46*i, 1.556-0.142*i, 2.344+1.313*i, -2.38+1.218*i, 1.548-0.624*i,
0.969+1.819*i, -0.086-2.191*i, -0.477+1.834*i, -0.904+1.322*i, -2.892+0.025*i}
local T = ld.delaunay(L) -- list of triangles
local n = #T
local num = 7 --we choose a triangle
local colors = ld.getpalette(ld.palGasFlame, n)
for k = 1, n do
    g:Dpolyline(T[k], true, 'fill=' .. colors[k])
end
g:Ddots(L)
g:Dcircle( {ld.circumcircle(T[num])}, "line width=0.4pt, gray, dashed" )
g:Ddots(T[num], "gray")
g:Show()
\end{luadraw}
```



Figure 25: Delaunay Triangulation

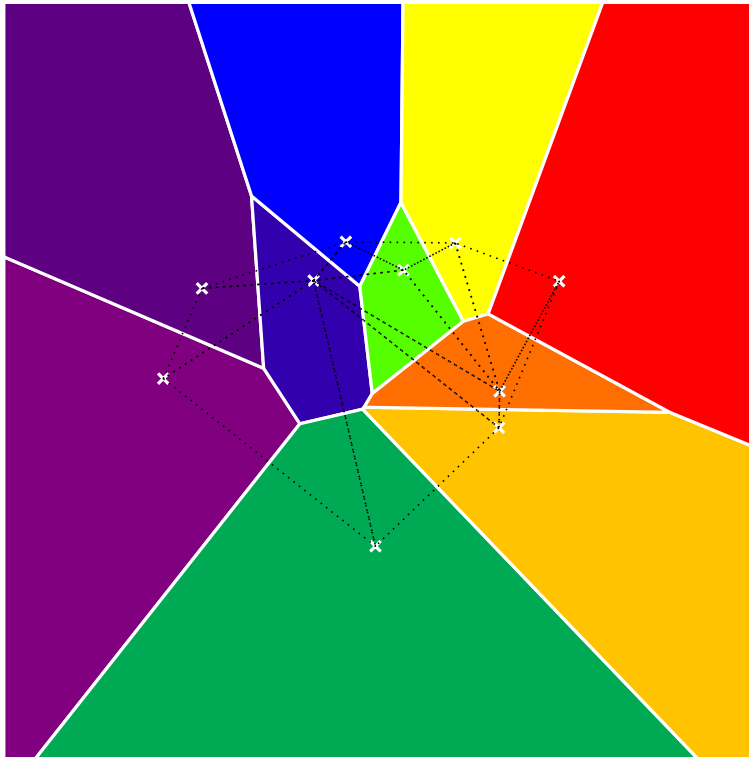


#### 4) voronoi()

The function `ld.voronoi(L [, window])`, where  $\langle L \rangle$  is a list of distinct complex numbers, determines the Voronoi diagram of the points in the list  $\langle L \rangle$ . This function returns a list of elements of the form  $\{A, \text{polygon}\}$ , where  $A$  is a point in the list  $\langle L \rangle$ , and `polygon` is a list of complex numbers representing the vertices of the cell associated with  $A$ . Thus, there is one cell for each point in  $\langle L \rangle$ . The cell for point  $A$  contains the points in the plane that are closer to  $A$  than to other points in  $\langle L \rangle$ . This function uses Delaunay triangulation. The optional argument  $\langle \text{window} \rangle$ , which defaults to  $\{-5, 5, -5, 5\}$ , is used to clip Voronoi cells that are unbounded; this window is automatically enlarged if necessary to contain all the points of  $\langle L \rangle$  as well as all the centers of the circles circumscribed about the Delaunay triangles (note: this does not change the 2D window of the current graph).

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

Figure 26: Voronoï diagram



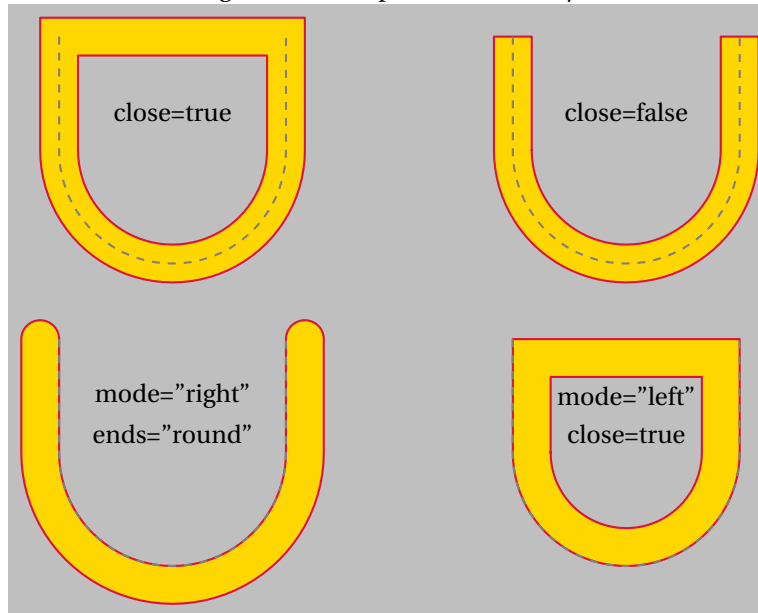
## 5) line2strip()

The function `ld.line2strip(L, width [, close, ends, mode])` where  $\langle L \rangle$  is a list of complex numbers, or a list of lists of complex numbers, returns a path representing a "strip" calculated on  $\langle L \rangle$  and of width  $\langle width \rangle$ . The optional argument  $\langle close \rangle$  is a boolean that indicates whether  $\langle L \rangle$  should be closed (`false` by default).

The optional argument  $\langle ends \rangle$  specifies how the two ends of the strip are drawn. Possible values are: `"none"`, `"butt"` (default), or `"round"`. To maintain compatibility with the older version, this argument can also take the values `true` (equivalent to `"butt"`) or `false` (equivalent to `"none"`).

The argument  $\langle mode \rangle$  can be either `"center"` (default value), or `"left"`, or `"right"`, in the first case the band is centered on  $\langle L \rangle$ , in the second case it is on the left of  $\langle L \rangle$ , and in the third case it is on the right of  $\langle L \rangle$ .

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

Figure 27: Example with *line2strip*

## 6) `parallel_polyline()`

The function `ld.parallel_polyline(L, width [, close])` where  $\langle L \rangle$  is a list of complex numbers, or a list of lists of complex numbers, returns a polygonal line parallel to  $\langle L \rangle$  and located at a "distance" equal to  $\langle width \rangle$ . The argument  $\langle width \rangle$  can be positive or negative to be on one side or the other of  $\langle L \rangle$  (this depends on the direction of traversal of  $\langle L \rangle$ ). The optional argument  $\langle close \rangle$  is a boolean that indicates whether  $\langle L \rangle$  should be closed (`false` by default).

## 7) `sss_triangle()`

The function `ld.sss_triangle(ab, bc, ca)`, where  $\langle ab \rangle$ ,  $\langle bc \rangle$ , and  $\langle ca \rangle$  are three lengths, computes and returns a list of three points (3 complex numbers)  $\{A, B, C\}$  forming the vertices of a direct triangle whose side lengths are the arguments, i.e.,  $AB = ab$ ,  $BC = bc$ , and  $CA = ca$ , when possible. Vertex  $A$  is always the complex number 0 and vertex  $B$  is always the complex number  $ab$ . This triangle can be drawn with the `g:Dpolyline` method.

## 8) `sas_triangle()`

The function `ld.sas_triangle(ab, alpha, ca)`, where  $\langle ab \rangle$  and  $\langle ca \rangle$  are two lengths and  $\langle alpha \rangle$  is an angle in degrees, calculates and returns a list of three points (3 complex numbers)  $\{A, B, C\}$  forming the vertices of a triangle such that  $AB = ab$ ,  $CA = ca$ , and angle  $(\vec{AB}, \vec{AC})$  has a measure of  $\langle alpha \rangle$ , whenever possible. Vertex  $A$  is always the complex number 0 and vertex  $B$  is always the complex number  $ab$ . This triangle can be drawn with the `g:Dpolyline` method.

## 9) `asa_triangle()`

The function `ld.asa_triangle(alpha, ab, beta)`, where  $\langle ab \rangle$  is a length,  $\langle alpha \rangle$  and  $\langle beta \rangle$  are two angles in degrees, calculates and returns a list of three points (3 complex numbers)  $\{A, B, C\}$  forming the vertices of a triangle such that  $AB = ab$ , such that angle  $(\vec{AB}, \vec{AC})$  has measure  $\langle alpha \rangle$ , and such that angle  $(\vec{BA}, \vec{BC})$  has measure  $\langle beta \rangle$ , whenever possible. Vertex  $A$  is always the complex number 0 and vertex  $B$  is always the complex number  $ab$ . This triangle can be drawn using the `g:Dpolyline` method.

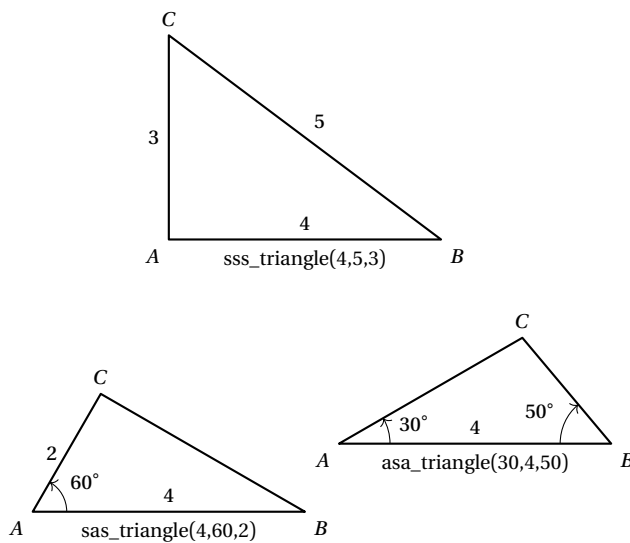
```
\begin{luadraw}{name=sss_triangles_and_co}
local ld = luadraw
local g = ld.graph:new{window={-5,5,-3,5},size={10,10}}
g:Labelsize("footnotesize"); g:Linewidth(8)
local Zp, i, deg = ld.cpx.Zp, ld.cpx.I, ld.deg
local T1 = ld.shift( ld.sss_triangle(4,5,3), 2*i-2)
local T2 = ld.shift( ld.sas_triangle(4,60,2),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

```

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

```

Figure 28: sss\_triangle, sas\_triangle and asa\_triangle



## IV Computations on Lists

### 1) concat

The function `ld.concat(table1, table2, ...)` concatenates all the tables passed as arguments and returns the resulting table.

- Each argument can be a real number, a complex number, or a table.
- Example: The instruction `ld.concat(1,2,3,4,5,6,7)` returns the table `{1,2,3,4,5,6,7}`.

### 2) cut

The function `ld.cut(L, A [, before])` cuts  $\langle L \rangle$  at the point  $\langle A \rangle$ , which is assumed to be located on the line  $\langle L \rangle$  (which is either a list of complex numbers or a polygonal line, i.e., a list of lists of complex numbers). If the argument  $\langle before \rangle$  is `false` (the default value), then the function returns the part before  $\langle A \rangle$ , followed by the part after  $\langle A \rangle$ ; otherwise, the reverse is true.

### 3) cutpolyline

The function `ld.cutpolyline(L, D [, close])` cuts the polygonal line  $\langle L \rangle$  with the straight line  $\langle D \rangle$ . The argument  $\langle L \rangle$  must be a list of complex numbers or a list of lists of complex numbers, the argument  $\langle D \rangle$  is a list of the form  $\{A, u\}$  where  $A$  is a complex (point on the line) and  $u$  is a non-zero complex (direction vector of the line). The argument  $\langle close \rangle$  indicates whether the line  $\langle L \rangle$  should be closed (`false` by default). The function returns three things:

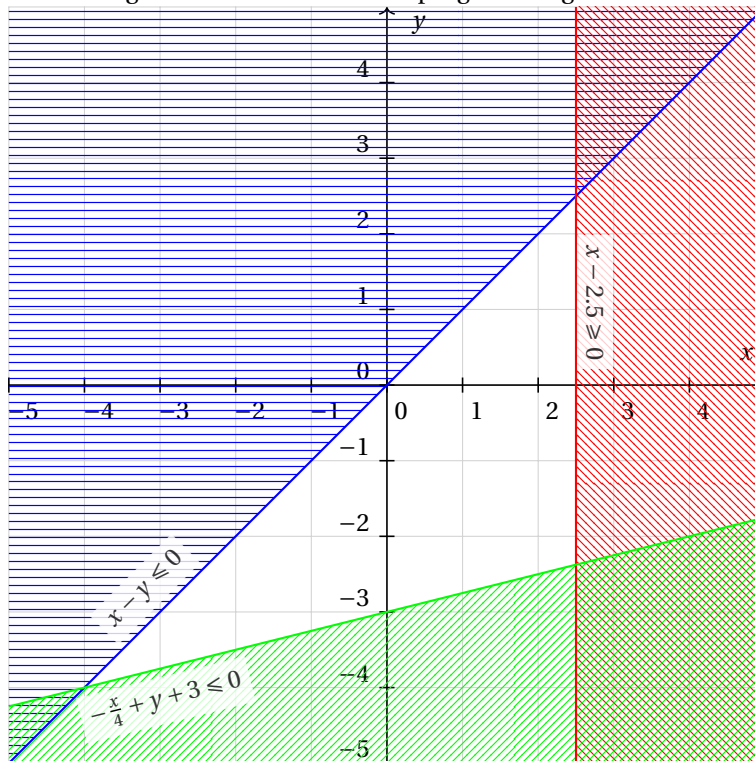
- The part of  $\langle L \rangle$  that is in the half-plane defined by the line to the "left" of  $u$  (i.e., containing the point  $A + iu$ ) (it is a polygonal line),
- followed by the part of  $\langle L \rangle$  that is in the other half-plane (polygonal line),
- followed by the list of intersection points between  $\langle L \rangle$  and the line  $\langle D \rangle$ .

```

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

```

Figure 29: Illustrate a linear programming exercise



#### 4) cutpolyline2

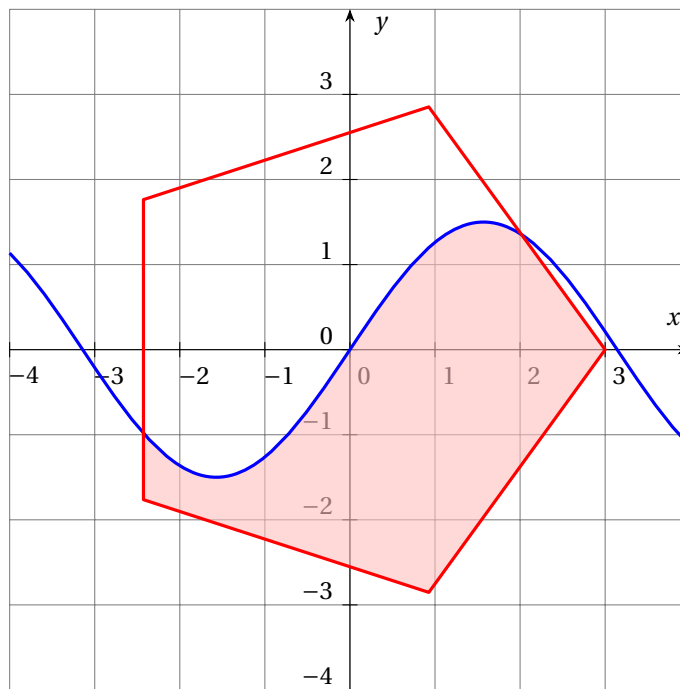
The function `ld.cutpolyline2(P, f, sg, [, close])` cuts the polygon  $\langle P \rangle$  with the graph of the function  $\langle f \rangle$ . The argument  $\langle P \rangle$  must be a list of complex numbers, and the argument  $\langle f \rangle$  is a function  $f: x \mapsto f(x) \in \mathbf{R}$ . The argument  $\langle sg \rangle$  is either  $>$  or  $<$ . In the first case, the function returns the subset of the polygon  $\langle P \rangle$  whose points satisfy  $y > f(x)$ , and in the second case, it returns the points satisfying  $y < f(x)$ . The optional argument  $\langle close \rangle$  indicates whether  $\langle P \rangle$  should be closed.

```

\begin{luadraw}{name=cupolyline2}
local ld = luadraw
local g = ld.graph:new{window = {-4, 4, -4, 4}, size = {10,10}}
g:Daxes({0,1,1}, {grid=true, arrows="-Stealth", legend={"$x$", "$y$"}})
local f = function(x) return 1.5*math.sin(x) end
local P = ld.polyreg(0,3,5) -- pentagone
local L = ld.cutpolyline2(P,f,'<',true) -- partie du polygone située sous la courbe de f
g:Dpolyline(L,true,"draw=none,fill=pink,fill opacity=0.6")
g:Linewidth(12)
g:Dcartesian(f, {draw_options="blue"})
g:Dpolyline(P,true,"red")
g:Show()
\end{luadraw}

```

Figure 30: cutpolyline2



## 5) getbounds

- The function `ld.getbounds(L)` returns the bounds `xmin, xmax, ymin, ymax` of the polygonal line  $\langle L \rangle$ .
- Example: `local xmin,xmax,ymin,ymax=ld.getbounds(L)` (where  $\langle L \rangle$  denotes a polygonal line).

## 6) getdot

The function `ld.getdot(x, L)` returns the point with abscissa  $\langle x \rangle$  (real between 0 and 1) along the connected component  $\langle L \rangle$  (list of complex numbers). The abscissa 0 corresponds to the first point and the abscissa 1 to the last. More generally,  $\langle x \rangle$  corresponds to a percentage of the length of  $L$ .

## 7) insert

The function `ld.insert(table1, table2 [, pos])` inserts the elements of  $\langle table2 \rangle$  into  $\langle table1 \rangle$  at position  $\langle pos \rangle$ .

- The argument  $\langle table2 \rangle$  can be a real number, a complex number, or a table.
- The argument  $\langle table1 \rangle$  must be a variable that designates a table; this will be modified by the function.
- If the argument  $\langle pos \rangle$  is `nil`, the insertion is performed at the end of  $\langle table1 \rangle$ .

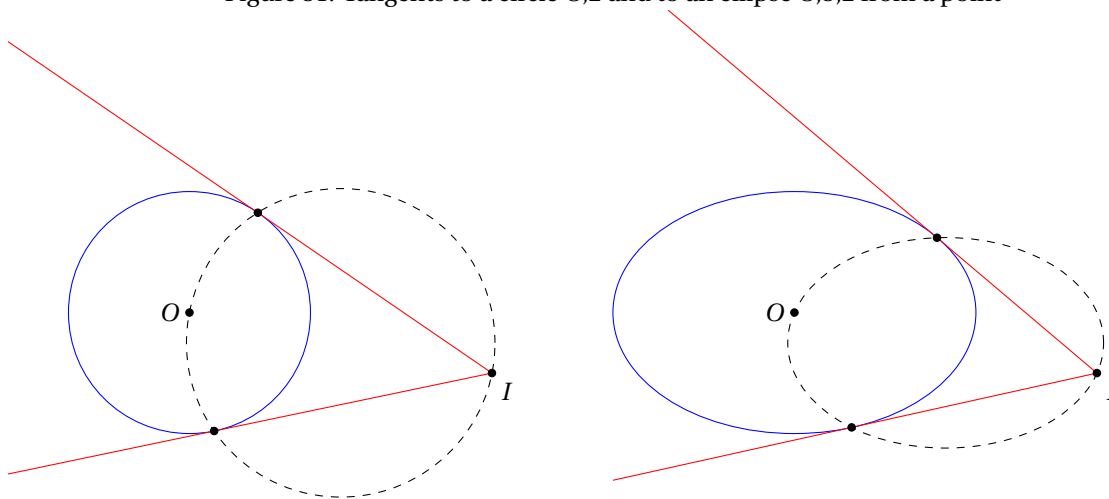
- Example: If a variable  $L$  is equal to  $\{1, 2, 6\}$ , then after the instruction `ld.insert(L, 3, 4, 5, 3)`, the variable  $L$  will be equal to  $\{1, 2, 3, 4, 5, 6\}$ .

## 8) interCC

The function `ld.interCC(C1, C2)` returns the intersection of circle  $\langle C1 \rangle$  with circle  $\langle C2 \rangle$ , where  $C1=\{O1, r1\}$  (circle with center  $O_1$  and radius  $r_1$ ), and  $C2=\{O2, r2\}$  (circle with center  $O_2$  and radius  $r_2$ ). The function returns a list containing 1 or 2 points or the entire circle if the intersection is not empty; otherwise, it returns `nil`.

```
\begin{luadraw}{name=interCC}
local ld = luadraw
local cpx = ld.cpx
local g = ld.graph:new{window={-10,10,-5,5}, margin={0,0,0,0},size={16,8}}
local i = cpx.I
-- pour le cercle {0,2}
g:Saveattr(); g:Viewport(-10,0,-5,5); g:Coordsystem(-4,6,-5,5)
local O = -1
local C1, I = {0, 2}, 4-i
local C2 = {(O+I)/2, cpx.abs(I-O)/2}
local rep = ld.interCC(C1, C2) -- points of tangency
g:Dcircle(C1, "blue"); g:Dcircle(C2, "dashed")
g:Dhline(I, rep[1], "red"); g:Dhline(I, rep[2], "red") --half-tangents
g:Ddots(rep); g:Ddots({0, I}); g:Dlabel("$I$", I, {pos="SE"}, "$O$", 0, {pos="W"},
"tangents to the circle arising from $I$", 1-5*i, {pos="N"})
g:Restoreattr()
-- pour l'ellipse (E) : {0,3,2}
g:Saveattr(); g:Viewport(0,10,-5,5); g:Coordsystem(-4,6,-5,5)
local mat = {0,1.5,i} -- This matrix transforms a circle {01,2} into the ellipse (E)
local inv_mat = ld.invmatrix(mat) -- inverse matrix
local O1, I1 = table.unpack( ld.mtransform({0,I}, inv_mat) ) -- antecedents of O and I
C1 = {O1, 2}
C2 = {(O1+I1)/2, cpx.abs(I1-O1)/2}
rep = ld.interCC(C1, C2) -- points of tangency (tangents from I1)
g:Composematrix(mat) -- The matrix is applied to find the ellipse; the tangency is preserved.
g:Dcircle(C1, "blue"); g:Dcircle(C2, "dashed")
g:Dhline(I1, rep[1], "red"); g:Dhline(I1, rep[2], "red")
g:Ddots(rep); g:Ddots({O1, I1}); g:Dlabel("$I$", I1, {pos="SE"}, "$O$", O1, {pos="W"},
"tangents to the ellipse arising from $I$", 1-5*i, {pos="N"})
g:Restoreattr()
g:Show()
\end{luadraw}
```

Figure 31: Tangents to a circle  $O, 2$  and to an ellipse  $O, 3, 2$  from a point



tangents to the circle arising from  $I$

tangents to the ellipse arising from  $I$

**9) interD**

The function **ld.interD(d1, d2)** returns the intersection point of the lines  $\langle d1 \rangle$  and  $\langle d2 \rangle$ . A line is a list of two complex numbers: a point on the line and a direction vector.

**10) interDC**

The function **ld.interDC(d, C)** returns the intersection of the line  $\langle d \rangle$  with the circle  $\langle C \rangle$ , where  $d=\{A, u\}$  (a line passing through  $A$  and directed by  $u$ ), and  $C=\{O, r\}$  (a circle with center  $O$  and radius  $r$ ). The function returns a list containing 1 or 2 points if the intersection is not empty; otherwise, it returns `nil`.

**11) interDL**

The function **ld.interDL(d, L)** returns the list of intersection points between the straight line  $\langle d \rangle$  and the polygonal line  $\langle L \rangle$ .

**12) interL**

The function **ld.interL(L1, L2)** returns the list of intersection points of the polygonal lines defined by  $\langle L1 \rangle$  and  $\langle L2 \rangle$ . These two arguments are two lists of complex numbers or two lists of lists of complex numbers.

**13) interP**

The function **ld.interP(P1, P2)** returns the list of intersection points of the paths defined by  $\langle P1 \rangle$  and  $\langle P2 \rangle$ . These two arguments are two lists of complex numbers and instructions (see *Dpath* page 37).

**14) linspace**

The function **ld.linspace(a, b [, nbdots])** returns a list of  $\langle nbdots \rangle$  equally distributed numbers from  $\langle a \rangle$  to  $\langle b \rangle$ . By default,  $\langle nbdots \rangle$  is 50.

**15) map**

The function **ld.map(f, list)** applies the function  $\langle f \rangle$  to each element of the  $\langle list \rangle$  and returns the table of results. When a result is `nil`, the complex `cpx.Jump` is inserted into the list.

**16) merge**

The function **ld.merge(L [, epsilon])** reassembles, if possible, the connected components of  $\langle L \rangle$ , which must be a list of lists of complex numbers. The function returns a new polygonal line. Comparisons are made to the nearest  $\langle epsilon \rangle$ , by default we have  $\langle epsilon \rangle = 10^{-10}$ .

**17) polyline2path**

The function **ld.polyline2path(L)** where  $\langle L \rangle$  is a list of complex numbers or a list of lists of complex numbers, returns  $\langle L \rangle$  as a path (which can be drawn with the **g:Dpath()** method).

**18) range**

The function **ld.range(a, b [, step])** returns the list of numbers from  $\langle a \rangle$  to  $\langle b \rangle$  with a step equal to  $\langle step \rangle$ , which is 1 by default.



## 19) read\_csv\_file

The function `ld.read_csv_file(file, options)` reads a *csv* file. The argument *file* is a string representing the file with the extension: "`<name>.csv`". The argument *options* is an table whose fields represent the options, which are (with their default values):

- `header=true`, with the value `true` this means that the first line of the file contains the column names.
- `dic=false`, with the value `true` this means that the returned result will be a list of dictionaries (one per line), the keys of these dictionaries being the column names. With the value `false` (default value), the result is a list of lists of values (one list of values per line). When the `header` option is set to `false`, the `dic` option automatically remains `false`.
- `sep=","`, a string indicating the separator for the values on each line.
- `num=true`, a boolean indicating whether the values should be automatically converted to numeric values (when this conversion fails, the value remains a string).
- `comment="%%"`, a string announcing a comment (a line starting with these characters is ignored). **Warning:** the search uses Lua's *match* method, for which the `%` character is a special character; it must be doubled ("`%%`") to be considered a character.

The result returned by this function is a sequence consisting, in this order, of:

1. A list of result lists (one result list per line).
2. The list of values in the first line when the `header` option is set to `true`.

## 20) Clipping Functions

- The function `ld.clipseg(A, B, xmin, xmax, ymin, ymax)` clips the segment  $[A; B]$  (complex numbers) with the window  $[x_{\min}; x_{\max}] \times [y_{\min}; y_{\max}]$  and returns the result.
- The function `ld.clipline(d, xmin, xmax, ymin, ymax)` clips the line  $\langle d \rangle$  with the window  $[x_{\min}; x_{\max}] \times [y_{\min}; y_{\max}]$  and returns the result. The line  $\langle d \rangle$  is a list of two complex numbers: a point and a direction vector.
- The function `ld.clippolyline(L, xmin, xmax, ymin, ymax [, close])` clips the polygonal line  $\langle L \rangle$  with the window  $[x_{\min}; x_{\max}] \times [y_{\min}; y_{\max}]$  and returns the result. The argument  $\langle L \rangle$  is a list of complex numbers or a list of lists of complex numbers. The optional argument *close* (`false` by default) indicates whether the polygonal line should be closed.
- The function `ld.clipdots(L, xmin, xmax, ymin, ymax)` clips the point list (complex numbers)  $\langle L \rangle$  with the window  $[x_{\min}; x_{\max}] \times [y_{\min}; y_{\max}]$  and returns the result (exterior points are simply excluded). The argument  $\langle L \rangle$  is a list of complex numbers or a list of lists of complex numbers.

## 21) Adding Mathematical Functions

In addition to the functions associated with graphics methods that perform calculations and return a polygonal line (such as `ld.cartesian`, `ld.periodic`, `ld.implicit`, `ld.odesolve`, etc), the *luadraw* package adds some mathematical functions that are not natively provided in the *math* module.

### Protected Evaluation: evalf

The `ld.evalf(f, ...)` function allows you to evaluate `f(...)` and return the result if there is no runtime error in Lua; otherwise, the function returns `nil`. For example, executing:

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

causes the runtime error attempt to perform arithmetic on a nil value (in the console), because here  $Z(1, 1/0)$  returns `nil`, and Lua does not accept an argument equal to `nil` in a calculation. On the other hand, executing:

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

does not cause an error from Lua, and there is no output to the console either since the value to be displayed is `nil`.

## int

The function **ld.int(f, a, b)** returns an approximate value of the integral of the function  $\langle f \rangle$  over the interval  $[a; b]$  (two complex numbers). The function  $\langle f \rangle$  is a real variable and has real or complex values. The method used is Simpson's method, accelerated twice with the Romberg method.

### Example:

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

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

## gcd

The function **ld.gcd(a, b)** returns the greatest common divisor between  $\langle a \rangle$  and  $\langle b \rangle$ , two integers.

## lcm

The function **ld.lcm(a, b)** returns the smallest positive common divisor between  $\langle a \rangle$  and  $\langle b \rangle$ , two integers.

## nth\_root

The function **ld.nth\_root(n, x)** returns the  $n$ th root of the real number  $\langle x \rangle$ , the argument  $\langle n \rangle$  must be an integer.

## solve

The function **ld.solve(f, a, b [, n, df])** numerically solves the equation  $f(x) = 0$  in the interval  $[a; b]$ , the argument  $\langle f \rangle$  is the function,  $\langle a \rangle$ ,  $\langle b \rangle$  are two complex numbers. The interval is subdivided into  $n$  pieces (25 by default). The interval is subdivided into  $n$  pieces (25 by default). The optional argument  $\langle df \rangle$  is a function that represents the derivative of  $\langle f \rangle$ ; when this argument is absent, an approximation of the derivative is used. The function returns a list of results or `nil`. The method used is a variant of Newton's method. The function returns a list of numbers or `nil`.

### Example 1:

```
\begin{luacode}
local ld = luadraw
resol = function(f,a,b)
    local y = ld.solve(f,a,b)
    if y == nil then tex.sprint("\emptyset")
    else
        local str = y[1]
        for k = 2, #y do
            str = str..", "..y[k]
        end
        tex.sprint(str)
    end
end
\end{luacode}
\def\solve#1#2#3{\directlua{resol(#1,#2,#3)}}%
\begin{luacode}
```

```
f1 = function(x) return math.cos(x)-x end
f2 = function(x) return x^3-2*x^2+1/2 end
\end{luacode}
Solving the equation  $\cos(x)=x$  in  $[0;\frac{\pi}{2}]$  gives  $\text{\texttt{\textbackslashsolve{f1}{0}{math.pi/2}}}$ .
Solving the equation  $\cos(x)=x$  in  $[\frac{\pi}{2};\pi]$  gives  $\text{\texttt{\textbackslashsolve{f1}{math.pi/2}{math.pi}}}$ .
Solving the equation  $x^3-2x^2+\frac{1}{2}=0$  in  $[-1;2]$  gives:  $\text{\texttt{\textbackslashsolve{f2}{-1}{2}}}$ .
```

**Result:**

Solving the equation  $\cos(x) = x$  in  $[0; \frac{\pi}{2}]$  gives 0.73908513321516.

Solving the equation  $\cos(x) = x$  in  $[\frac{\pi}{2}; \pi]$  gives  $\emptyset$ .

Solving the equation  $x^3 - 2x^2 + \frac{1}{2} = 0$  in  $[-1; 2]$  gives:  $\{-0.45160596295578, 0.59696828323732, 1.8546376797185\}$ .

**Example 2:** We want to plot the curve of the function  $f$  defined by the condition:

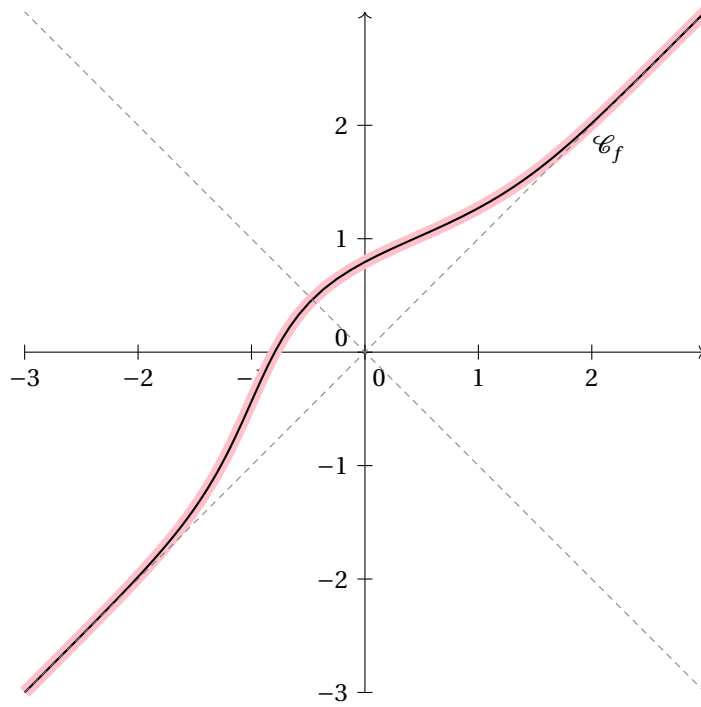
$$\forall x \in \mathbf{R}, \int_x^{f(x)} \exp(t^2) dt = 1.$$

We have two possible methods:

1. We consider the function  $G: (x, y) \mapsto \int_x^y \exp(t^2) dt - 1$ , and we draw the implicit curve with equation  $G(x, y) = 0$ .
2. We determine a real number  $y_0$  such that  $\int_0^{y_0} \exp(t^2) dt = 1$  and we draw the solution to the differential equation  $y' = e^{x^2 - y^2}$  satisfying the initial condition  $y(0) = y_0$ .

Let's draw both:

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

Figure 32: Function  $f$  defined by  $\int_x^{f(x)} \exp(t^2) dt = 1$ .

We see that the two curves overlap well, however the first method (implicit curve) is much more computationally intensive, so method 2 is preferable.

## V Transformations

In the following:

- the argument  $\langle L \rangle$  is either a complex number, a list of complex numbers, or a list of lists of complex numbers,
- the line  $\langle d \rangle$  is a list of two complex numbers: a point on the line and a direction vector.

### 1) **affin**

The function **ld.affin**(**L**, **d**, **v**, **k**) returns the image of  $\langle L \rangle$  by the affinity of base line  $\langle d \rangle$ , parallel to the vector  $\langle v \rangle$  and of ratio  $\langle k \rangle$ .

### 2) **ftransform**

The function **ld.ftransform**(**L**, **f**) returns the image of  $\langle L \rangle$  by the function  $\langle f \rangle$ , which must be a function of the complex variable. If one of the elements of  $\langle L \rangle$  is the complex number `cpx . Jump`, then it is returned as is in the result.

### 3) **hom**

The function **ld.hom**(**L**, **factor** [, **center**]) returns the image of  $\langle L \rangle$  by the homothety with center  $\langle center \rangle$  and ratio  $\langle factor \rangle$ . By default, the argument  $\langle center \rangle$  is 0.

### 4) **inv**

The function **ld.inv**(**L**, **radius** [, **center**]) returns the image of  $\langle L \rangle$  by the inversion with respect to the circle with center  $\langle center \rangle$  and radius  $\langle radius \rangle$ . By default, the argument  $\langle center \rangle$  is 0.

### 5) **proj**

The function **ld.proj**(**L**, **d**) returns the image of  $\langle L \rangle$  by the orthogonal projection onto the line  $\langle d \rangle$ .

## 6) projO

The function **ld.projO(L, d, v)** returns the image of  $\langle L \rangle$  by projection onto the line  $\langle d \rangle$  parallel to the vector  $\langle v \rangle$ .

## 7) rotate

The function **ld.rotate(L, angle [, center])** returns the image of  $\langle L \rangle$  by rotation with center *center* and angle  $\langle angle \rangle$  (in degrees). By default, the argument  $\langle center \rangle$  is 0.

## 8) shift

The function **ld.shift(L, u)** returns the image of  $\langle L \rangle$  by translation of vector  $\langle u \rangle$ .

## 9) simil

The function **ld.simil(L, factor, angle [, center])** returns the image of  $\langle L \rangle$  by the similarity of center  $\langle center \rangle$ , ratio  $\langle factor \rangle$ , and angle  $\langle angle \rangle$  (in degrees). By default, the argument *center* is 0.

## 10) sym

The function **ld.sym(L, d)** returns the image of  $\langle L \rangle$  by the orthogonal symmetry of axis  $\langle d \rangle$ .

## 11) symG

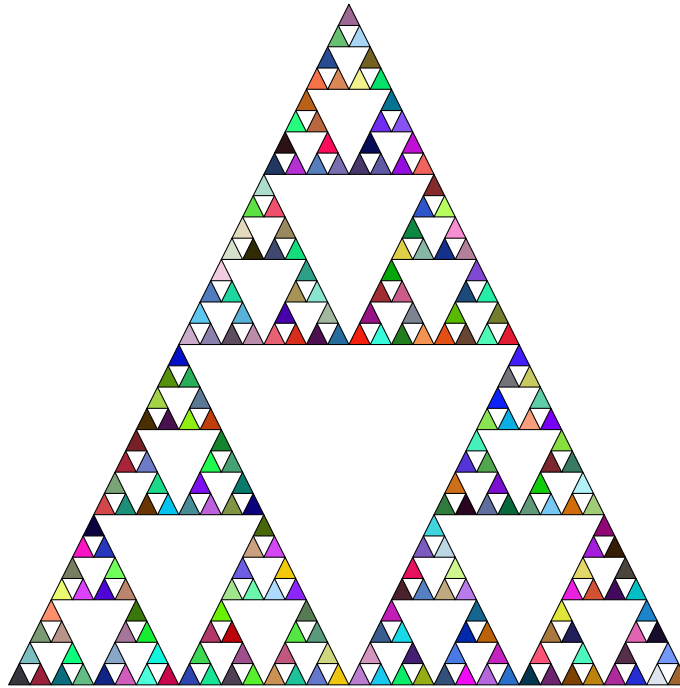
The function **ld.symG(L, d, v)** returns the image of  $\langle L \rangle$  by the symmetry about the line  $\langle d \rangle$  followed by the translation of vector  $\langle v \rangle$  (sliding symmetry).

## 12) symO

The function **ld.symO(L, d)** returns the image of  $\langle L \rangle$  by symmetry with respect to the line  $\langle d \rangle$  and parallel to the vector  $\langle v \rangle$  (oblique symmetry).

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

Figure 33: Using Transformations



## VI Matrix Calculus

If  $f$  is an affine application of the complex plane, we will call the list (table) of  $f$  the matrix:

$$\{f(0), Lf(1), Lf(i)\}$$

where  $Lf$  denotes the linear part of  $f$  (we have  $Lf(1) = f(1) - f(0)$  and  $Lf(i) = f(i) - f(0)$ ). The identity matrix is the global variable denoted `ID` in the *luadraw* package; it simply corresponds to the list  $\{0, 1, i\}$ .

### 1) Matrix Calculations

#### `applymatrix` and `applyLmatrix`

- The function `ld.applymatrix(z, M)` applies the matrix  $\langle M \rangle$  to the complex  $\langle z \rangle$  and returns the result (which is equivalent to calculating  $f(z)$  if  $\langle M \rangle$  is the matrix of  $f$ ). When  $\langle z \rangle$  is the complex `cpx . Jump` then the result is `cpx . Jump`. When  $\langle z \rangle$  is a string then the function returns  $\langle z \rangle$ .
- The function `ld.applyLmatrix(z, M)` applies the linear part of the matrix  $\langle M \rangle$  to the complex  $\langle z \rangle$  and returns the result (which is equivalent to calculating  $Lf(z)$  if  $\langle M \rangle$  is the matrix of  $f$ ). When  $\langle z \rangle$  is the complex `cpx . Jump` then the result is `cpx . Jump`.

#### `composematrix`

The function `ld.composematrix(M1, M2)` performs the matrix product  $\langle M1 \rangle \times \langle M2 \rangle$  and returns the result.

#### `invmatrix`

The function `ld.invmatrix(M)` calculates and returns the inverse of the matrix  $\langle M \rangle$  when possible.

#### `matrixof`

- The function `ld.matrixof(f)` calculates and returns the matrix of  $\langle f \rangle$  (which must be an affine application of the complex plane).
- Example: `ld.matrixof(function(z) return ld.proj(z,0,Z(1,-1)) end)` returns `{0,Z(0.5,-0.5),Z(-0.5,0.5)}` (matrix of the orthogonal projection onto the second bisector).

**mtransform and mLtransform**

- The function **ld.mtransform(L, M)** applies the matrix  $\langle M \rangle$  to the list  $\langle L \rangle$  and returns the result.  $\langle L \rangle$  must be a list of complex numbers or a list of lists of complex numbers. If one of them is the complex `cpx.Jump` or a string, then it is returned as is.
- The function **ld.mLtransform(L, M)** applies the linear part of the matrix  $\langle M \rangle$  to the list  $\langle L \rangle$  and returns the result.  $\langle L \rangle$  must be a list of complex numbers or a list of lists of complex numbers. If one of them is the complex `cpx.Jump` or a string, then it is returned as is.

**2) Matrix associated with the graph**

When creating a graph in the *luadraw* environment, for example:

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

The created *g* object has a transformation matrix that is initially the identity. All graphics methods used automatically apply the graph's transformation matrix. This matrix is designated *g.matrix*, but to manipulate it, the following methods are available.

**g:Composematrix()**

The **g:Composematrix(M)** method multiplies the graph matrix *g* by the matrix  $\langle M \rangle$  (with  $\langle M \rangle$  on the right) and assigns the result to the graph matrix. The argument  $\langle M \rangle$  must therefore be a matrix.

**g:Det2d()**

The **g:Det2d()** method returns 1 when the transformation matrix has a positive determinant, and  $-1$  otherwise. This information is useful when we need to know whether the plane orientation has been changed or not.

**g:IDmatrix()**

The **g:IDmatrix()** method reassigns the identity to the graph matrix *g*.

**g:Mtransform()**

The **g:Mtransform(L)** method applies the graph matrix *g* to  $\langle L \rangle$  and returns the result. The argument  $\langle L \rangle$  must be a list of complex numbers, or a list of lists of complex numbers.

**g:MLtransform()**

The **g:MLtransform(L)** method applies the linear part of the graph matrix *g* to  $\langle L \rangle$  and returns the result. The argument  $\langle L \rangle$  must be a list of complex numbers, or a list of lists of complex numbers.

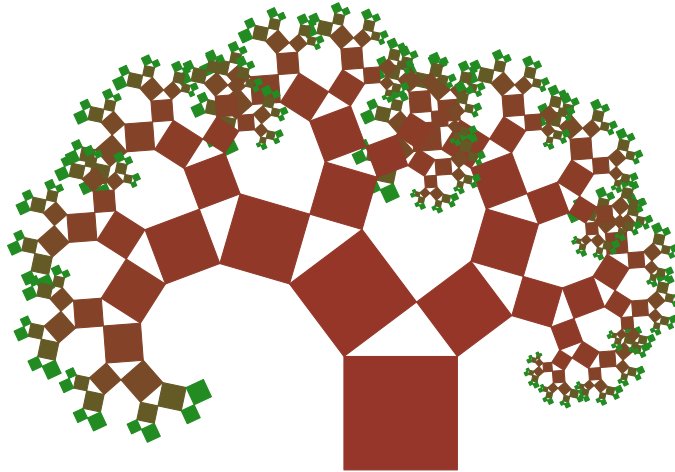
```
\begin{luadraw}{name=Pythagore}
local ld = luadraw
local g = ld.graph:new{window={-15,15,0,22},size={10,10}}
local a, b, c = 3, 4, 5 -- un triplet de Pythagore
local i, arccos, exp = ld.cpx.I, math.acos, ld.cpx.exp
local f1 = function(z)
    return (z-c)*a/c*exp(-i*arccos(a/c))+c+i*c end
local M1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

```

    g:Savematrix(); g:Composematrix(M1); arbre(n-1)
    g:Restorematrix(); g:Savematrix(); g:Composematrix(M2)
    arbre(n-1); g:Restorematrix()
  end
end
arbre(8)
g:Show()
\end{luadraw}

```

Figure 34: Using the Graph Matrix

**g:Rotate()**

The **g:Rotate(*angle* [, *center*])** method modifies the transformation matrix of the graph *g* by composing it with the rotation matrix with angle *angle* (in degrees) and center *center*. The argument *center* is a complex matrix that defaults to 0.

**g:Scale()**

The **g:Scale(*factor* [, *center*])** method modifies the transformation matrix of the graph *g* by composing it with the homothety matrix with ratio *factor* and center *center*. The argument *center* is a complex matrix that defaults to 0.

**g:Savematrix() and g:Restorematrix()**

- The **g:Savematrix()** method saves the transformation matrix of graph *g* to a stack.
- The **g:Restorematrix()** method restores the transformation matrix of graph *g* to its last saved value.

**g:Setmatrix()**

The **g:Setmatrix(*M*)** method assigns matrix *M* to the transformation matrix of graph *g*.

**g:Shift()**

The **g:Shift(*v*)** method modifies the transformation matrix of graph *g* by composing it with the translation matrix of vector *v*, which must be a complex matrix.

```

\begin{luadraw}{name=free_art}
local ld = luadraw
local du = math.sqrt(2)/2
local g = ld.graph:new{window={1-du,4+du,1-du,4+du},
    margin={0,0,0,0},size={7,7}}
local i = ld.cpx.I
g:Linestyle("noline")
g:Filloptions("full","Navy",0.1)

```

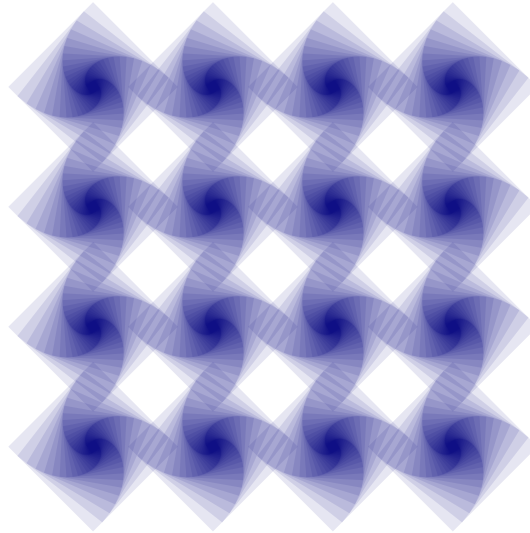


```

for X = 1, 4 do
  for Y = 1, 4 do
    g:Saveatrix()
    g:Shift(X+i*Y); g:Rotate(45)
    for k = 1, 25 do
      g:Dsquare((1-i)/2, (1+i)/2, 1)
      g:Rotate(7); g:Scale(0.9)
    end
    g:Restoreatrix()
  end
end
g:Show()
\end{luadraw}

```

Figure 35: Using Shift, Rotate and Scale



### 3) View change. Change of coordinate system

**View change:** when creating a new graph, for example:

```

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

```

The option `window={xmin,xmax,ymin,ymax}` sets the view for graph `g`. This will be the  $[x_{\min}; x_{\max}] \times [y_{\min}; y_{\max}]$  block of  $\mathbf{R}^2$ , and all plots will be clipped by this window (except labels, which can overflow into the margins, but not beyond). Within this box, it is possible to define another box to create a new view, using the `g:Viewport(x1, x2, y1, y2)` method. The values of  $\langle x1 \rangle, \langle x2 \rangle, \langle y1 \rangle, \langle y2 \rangle$  refer to the initial window defined by the `window` option. From then on, everything outside this new area will be clipped, and the graph matrix will be reset to the same value. Therefore, you must first save the current graphics settings:

```

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

```

To return to the previous view with the previous matrix, simply restore the graphics settings with the `g:Restoreattr()` method.

**Warning:** Each `g:Saveattr()` instruction must correspond to a `g:Restoreattr()` instruction, otherwise a compilation error will occur.

**Changing the coordinate system:** The coordinate system of the current view can be changed with `g:Coordsystem(x1, x2, y1, y2 [, ortho])`.

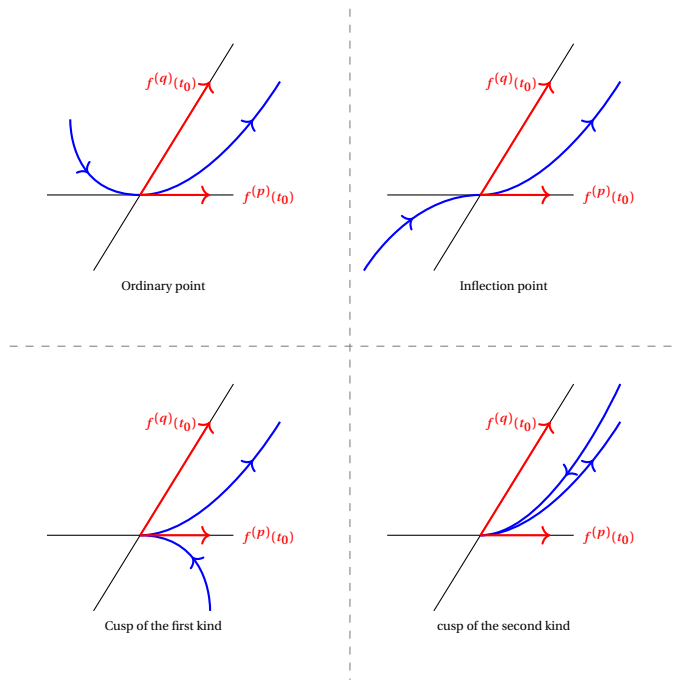
This method will modify the graph matrix so that everything occurs as if the current view corresponded to the  $[x_1; x_2] \times [y_1; y_2]$  box. The optional Boolean argument  $\langle ortho \rangle$  indicates whether the new coordinate system should be orthonormal or not (`false` by default). Since the graph matrix is modified, it is best to save the graphical parameters first and restore them later. This can be used, for example, to create multiple figures in the current graph.

```

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

```

Figure 36: Classification of the points of a parametric curve



## VII Adding Your Own Methods

Without having to modify the Lua source files associated with the *luadraw* package, you can add your own methods to the *ld.graph* class, or modify an existing method. This is only useful if these modifications will be used in different graphs and/or different documents (otherwise, you can simply write a function locally in the graph where it's needed).

## 1) An Example

In the graph on page 17, we drew a vector field. To do this, we wrote a function that calculates the vectors before drawing, but this function is local. We could make it a global function in the namespace *luadraw*, which would then be usable throughout the document, but not in another document!

To generalize this function, we will need to create a Lua file that can then be imported into documents if necessary. To make the example a bit more consistent, we'll create a file that defines a function that calculates the vectors of a field, and that will add two new methods to the *ld.graph* class: one to draw a vector field of a function  $f: (x, y) \rightarrow f(x, y) \in \mathbf{R}^2$ , we'll name it **ld.graph:Dvectorfield**, and another to draw a gradient field of a function  $f: (x, y) \rightarrow f(x, y) \in \mathbf{R}$ , we'll name it **ld.graph:Dgradientfield**. Therefore, we'll call this file: *luadraw\_fields.lua*.

### File contents:

```
-- luadraw_fields.lua
-- added methods to the graph class of the luadraw package
-- to draw vector or gradient fields

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

function ld.field(f,x1,x2,y1,y2,grid,long) -- mathematical function, independent of the graph
-- calculates a vector field in the tile [x1,x2]x[y1,y2]
-- f function of two variables with values in R^2
-- grid = {nbx, nby} : number of vectors along x and along y
-- long = length of a vector
    if grid == nil then grid = {25,25} end
    local deltax, deltax = (x2-x1)/(grid[1]-1), (y2-y1)/(grid[2]-1) -- x and y step
    if long == nil then long = math.min(deltax,deltay) end -- default length
    local vectors = {} -- will contain the list of vectors
    local x, y, v = x1
    for _ = 1, grid[1] do -- a loop on x
        y = y1
        for _ = 1, grid[2] do -- a loop on y
            v = f(x,y) -- we assume that v is well defined
            v = Z(v[1],v[2]) -- to complex number
            v = cpx.normalize(v)
            if v ~= nil then
                table.insert(vectors, {Z(x,y)-long/2*v, Z(x,y)+long/2*v} ) -- on ajoute le vecteur
            end
            y = y+deltay
        end
        x = x+deltax
    end
    return vectors -- we return the result (polygonal line)
end

function graph:Dvectorfield(f,args) -- added a method to the graph class
-- draws a vector field
-- f is a function of two variables with values in R^2
-- args is a 4-field table:
-- { view={x1,x2,y1,y2}, grid={nbx,nby}, long=, draw_options="" }
    args = args or {}
    local view = args.view or {self:Xinf(),self:Xsup(),self:Yinf(),self:Ysup()} -- default user reference
    local vectors = ld.field(f,view[1],view[2],view[3],view[4],args.grid,args.long) -- field calculation
    self:Dpolyline(vectors,false,args.draw_options) -- the drawing (non-closed polygonal line)
end

function graph:Dgradientfield(f,args) -- added another method to the graph class
-- draws a gradient field
-- f is a function of two variables with values in R
-- args is a 4-field table:
-- { view={x1,x2,y1,y2}, grid={nbx,nby}, long=, draw_options="" }
```

```

local h = 1e-6
local grad_f = function(x,y) -- gradient function of f
    return { (f(x+h,y)-f(x-h,y))/(2*h), (f(x,y+h)-f(x,y-h))/(2*h) }
end
self:Dvectorfield(grad_f,args) -- we use the previous method
end

```

## 2) How to import the file

There are two methods for this:

1. With the Lua instruction *dofile*. This can be written, for example, in the preamble after the package declaration:

```

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

```

Of course, you will need to replace `<path>` with the path to this file.

The instruction `\directlua{dofile("<path>/luadraw_fields.lua")}` can be placed elsewhere in the document, as long as it is after the package has been loaded (otherwise the *graph* class will not be recognized when reading the file). We can also place the instruction `dofile("<path>/luadraw_fields.lua")` in a *luacode* environment, and therefore in particular in a *luadraw* environment.

As soon as the file is imported, the new methods are available for the rest of the document.

This approach has at least two drawbacks: you need to remember the path each time you use it, and secondly, the *dofile* instruction does not check whether the file has already been read. For these reasons, we prefer the following method.

2. With the Lua instruction *require*. For example, we can write it in the preamble after the package declaration:

```

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

```

Note the absence of the path (and the lua extension is unnecessary).

The `\directlua{require "luadraw_fields"}` instruction can be placed elsewhere in the document, provided it is after the package has been loaded (otherwise the *graph* class will not be recognized when reading the file). We can also place the `require "luadraw_fields"` instruction in a *luacode* environment, and therefore in particular in a *luadraw* environment.

The *require* instruction checks whether the file has already been loaded or not, which is preferable. However, Lua must be able to find this file, and the easiest way to do this is for it to be somewhere in a tree structure known to TeX. For example, you can create the following path in your local *texmf*:

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

then copy the file *luadraw\_fields.lua* into the *myluafiles* folder.

```

\begin{luadraw}{name=fields}
require "luadraw_fields" -- import of new methods
local ld = luadraw
local g = ld.graph:new{window={0,21,0,10},size={16,10}}
local i = ld.cpx.I
g:Labelsize("footnotesize")
local f = function(x,y) return {x*x*y,-y*x*y} end -- Volterra
local F = function(x,y) return x^2+y^2+x*y-6 end
local H = function(t,Y) return f(Y[1],Y[2]) end
-- top
g:Saveattr();g:Viewport(0,10,0,10);g:Coordsystem(-5,5,-5,5)
g:Dgradbox({-4.5-4.5*i,4.5+4.5*i,1,1}, {originloc=0,originnum={0,0},grid=true,
    title="gradient field, $f(x,y)=x^2+y^2+xy-6$"})
g:Arrows("->"); g:Lineoptions(nil,"blue",6)
g:Dgradientfield(F,{view={-4,4,-4,4},grid={15,15},long=0.5})
g:Arrows("-"); g:Lineoptions(nil,"Crimson",12); g:Dimplicit(F, {view={-4,4,-4,4}})

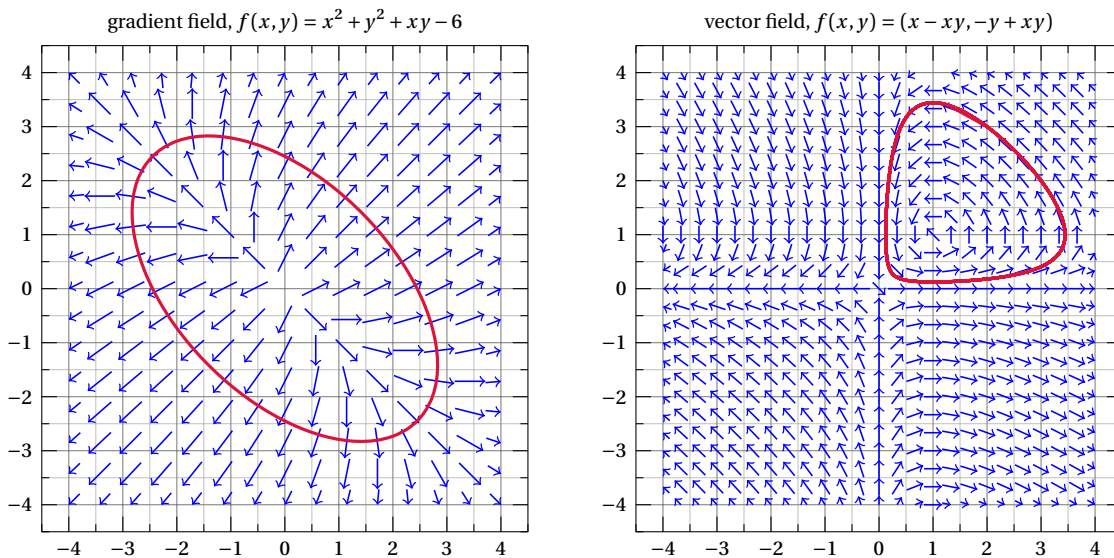
```

```

g:Restoreattr()
-- bottom
g:Saveattr();g:Viewport(11,21,0,10);g:Coordsystem(-5,5,-5,5)
g:Dgradbox({{-4.5-4.5*i,4.5+4.5*i,1,1}, {originloc=0,originnum={0,0},grid=true,
  title="vector field, $f(x,y)=(x-xy,-y+xy)$"})
g:Arrows("->"); g:Lineoptions(nil,"blue",6); g:Dvectorfield(f,{view={-4,4,-4,4}})
g:Arrows("-");g:Lineoptions(nil,"Crimson",12)
g:Dodesolve(H,0,{2,3},{t={0,50},out={2,3},nbdots=250})
g:Restoreattr()
g:Show()
\end{luadraw}

```

Figure 37: Using the new methods



### 3) Modifying an existing method

Let's take for example the method `g:DplotXY(X, Y [, draw_options])`, which takes two lists (tables) of real numbers as arguments and draws the polygonal line formed by the points with coordinates  $(X[k], Y[k])$ . We'll modify it to account for the case where  $\langle X \rangle$  is a list of names (strings). In this case, the names will be displayed below the x-axis (with x-axis  $k$  for the  $k$ th name) and the polygonal line formed by the points with coordinates  $(k, Y[k])$  will be drawn. Otherwise, we'll use the same method as the old method. To do this, simply rewrite the method (in a Lua file so that it can be imported later):

```

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

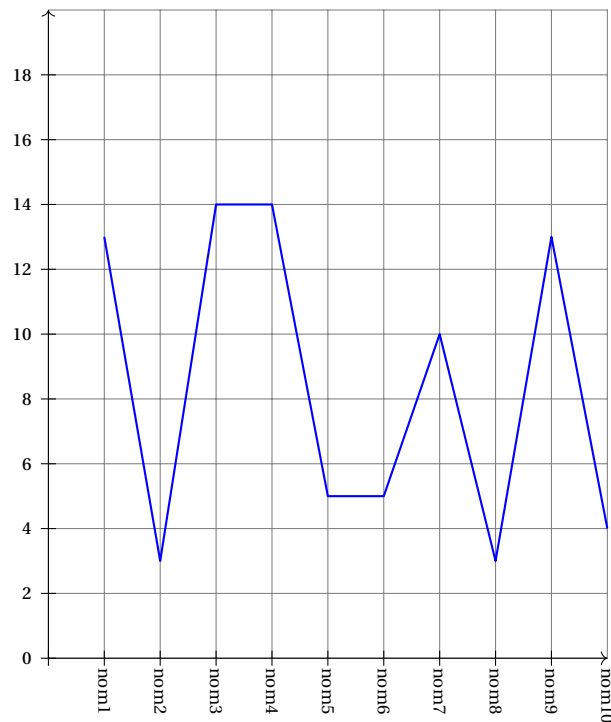
function ld.graph:DplotXY(X,Y,draw_options)
-- X is a list of real numbers or strings
-- Y is a list of real numbers of the same length as X
  local L = {} -- list of points to draw
  if type(X[1]) == "number" then -- list of real numbers
    for k,x in ipairs(X) do
      table.insert(L,Z(x,Y[k]))
    end
  else
    local noms = {} -- list of labels to place
    for k = 1, #X do
      table.insert(L,Z(k,Y[k]))
      insert(noms,{X[k],k,{pos="E",node_options="rotate=-90"}})
    end
    self:Dlabel(table.unpack(noms)) -- drawing labels
  end
  self:Dpolyline(L,draw_options) -- drawing the curve
end

```

As soon as the file is imported, this new definition will overwrite the old one (for the rest of the document). Of course, you could imagine adding other options to the drawing style, for example (lines, bars, dots, etc.).

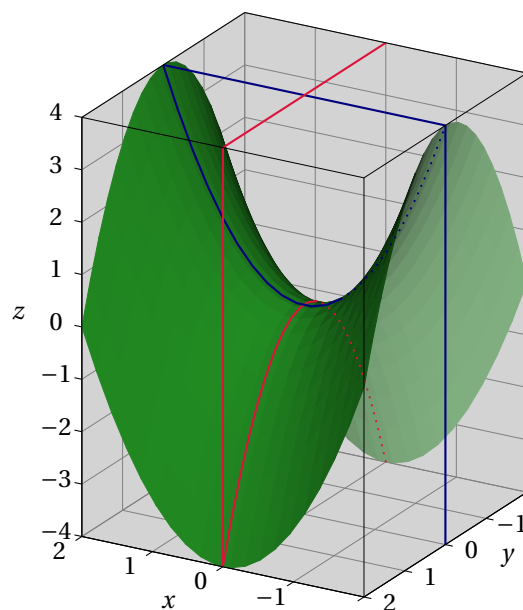
```
\begin{luadraw}{name=newDplotXY}
require "luadraw_fields" -- import of the modified method
local ld = luadraw
local g = ld.graph:new{window={-0.5,11,-1,20}, margin={0.5,0.5,0.5,1}, size={10,10,0}}
g:Labelsize("scriptsize")
local X, Y = {}, {} -- we define two lists X and Y, we could also read them in a file
for k = 1, 10 do
    table.insert(X,"nom"..k)
    table.insert(Y,math.random(1,20))
end
ld.defaultlabelshift = 0
g:Daxes({0,1,2},{limits={{0,10},{0,20}}, labelpos={"none","left"},arrows="->", grid=true})
g:DplotXY(X,Y,"line width=0.8pt, blue")
g:Show()
\end{luadraw}
```

Figure 38: Modifying an existing method



# 3D Drawing

Figure 1: Col point at  $M(0,0,0)$  ( $z = x^2 - y^2$ )



## I Introduction

As a reminder, we use the following shortcuts:

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

### 1) Prerequisites

- This chapter presents the use of the *luadraw* package with the *3d* global option: `\usepackage[3d]{luadraw}`.
- The package loads the *luadraw\_graph2d.lua* module, which defines the *ld.graph* class and provides the *luadraw* environment for creating graphs in Lua. Everything said in the previous chapter (Drawing 2D) therefore applies, and is assumed to be known here.
- The *3d* global option also allows the loading of the *luadraw\_graph3d.lua* module. This also defines the *ld.graph3d* class (which relies on the *ld.graph* class) for 3D drawings.

## 2) Some reminders

- Another global package option: `noexec`. When this global option is mentioned, the default value of the `exec` option for the *luadraw* environment will be `false` (and no longer `true`).
- When a graph is finished, it is exported in TikZ format, so this package also loads the TikZ package and the libraries:
  - *patterns*
  - *plotmarks*
  - *arrows.meta*
  - *decorations.markings*
- Graphs are created in a *luadraw* environment, which calls *luacode*, so the **Lua language** must be used in this environment.
- Saving the *.tkz* file: the chart is exported in TikZ format to a file (with the *tkz* extension). By default, it is saved in the *\_luadraw* folder, which is a subfolder of the current folder (containing the master document), but it is possible to specify a path to another subfolder, with the global option `cachedir=...`.
- The environment options are:
  - `name=...`: allows you to name the resulting TikZ file. It is given a name without an extension (the extension will be automatically added; it is *.tkz*). If this option is omitted, then a default name is used, which is the name of the master file followed by a number.
  - `exec=true/false`: allows you to execute or not the Lua code included in the environment. By default, this option is true, **EXCEPT** if the global option `noexec` was mentioned in the preamble with the package declaration. When a complex graph that requires a lot of calculations is developed, it may be useful to add the option `exec=false`; this will avoid recalculating the same graph for future compilations.
  - `auto=true/false`: allows you to automatically include or not the TikZ file in place of the *luadraw* environment when the `exec` option is `false`. By default, the `auto` option is `true`.

## 3) Creating a 3D Graph

```
\begin{luadraw}{ name=<filename>, exec=true/false, auto=true/false }
local ld = luadraw
-- create a new graph and give it a local name
local g = ld.graph3d:new{ window3d={x1,x2,y1,y2,z1,z2 [,xscale,yscale,zscale]}, adjust2d=true/false, viewdir={30,60},
  ↳ window={x1,x2,y1,y2 [,xscale,yscale]}, margin={top,right,bottom,left}, size={width,height,ratio}, bg="color",
  ↳ border=true/false }
-- build graph g
graph instructions in Lua language ...
-- display graph g and save it in the file <filename>.tkz
g:Show()
-- or Save only in the <filename>.tkz file
g:Save()
\end{luadraw}
```

**Important:** throughout the rest of this chapter, points in space are called *3D points*, they are triplets of  $\mathbf{R}^3$ . In the *luadraw* environment, the 3D point  $(x, y, z)$  will be denoted **pt3d.M(x, y, z)** (**pt3d.M** is a function that creates and returns a 3D point).

Creation is done in a *luadraw* environment. This creation is done on the first line inside the environment by naming the graph:

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

The *graph3d* class is defined in the *luadraw* package using the global option `3d`. This class is instantiated by invoking its constructor and giving it a name (here it's *g*). This is done locally so that the graph *g* thus created will no longer exist once it leaves the environment (otherwise *g* would remain in memory until the end of the document).



- The option `window3d={x1,x2,y1,y2,z1,z2 [,xscale,yscale,zscale]}` defines the  $\mathbf{R}^3$  parallelepiped corresponding to the graph: it is  $[x_1;x_2] \times [y_1;y_2] \times [z_1;z_2]$ , as well as the scale on the three axes: *xscale*, *yscale*, and *zscale*, these are optional and default to 1. The default parallelepiped is  $[-5;5] \times [-5;5] \times [-5;5]$ .

**Caution:** The three scales determine the initial 3D matrix of the graph. When one of them is not 1, this matrix is not the identity matrix. If you need to change the graph matrix later, you must use the method `g:Composematrix3d()` and not `g:Setmatrix3d()`, and remember to save the initial matrix beforehand with the method `g:Savematrix()`, you can then restore it with the method `g:Restorematrix()`.

- The option `adjust2d` indicates whether the 2D window that will contain the projection of the 3D drawing should be determined automatically (`false` by default). This 2D window corresponds to the `window` option.
- The option `viewdir` is a table that defines the projection method and the two viewing angles (in degrees). By default `viewdir={"ortho","30",60}` (orthographic projection). The following figure shows what these two angles correspond to.

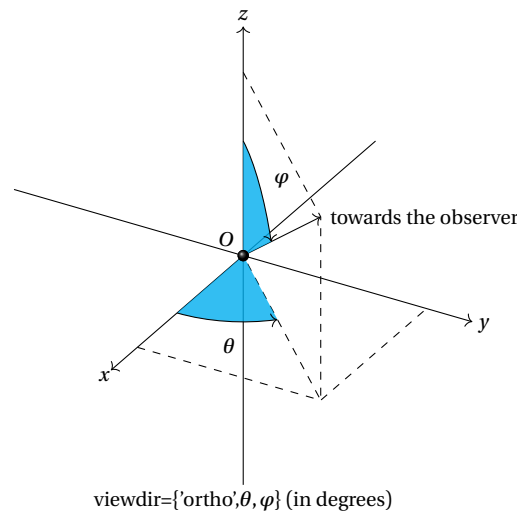


Figure 2: Viewing Angles

- The other options are those of the `ld.graph` class, described in Chapter 1.

#### Graph construction.

- The instantiated object (*g* in the example) has all the methods of the `ld.graph` class, plus methods specific to 3D.
- The `ld.graph3d` class also provides a number of mathematical functions specific to 3D.

## 4) Affine Projection Modes

- Orthogonal Projections:
  - `viewdir={"ortho",theta,phi}`, or `viewdir={theta,phi}`: orthographic projection (orthogonal projection onto the screen), this is the default projection with `theta=30` and `phi=60` (degrees),
  - `viewdir="xOy"`: orthogonal projection onto the *xy* plane,
  - `viewdir="xOz"`: orthogonal projection onto the *xz* plane,
  - `viewdir="yOz"`: orthogonal projection onto the *yz* plane.
- Non-orthogonal Projections:
  - `viewdir={"yz",k,alpha}`: cavalier perspective on the *yz* plane,
  - `viewdir={"xz",k,alpha}`: cavalier perspective on the *xz* plane,

- `viewdir={"xy",k,alpha}`: cavalier perspective on the  $xy$  plane,
- `viewdir="iso"`: isometric perspective.

The three cavalier perspectives are defined using two parameters: a positive number  $k$  and an angle in degrees  $\alpha$ , which are highlighted in the following figure.

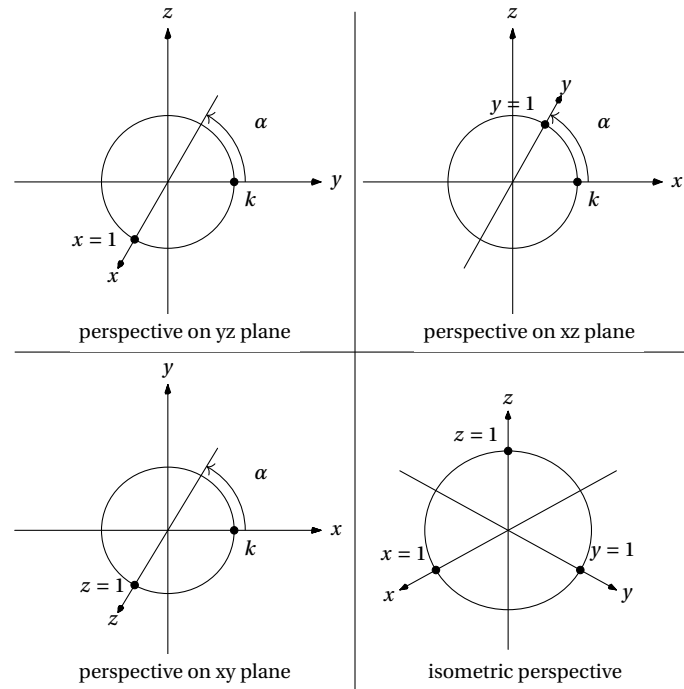


Figure 3: Affine Projection Modes

All values that can be passed to the `viewdir` option during creation can also be used in the `g:Setviewdir()` method during graphing.

## 5) Central Projection

Since version 2.4, *luadraw* also offers central projection. Unlike previous modes, **this projection is not affine**, and furthermore, it is not defined for all points in space, which can lead to errors, thus requiring thought and adjustments. This projection is defined by:

- A camera, which is a point in space stored in a global variable called `ld.camera` and which should not be modified directly.
- A target, which is a point in space stored in a global variable called `ld.target` and which should not be modified directly.

The plane passing through the `ld.target` and orthogonal to the `ld.target` - `ld.camera` axis is the projection plane; it represents the screen. As with the previous modes, the central projection is accessed via the `viewdir` option, or the `g:Setviewdir` method:

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

or

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

In the first case, the values of  $\langle camera \rangle$  and  $\langle target \rangle$  are given (3D points; by default,  $\langle target \rangle$  is the origin). In the second case, the three arguments  $\langle theta \rangle$ ,  $\langle phi \rangle$ , and  $\langle d \rangle$  are used to calculate the position of the camera according to the following diagram:

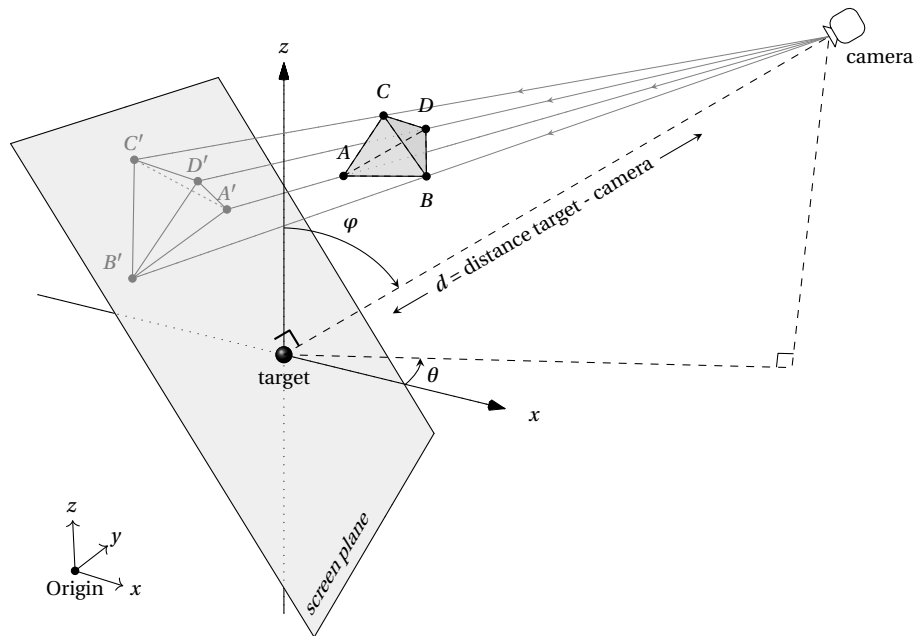


Figure 4: Central projection

The default values are:  $\langle \theta \rangle = 30$  (degrees),  $\langle \phi \rangle = 60$ ,  $\langle d \rangle = 15$ ,  $\langle \text{target} \rangle = \text{pt3d.Origin}$  (the variable `pt3d.Origin` is the 3D point  $(0, 0, 0)$ ).

## II The pt3d Class

### 1) Representation of Points and Vectors

The usual space is  $\mathbf{R}^3$ , so points and vectors are triplets of real numbers, we will call them: 3D points. The class `pt3d` (which is automatically loaded) allows you to manage 3D points, possible operations, and a number of methods and constants.

- Four triplets have specific names (global variables), namely:
  - `pt3d.Origin`, which represents the triplet  $(0, 0, 0)$ .
  - `pt3d.vecI`, which represents the triplet  $(1, 0, 0)$ .
  - `pt3d.vecJ`, which represents the triplet  $(0, 1, 0)$ .
  - `pt3d.vecK`, which represents the triplet  $(0, 0, 1)$ .
- To create a 3D point, there are three methods:
  - Cartesian definition: the function `pt3d.M(x, y, z)` returns the triplet  $(x, y, z)$ . This triplet can also be obtained by doing:  $\mathbf{x} * \text{vecI} + \mathbf{y} * \text{vecJ} + \mathbf{z} * \text{vecK}$ .
  - Cylindrical definition: the function `pt3d.Mc(r, theta, z)` (angle expressed in radians) returns the triplet  $(r \cos(\theta), r \sin(\theta), z)$ .
  - Spherical definition: the function `pt3d.Ms(r, theta, varphi)` returns the triplet  $(r \cos(\theta) \sin(\varphi), r \sin(\theta) \sin(\varphi), r \cos(\varphi))$  (angles expressed in radians).

Accessing the components of a 3D point: if a variable  $A$  denotes a 3D point, then its three components are  $A.x$ ,  $A.y$ , and  $A.z$ .

To test whether a variable  $A$  designates a 3D point, we use the `pt3d.isPoint3d()` function, which returns a Boolean.

Conversion: To convert a real or complex number into a 3D point, we use the `pt3d.toPoint3d()` function.

### 2) Operations on 3D Points

These operations are the usual operations with the usual symbols:

- Addition (+), difference (-), and negative (-).

- The product by a scalar, if  $k$  is a real number,  $k * M(x, y, z)$  returns  $M(k * x, k * y, k * z)$ .
- A 3D point can be divided by a scalar; for example, if  $A$  and  $B$  are two 3D points, then the midpoint is simply written  $(A + B) / 2$ .
- The equality of two 3D points can be tested with the symbol  $=$ .

### 3) Methods of the class *pt3d*

These are:

- **pt3d.abs(u)**: Returns the Euclidean norm of the 3D point  $\langle u \rangle$ .
- **pt3d.abs2(u)**: Returns the squared Euclidean norm of the 3D point  $\langle u \rangle$ .
- **pt3d.N1(u)**: Returns the 1-norm of the 3D point  $\langle u \rangle$ . If  $u = M(x, y, z)$ , then **pt3d.N1(u)** returns  $|x| + |y| + |z|$ .
- **pt3d.dot(u, v)**: Returns the dot product between the vectors (3D points)  $\langle u \rangle$  and  $\langle v \rangle$ .
- **pt3d.det(u, v, w)**: Returns the determinant between the vectors (3D points)  $\langle u \rangle$ ,  $\langle v \rangle$ , and  $\langle w \rangle$ .
- **pt3d.prod(u, v)**: Returns the cross product between the vectors (3D points)  $\langle u \rangle$  and  $\langle v \rangle$ .
- **pt3d.angle3d(u, v [, epsilon])**: Returns the angular difference (in radians) between the vectors (3D points)  $\langle u \rangle$  and  $\langle v \rangle$ , assumed to be non-zero. The (optional) argument  $\langle epsilon \rangle$  is 0 by default; it indicates how close a given equality test is to a floating point.
- **pt3d.normalize(u)**: Returns the normalized vector (3D point)  $\langle u \rangle$  (returns **nil** if  $u$  is zero).
- **pt3d.round(u, nbDeci)**: Returns a 3D point whose components are those of the 3D point  $\langle u \rangle$  rounded to  $\langle nbDeci \rangle$  decimal places.
- **pt3d.isobar3d(L)**: Returns the isobarycenter of the 3D points in the list (table)  $\langle L \rangle$  (elements of  $\langle u \rangle$  that are not 3D points are ignored).
- **pt3d.insert3d(L, A [, epsilon])**: This function inserts the 3D point  $\langle A \rangle$  into the list  $\langle u \rangle L$ , which must be a **variable** (and which will therefore be modified). Point  $\langle A \rangle$  is inserted **without duplicates** and the function returns its position (index) in list  $\langle L \rangle$  after insertion. The (optional) argument  $\langle epsilon \rangle$  is 0 by default, indicating how closely the comparisons are made.

### 4) Displaying a Variable in the Terminal

The instruction **ld.whatis(variable [, msg])** displays the type of the  $\langle variable \rangle$  and its contents in the terminal during compilation. Recognized types include the predefined types plus: *complex number*, *list of (complex) numbers*, and *list of lists of (complex) numbers*, *3D point*, *list of 3D points*, *list of lists of 3D points*. The argument  $\langle msg \rangle$  is an optional string (empty by default) which is displayed with the type to locate the variable in the terminal.

## III Graphics Methods

All 2D graphics methods apply. Added to this is the ability to draw polygonal lines, segments, straight lines, curves, paths, points, labels, planes, and solids in space. With solids, we also have the concept of facets, which was not found in 2D.

3D graphics methods will automatically calculate the projection onto the screen plane. After applying the 3D transformation matrix associated with the graphic (which is the default identity) to the objects, the 2D graphics methods will then take over.

The method that applies the 3D matrix and performs the projection onto the screen (plane passing through the origin and normal to the unit vector directed towards the observer and defined by the viewing angles) is: **g:Proj3d(L)** where  $\langle L \rangle$  is either a 3D point, a list of 3D points, or a list of lists of 3D points. This function returns complex numbers (affixes of the projected points onto the screen).

**Warning:** when the 3D matrix of the graph is an affine transformation but not linear, the projection onto the screen of a vector  $u$  in space is not **g:Proj3d(u)**, but **g:Proj3d(A+u)-g:Proj3d(A)** where  $A$  denotes any point in space. To

avoid these calculations, the method **g:Proj3dV()** has been introduced, it projects the **vectors** onto the screen, and returns complex numbers (affixes of the projected vectors onto the screen).

## 1) Line Drawing

### Polygonal Line: Dpolyline3d

The method **g:Dpolyline3d(L [, close, draw\_options, clip])** (where *g* denotes the graph being created),  $\langle L \rangle$  is a 3D polygonal line (list of 3D point lists),  $\langle close \rangle$  is an optional argument that is **true** or **false** indicating whether the line should be closed or not (**false** by default), and  $\langle draw\_options \rangle$  is a string that will be passed directly to the *\draw* instruction in the export. The  $\langle clip \rangle$  argument is set to **false** by default, it indicates whether the line  $\langle L \rangle$  should be clipped to the current 3D window.

### Right Angle: Dangle3d

The **g:Dangle3d(B, A, C [, r, draw\_options, clip])** method draws the angle *BAC* with a parallelogram (only two sides are drawn). The optional argument  $\langle r \rangle$  specifies the length of one side (0.25 by default). The parallelogram is in the plane defined by points  $\langle A \rangle$ ,  $\langle B \rangle$ , and  $\langle C \rangle$ , so they should not be aligned. The  $\langle draw\_options \rangle$  argument is a string (empty by default) that will be passed as is to the *\draw* instruction. The  $\langle clip \rangle$  argument is set to **false** by default; it indicates whether the plot should be clipped to the current 3D window.

### Segment: Dseg3d

The **g:Dseg3d(seg [, scale, draw\_options, clip])** method draws the segment defined by the  $\langle seg \rangle$  argument, which must be a list of two 3D points. The optional  $\langle scale \rangle$  argument (1 by default) is a number that allows you to increase or decrease the length of the segment (the natural length is multiplied by  $\langle scale \rangle$ ). The  $\langle draw\_options \rangle$  argument is a string (empty by default) that will be passed as is to the *\draw* instruction. The  $\langle clip \rangle$  argument is set to **false** by default; it indicates whether the plot should be clipped to the current 3D window.

### Line: Dline3d

The method **g:Dline3d(d [, draw\_options, clip])** draws the line  $\langle d \rangle$ , which is a list of type  $\langle d \rangle = \{A, u\}$  where *A* represents a point on the line (3D point) and *u* a direction vector (a non-zero 3D point).

Variant: the method **g:Dline3d(A, B [, draw\_options, clip])** draws the line passing through the points  $\langle A \rangle$  and  $\langle B \rangle$  (two 3D points). The  $\langle draw\_options \rangle$  argument is a string (empty by default) that will be passed as is to the *\draw* instruction. The  $\langle clip \rangle$  argument is set to **false** by default; it indicates whether the plot should be clipped to the current 3D window.

The **g:Line3d2seg(d [, scale])** method returns a table consisting of two 3D points representing a segment. This segment is the portion of the line  $\langle d \rangle$  inside the current 3D window. The  $\langle scale \rangle$  argument (1 by default) allows you to vary the size of this segment. When the window is too small, the intersection may be empty.

### Circular arc: Darc3d

- The method **g:Darc3d(B, A, C, r, sens [, normal, draw\_options, clip])** draws a circular arc with center  $\langle A \rangle$  (3D point), radius  $\langle r \rangle$ , going from  $\langle B \rangle$  (3D point) to  $\langle C \rangle$  (3D point) in the forward direction if the argument  $\langle sens \rangle$  is 1, and in the reverse direction otherwise. This arc is drawn in the plane containing the three points  $\langle A \rangle$ ,  $\langle B \rangle$ , and  $\langle C \rangle$ . When these three points are aligned, the argument  $\langle normal \rangle$  (non-zero 3D point) must be specified, which represents a vector normal to the plane. This plane is oriented by the vector product  $\vec{AB} \wedge \vec{AC}$  or by the vector  $\langle normal \rangle$  if it is specified. The  $\langle draw\_options \rangle$  argument is a string (empty by default) that will be passed as is to the *\draw* instruction. The  $\langle clip \rangle$  argument is set to **false** by default; it indicates whether the plot should be clipped to the current 3D window.
- The **ld.arc3d(B, A, C, r, sens [, normal])** function returns the list of points of this arc (3D polygonal line).
- The **ld.arc3db(B, A, C, r, sens [, normal])** function returns this arc as a 3D path ((see *Dpath3d* page 86)) using Bézier curves.

**Circle: Dcircle3d**

- The method **g:Dcircle3d(I, R, normal [, draw\_options, clip])** draws the circle with center  $\langle I \rangle$  (3D point) and radius  $\langle R \rangle$ , in the plane containing  $\langle I \rangle$  and normal to the vector defined by the argument  $\langle normal \rangle$  (non-zero 3D point). The  $\langle draw\_options \rangle$  argument is a string (empty by default) that will be passed as is to the `\draw` instruction. The  $\langle clip \rangle$  argument is set to `false` by default; it indicates whether the plot should be clipped to the current 3D window.

Another possible syntax: **g:Dcircle3d(C [, draw\_options, clip])** where  $\langle C \rangle = \{\langle I \rangle, \langle R \rangle, \langle normal \rangle\}$ .

- The **ld.circle3d(I, R, normal)** function returns the list of points on this circle (3D polygonal line).
- The **ld.circle3db(I, R, normal)** function returns this circle as a 3D path ((see *Dpath3d* page 86)) using Bézier curves.

**3D Path: Dpath3d**

The method **g:Dpath3d(path [, draw\_options, clip])** draws the  $\langle path \rangle$ . The  $\langle draw\_options \rangle$  argument is a string (empty by default) that will be passed as is to the `\draw` instruction. The  $\langle clip \rangle$  argument is set to `false` by default; it indicates whether the plot should be clipped to the current 3D window. The  $\langle path \rangle$  argument is a list of 3D points followed by instructions (strings) that function **on the same principle as in 2D**. Instructions available and their syntax, the word *last* represents the last point of the previous piece:

- $p_1, "m"$  (moveto), which starts a new component of the path at the 3D point  $p_1$ .
- $p_1, \dots, p_n, "l"$  (lineto) draws the 3D polygonal line  $\{last, p_1, \dots, p_n\}$ .
- $c_1, c_2, p_2, "b"$  (Bézier) draws the Bézier curve  $\{last, c_1, c_2, p_2\}$ , where  $c_1$  and  $c_2$  are the two control 3D points.
- $p_1, n, "c"$  (circle) draws the circle centered at  $p_1$  and passing through the point *last*, and normal to the 3D vector  $n$ .
- $p_1, p_2, r, sens, n, "ca"$  (circle arc) draws a circular arc centered at  $p_1$ , with radius  $r$ , extending from *last* to  $p_2$ , in the clockwise direction when  $sens=1$  (and therefore in the counterclockwise direction if  $sens=-1$ ). The 3D vector  $n$  is optional; it indicates a normal vector to the plane of the circle when the points *last*,  $p_1$ , and  $p_2$  are collinear (and in this case, the vector  $n$  is mandatory).
- `"c1"` (closepath) is used alone; it closes the current component by drawing a line segment connecting the last point to the first point (of the current component).

Here, for example, is the code in figure 2.

```
\begin{luadraw}{name=viewdir}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK
local g = ld.graph3d:new{ size={8,8} }
local i, M = ld.cpx.I, ld.pt3d.M
local O, A = Origin, M(4,4,4)
local B, C, D, E = ld.pxy(A), ld.px(A), ld.py(A), ld.pz(A) --projected from A onto the xOy plane and onto the axes
g:Dpolyline3d( {{O,A},{-5*vecI,5*vecI},{-5*vecJ,5*vecJ},{-5*vecK,5*vecK}}, {">"} -- axes
g:Dpolyline3d( {{E,A,B,O}}, {C,B,D}}, {"dashed"})
g:Dpath3d( {C,O,B,2.5,1,"ca",O,"l","c1"}, {"draw=none,fill=cyan,fill opacity=0.8"}) --angular sector
g:Darc3d(C,O,B,2.5,1,">") -- arc de cercle pour theta
g:Dpath3d( {E,O,A,2.5,1,"ca",O,"l","c1"}, {"draw=none,fill=cyan,fill opacity=0.8"}) --angular sector
g:Darc3d(E,O,A,2.5,1,">") -- arc of a circle for phi
g:Dballdots3d(O) -- the point of origin in the form of a small sphere
g:Labelsize("footnotesize")
g:Dlabel3d(
  "$x$", 5.25*vecI,{}, "$y$", 5.25*vecJ,{}, "$z$", 5.25*vecK,{},
  "towards the observer", A, {pos="E"},
  "$O$", O, {pos="NW"},
  "$\\theta$", (B+C)/2, {pos="N", dist=0.15},
  "$\\varphi$", (A+E)/2, {pos="S", dist=0.25})
g:Dlabel("viewdir=\\{\"ortho\", \"$\\theta$,\\varphi$\\} (in degrees)\",-5*i,{pos="N"}) -- label 2D
g:Show()
\end{luadraw}
```

**Conversion:** The function `ld.polyline2path3d(L)` : Returns  $\langle L \rangle$  which is a list of 3D points or a list of lists of 3D points, in the form of a path.

### Plane: Dplane

- The method `g:Dplane(P, V, L1, L2 [, mode, draw_options])` draws the edges of the plane  $\langle P \rangle = \{A, u\}$  where  $A$  is a point in the plane and  $u$  is a normal vector to the plane ( $\langle P \rangle$  is therefore a table of two 3D points). The argument  $\langle V \rangle$  must be a non-zero vector in the plane  $\langle P \rangle$ . The arguments  $\langle L1 \rangle$  and  $\langle L2 \rangle$  are two lengths. The method constructs a parallelogram centered on  $A$ , with one side  $L_1 \frac{V}{\|V\|}$  and the other  $L_2 \frac{W}{\|W\|}$  where  $W = u \wedge V$ . The argument  $\langle mode \rangle$  is a natural number indicating the edges to draw. To calculate this integer, we use the predefined variables: `ld.top` (=8), `ld.right` (=4), `ld.bottom` (=2), `ld.left` (=1), and `ld.all` (=15), which can be added together, for example:

- `mode=ld.bottom+ld.left`: for the bottom and left sides
- `mode=ld.top+ld.right+ld.bottom`: for the top, right, and bottom sides
- etc.

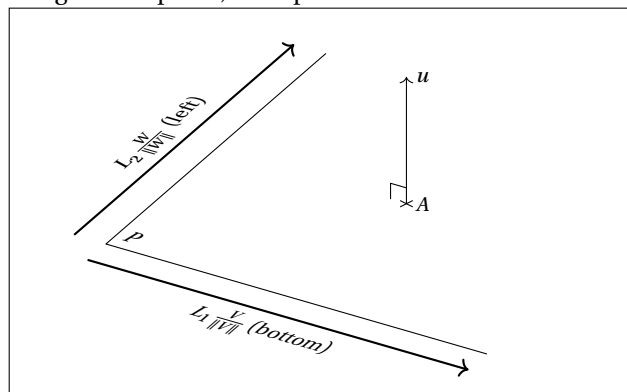
By default, the mode is `ld.all`, which corresponds to `mode=ld.top+ld.right+ld.bottom+ld.left`.

- The function `ld.plane2rectangle(P [, V, L1, L2])` calculates and returns the rectangle (list of 3D points) drawn by the `g:Dplane()` method. The vector  $\langle V \rangle$  is optional; it allows you to define an edge of the rectangle (it must lie on the plane). If it is omitted, the function will choose a vector  $V$  itself. The lengths  $\langle L1 \rangle$  and  $\langle L2 \rangle$  are optional and default to 5. When the argument  $\langle L2 \rangle$  is omitted, it implicitly has the same value as  $\langle L1 \rangle$ . The result can be drawn using the `g:Dpolyline3d()` method, or drawn as a facet.

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

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

Figure 5: Dplane, example with mode = left+bottom





**Warning** : The concepts of top, right, bottom, and left are relative! They depend on the direction of the vectors  $u$  (vector normal to the plane) and  $V$  (vector given in the plane). The third vector  $W$  is the cross product  $u \wedge V$ .

### Parametric curve: Dparametric3d

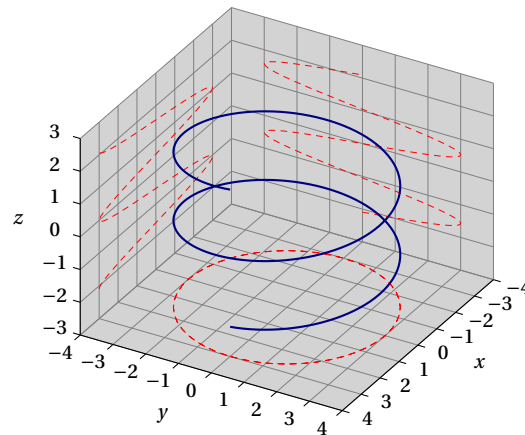
- The function **ld.parametric3d(p, t1, t2 [, nbdots, discont, nbdiv])** calculates the points of the curve and returns a 3D polygonal line (no drawing).
  - The argument  $\langle p \rangle$  is the parameterization. It must be a function of a real variable  $t$  with values in  $\mathbf{R}^3$  (the images are 3D points), for example: `local p=function(t) return Mc(3,t,t/3) end.`
  - The arguments  $\langle t1 \rangle$  and  $\langle t2 \rangle$  are mandatory with  $t_1 < t_2$ ; they form the bounds of the interval for the parameter.
  - The argument  $\langle nbdots \rangle$  is optional; it is the (minimum) number of points to calculate; it is 40 by default.
  - The argument  $\langle discont \rangle$  is an optional boolean that indicates whether there are discontinuities or not. It is `false` by default.
  - The argument  $\langle nbdiv \rangle$  is a positive integer equal to 5 by default and indicates the number of times the interval between two consecutive parameter values can be split in two (dichotomized) when the corresponding points are too far apart.
- The method **g:Dparametric3d(p, options)** calculates the points and draws the curve parameterized by  $\langle p \rangle$ . The argument  $\langle options \rangle$  is a table specifying the possible options. Here are these options with their default values:
  - `t={g:Xinf(),g:Xsup()},`
  - `nbdots=40,`
  - `discont=false,`
  - `nbdiv=5,`
  - `clip=false,` it indicates whether the curve should be clipped with the current 3D window.
  - `draw_options=""`, string that will be passed as is to the `\draw` instruction.

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

local g = ld.graph3d:new{window3d={-4,4,-4,4,-3,3}, window={-7.5,6.5,-7,6}, size={8,8}}
local pi = math.pi
g:Labelsize("footnotesize")
local p = function(t) return Mc(3,t,t/3) end
local L = ld.parametric3d(p,-2*pi,2*pi,25,false,2)
g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
g:Lineoptions("dashed","red",2)
-- projection onto the plane y=-4
g:Dpolyline3d(ld.proj3d(L,{M(0,-4,0),vecJ}))
-- projection onto the plane x=-4
g:Dpolyline3d(ld.proj3d(L,{M(-4,0,0),vecI}))
-- projection onto the plane z=-3
g:Dpolyline3d(ld.proj3d(L,{M(0,0,-3),vecK}))
-- curve drawing
g:Lineoptions("solid","Navy",8)
g:Dparametric3d(p,{t={-2*pi,2*pi}})
g:Show()
\end{luadraw}
```



Figure 6: A curve and its projections onto three planes



### Parameterization of a Polygonal Line: *curvilinear\_param3d*

Let  $L$  be a list of 3D points representing a continuous line. It is possible to obtain a parameterization of this line based on a parameter  $t$  between 0 and 1 ( $t$  is the curvilinear abscissa divided by the total length of  $L$ ).

The function `ld.curvilinear_param3d(L [, close])` returns a function of one variable  $t \in [0; 1]$  and values on the line  $\langle L \rangle$  (3D points). The value at  $t = 0$  is the first point of  $\langle L \rangle$ , and the value at  $t = 1$  is the last point; This function is followed by a number representing the total length of  $\langle L \rangle$ . The optional argument  $\langle close \rangle$  indicates whether the line should be closed (`false` by default).

### The reference: *Dboxaxes3d*

The `g:Dboxaxes3d(options)` method allows you to draw the three axes, with a number of options defined in the  $\langle options \rangle$  table. These options are:

- `xaxe=true`, `yaxe=true`, and `zaxe=true`: Indicates whether the corresponding axes should be drawn or not.
- `drawbox=false`: Indicates whether a box should be drawn with the axes.
- `grid=false`: Indicates whether a grid should be drawn (one for  $x$ , one for  $y$ , and one for  $z$ ). When this option is true, the following options can also be used:
  - `gridwidth=1`: Indicates the grid line thickness in tenths of a point,
  - `gridcolor="black"`: Indicates the grid color,
  - `fillcolor=""`: color to paint the grid background.
- `xlimits={x1,x2}`, `ylimits={y1,y2}`, `zlimits={z1,z2}`: Allows you to define the three intervals used for the axis lengths. By default, these are the values provided to the `window3d` argument when creating the graph.
- `xgradlimits={x1,x2}`, `ygradlimits={y1,y2}`, `zgradlimits={z1,z2}`: Allows you to define the three graduation intervals on the axes. By default, these options are set to `"auto"`, meaning they take the same values as `xlimits`, `ylimits`, and `zlimits`.
- `xyzstep=1`: Specifies the tick step on all three axes.
- `xstep=xyzstep`, `ystep=xyzstep`, `zstep=xyzstep`: Specifies the tick step on each axis (value of `xyzstep` by default).
- `xyzticks=0.2`: Specifies the length of the tick marks.
- `labels=true`: Specifies whether or not to display the tick mark values.
- `xlabelsep=0.25`, `ylabelsep=0.25`, `zlabelsep=0.25`: Specifies the distance between the labels and the graduations.

- `xlabelstyle=<current style>`, `ylabelstyle=<current style>`, `zlabelstyle=<current style>`: Specifies the label style, i.e., the position relative to the anchor point. By default, the current style is applied.
- `xlegend="$x$", ylegend="$y$", zlegend="$z$"`: Allows you to define a legend for the axes.
- `xlegendsep=0.5, ylegendsep=0.5, zlegendsep=0.5`: Specifies the distance between the legends and the graduations.

## 2) Points and Labels

### 3D Points: `Ddots3d`, `Dballdots3d`, `Dcrossdots3d`

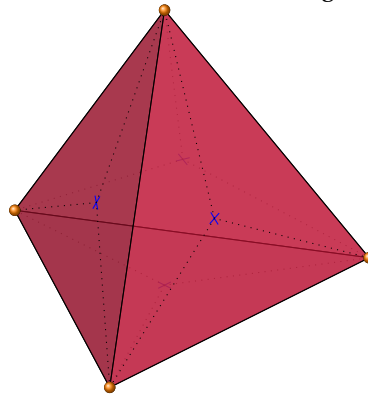
There are three ways to draw 3D points. For the first two, the argument  $\langle L \rangle$  can be either a single 3D point, a list (a table) of 3D points, or a list of lists of 3D points:

- The `g:Ddots3d(L [, mark_options, clip])` method. The principle is the same as in the 2D version; the points are drawn in the current line color with the current style. The  $\langle mark\_options \rangle$  argument is an optional string that will be passed as is to the `\draw` instruction (local modifications). The  $\langle clip \rangle$  argument is set to `false` by default; it indicates whether the plot should be clipped to the current 3D window.
- The `g:Dballdots3d(L [, color, scale, clip])` method draws the points of  $\langle L \rangle$  as a sphere. The optional  $\langle color \rangle$  argument specifies the color of the sphere ("black" by default), and the optional  $\langle scale \rangle$  argument allows you to change the size of the sphere (1 by default).
- The `g:Dcrossdots3d(L [, color, scale, clip])` method draws the points of  $\langle L \rangle$  as a plane cross. The argument  $\langle L \rangle$  is a list of the form {3D point, normal vector} or { {3D point, normal vector}, {3D point, normal vector}, ...}. For each 3D point, the associated normal vector is used to determine the plane containing the cross. The optional  $\langle color \rangle$  argument specifies the color of the sphere ("black" by default), and the optional  $\langle scale \rangle$  argument allows you to change the size of the sphere (1 by default).

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

local g = ld.graph3d:new{viewdir={15,60},bbox=false,size={8,8}}
local A, B, C, D = 4*M(1,0,-0.5), 4*M(-1/2,math.sqrt(3)/2,-0.5), 4*M(-1/2,-math.sqrt(3)/2,-0.5), 4*M(0,0,1)
local u, v, w = B-A, C-A, D-A
-- drawing of the center of gravity of hidden faces
for _, F in ipairs({{A,B,C},{B,C,D}}) do
local G, u = pt3d.isobar3d(F), pt3d.prod(F[2]-F[1],F[3]-F[1])
g:Dcrossdots3d({G,u}, "blue", 0.75)
g:Dpolyline3d({{F[1],G,F[2]},{G,F[3]}}, "dotted")
end
-- drawing of the tetrahedron constructed on A, B, C and D
g:Dpoly( ld.tetra(A,u,v,w),{mode=mShaded,opacity=0.7,color="Crimson"})
-- drawing of the center of gravity of visible faces
for _, F in ipairs({{A,B,D},{A,C,D}}) do
local G, u = pt3d.isobar3d(F), pt3d.prod(F[2]-F[1],F[3]-F[1])
g:Dcrossdots3d({G,u}, "blue", 0.75)
g:Dpolyline3d({{F[1],G,F[2]},{G,F[3]}}, "dotted")
end
g:Dballdots3d({A,B,C,D}, "orange") --vertices
g:Show()
\end{luadraw}
```

Figure 7: A tetrahedron and the centers of gravity of each face



### 3D Labels: Dlabel3d

The method for placing a label in space is:

**g:Dlabel3d(text1, anchor1, options1, text2, anchor2, options2, ...).**

- The arguments  $\langle text1 \rangle, \langle text2 \rangle, \dots$ , are strings; they are the labels.
- The arguments  $\langle anchor1 \rangle, \langle anchor2 \rangle, \dots$ , are 3D points representing the anchor points of the labels.
- $T\langle options1 \rangle, \langle options2 \rangle, \dots$ , allow you to locally define the label options. These options are (with their default value)!
  - `pos="center"`: indicates the label's position in the screen plane relative to the anchor point. It can be "N" for north, "NE" for northeast, "NW" for northwest, or "S", "SE", "SW". By default, it is "center", and in this case, the label is centered on the anchor point.
  - `dist=0`: distance in cm between the label and its anchor point when `pos` is not equal to "center".
  - `dir={}`: indicates the direction of writing in space, the empty list means the default direction). In general, `dir={dirX, dirY, dep}`, the three values `dirX`, `dirY` and `dep` are three 3D points representing three vectors. The first two indicate the direction of writing, the third a displacement (translation) of the label relative to the anchor point.
  - `node_options=""`: string intended to receive options that will be passed directly to TikZ in the `\node[]` instruction.
  - The labels are drawn in the current color of the document text, but the color can be changed with the `node_options` argument, for example, by setting: `node_options="color=blue"`.

**Warning:** The options chosen for a label also apply to subsequent labels if they are unchanged.

## 3) Basic Solids (Without Facets)

The four methods described below are sensitive to the global variable `Hiddenlines` which is `false` by default.

### Cylinder: Dcylinder

Draw a cylinder with a circular base (right or tilted). Several possible syntaxes:

- Old syntax: **g:Dcylinder(A, V, r [, options])** draws a right cylinder, where  $\langle A \rangle$  is a 3D point representing the center of one of the circular faces,  $\langle V \rangle$  is a 3D point, a vector representing the axis of the cone, the center of the opposite circular face is the point  $A + V$  (this face is orthogonal to  $V$ ), and  $\langle r \rangle$  is the radius of the circular base.
- Syntax: **g:Dcylinder(A, r, B [, options])** draws a right cylinder, where  $\langle A \rangle$  is a 3D point representing the center of one of the circular faces,  $\langle B \rangle$  is the center of the opposite face, and  $\langle r \rangle$  is the radius. The cylinder is right, meaning that the circular faces are orthogonal to the axis ( $AB$ ).
- For a tilted cylinder: **g:Dcylinder(A, r, V, B [, options])**, where  $\langle A \rangle$  is a 3D point representing the center of one of the circular faces,  $\langle B \rangle$  is the center of the opposite circular face,  $\langle r \rangle$  is the radius, and  $\langle V \rangle$  is a non-zero 3D vector orthogonal to the plane of the circular faces.

For all three syntaxes,  $\langle options \rangle$  is a table whose fields define the options. These options are (with their default values):

- `mode=ld.mWireframe`, two possible values, `ld.mGrid` or `ld.mWireframe`. In `ld.mWireframe` mode, this is a wireframe drawing; in `ld.mGrid` mode, this is a grid drawing (as if there were facets).
- `hiddenstyle=ld.Hiddenlinestyle`, sets the line style for hidden areas (set it to `"noline"` to hide them). By default, this option has the value of the global variable `ld.Hiddenlinestyle`, which is itself initialized with the value `"dotted"`.
- `hiddencolor=edgecolor`, sets the color of hidden lines. By default, it is the same value as the `edgecolor` option.
- `edgecolor=<current color>`, sets the color of the lines.
- `edgestyle=<current style>`, defines the line style for visible edges.
- `edgewidth=<current width>`, sets the thickness of the visible edges in tenths of a point.
- `color=""`, when this option is an empty string (the default value), there is no fill; when it is a color (as a string), there is a fill with a linear gradient.
- `opacity=1`, sets the transparency of the drawing.
- Gradient parameters: when there is a color fill, a linear gradient is used for the side of the cylinder and another for the section; in both cases, the gradient is defined as follows:

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

where  $\langle color \rangle$  denotes the color used, and  $\langle l \rangle$ ,  $\langle m \rangle$ ,  $\langle r \rangle$  are three numbers between 0 and 100, these three numbers constitute the parameters of the gradient in the form of a table  $\{\langle l \rangle, \langle m \rangle, \langle r \rangle\}$ :

- For the side of the cylinder, it's the `gradside` option, by default: `gradside={50,10,100}`.
- For the cylinder section, it's the `gradsection` option, by default: `gradsection={25,18,50}`.

### Cone: Dcone

Draw a circular cone (right or inclined). Several possible syntaxes:

- Old syntax: `g:Dcone(A, V, r [, options])` draws a right cone, where  $\langle A \rangle$  is a 3D point representing the cone's vertex,  $\langle V \rangle$  is a 3D point, a vector representing the cone's axis, the center of the circular face is point  $A + V$  (this face is orthogonal to  $\langle V \rangle$ ), and  $\langle r \rangle$  is the radius of the circular base.
- Syntax: `g:Dcone(C, r, A [, options])` draws a right cone, where  $\langle A \rangle$  is a 3D point representing the cone's vertex,  $\langle C \rangle$  is the center of the circular face, and  $\langle r \rangle$  is the radius. The cone is right, meaning that the circular face is orthogonal to the axis  $(AC)$ .
- For a tilted cone: `g:Dcone(C, r, V, A [, options])`, where  $\langle A \rangle$  is a 3D point representing the cone's vertex,  $\langle C \rangle$  is the center of the circular face,  $\langle r \rangle$  is the radius, and  $\langle V \rangle$  is a non-zero 3D vector orthogonal to the plane of the circular face.

For all three syntaxes,  $\langle options \rangle$  is a table whose fields define the options. These options are (with their default values):

- `mode=ld.mWireframe`, two possible values, `ld.mGrid` or `ld.mWireframe`. In `ld.mWireframe` mode, this is a wireframe drawing; in `ld.mGrid` mode, this is a grid drawing (as if there were facets).
- `hiddenstyle=ld.Hiddenlinestyle`, sets the line style for hidden areas (set it to `"noline"` to hide them). By default, this option has the value of the global variable `ld.Hiddenlinestyle`, which is itself initialized with the value `"dotted"`.
- `hiddencolor=edgecolor`, sets the color of hidden lines. By default, it is the same value as the `edgecolor` option.
- `edgecolor=<current color>`, sets the color of the lines.
- `edgestyle=<current style>`, defines the line style for visible edges.

- `edgewidth=<current width>`, sets the thickness of the visible edges in tenths of a point.
- `color=""`, when this option is an empty string (the default value), there is no fill; when it is a color (as a string), there is a fill with a linear gradient.
- `opacity=1`, sets the transparency of the drawing.
- Gradient parameters: when there is a color fill, a linear gradient is used for the side of the cone and another for the section; in both cases, the gradient is defined as follows:

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

where `<color>` denotes the color used, and `<l>`, `<m>`, `<r>` are three numbers between 0 and 100, these three numbers constitute the parameters of the gradient in the form of a table `{<l>,<m>,<r>}`:

- For the side of the cone, it's the `gradside` option, by default: `gradside={50,10,100}`.
- For the cone section, it's the `gradsection` option, by default: `gradsection={25,18,50}`.

### Frustum: Dfrustum

Draw a truncated cone with a circular base (right or slanted). Two possible syntaxes:

- The syntax: `g:Dfrustum(A, R, r, V [, options])` for a right truncated cone, `<A>` is a 3D point representing the center of the face with radius `<R>`, `<V>` is a 3D vector representing the axis of the truncated cone, the center of the second circular face is the point `A + V`, and its radius is `<r>` (the faces are orthogonal to `<V>`). When `R = r` we simply have a cylinder.
- Syntax: `g:Dfrustum(A, R, r, V, B [, options])` for a tilted cone frustum, `<A>` is a 3D point representing the center of the face with radius `<R>`, `<V>` is a 3D vector representing a normal vector to the circular faces, the center of the second circular face is point `<B>`, and its radius is `<r>`. When `R = r` we have a tilted cylinder.

In both cases, `<options>` is a table whose fields define the options. These options are (with their default values):

- `mode=ld.mWireframe`, two possibles values, `ld.mGrid` or `ld.mWireframe`. In `ld.mWireframe` mode, this is a wireframe drawing; in `ld.mGrid` mode, this is a grid drawing (as if there were facets).
- `hiddenstyle=ld.Hiddenlinestyle`, sets the line style for hidden areas (set it to `"noline"` to hide them). By default, this option has the value of the global variable `ld.Hiddenlinestyle`, which is itself initialized with the value `"dotted"`.
- `hiddencolor=edgecolor`, sets the color of hidden lines. By default, it is the same value as the `edgecolor` option.
- `edgecolor=<current color>`, sets the color of the lines.
- `edgestyle=<current style>`, defines the line style for visible edges.
- `edgewidth=<current width>`, sets the thickness of the visible edges in tenths of a point.
- `color=""`, when this option is an empty string (the default value), there is no fill; when it is a color (as a string), there is a fill with a linear gradient.
- `opacity=1`, sets the transparency of the drawing.
- Gradient parameters: when there is a color fill, a linear gradient is used for the side of the cone and another for the section; in both cases, the gradient is defined as follows:

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

where `<color>` denotes the color used, and `<l>`, `<m>`, `<r>` are three numbers between 0 and 100, these three numbers constitute the parameters of the gradient in the form of a table `{<l>,<m>,<r>}`:

- For the side of the cone, it's the `gradside` option, by default: `gradside={50,10,100}`.
- For the cone section, it's the `gradsection` option, by default: `gradsection={25,18,50}`.

**Sphere: Dsphere**

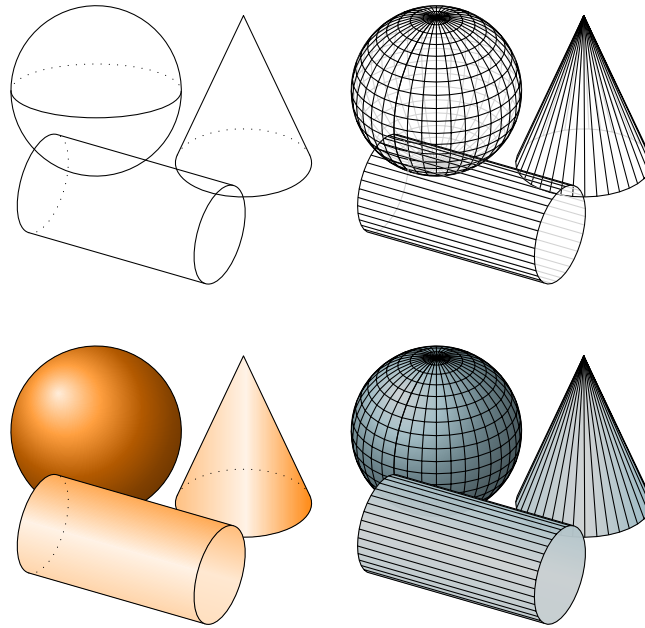
The `g:Dsphere(A, r [, options])` method draws a sphere.

- $\langle A \rangle$  is a 3D point representing the center of the sphere.
- $\langle r \rangle$  is the radius of the sphere.
- $\langle options \rangle$  is a table whose fields define the options. These options are (with their default values):
  - `mode=ld.mWireframe`, three possibles values, `ld.mGrid` or `ld.mWireframe` or `ld.mBorder`. In `ld.mWireframe` mode, the outline (circle) and the equator are drawn; in `ld.mGrid` mode, the outline with meridians and spindles (grid) is drawn; and in `ld.mBorder` mode, the outline only is drawn.
  - `hiddenstyle=ld.Hiddenlinestyle`, sets the line style for hidden areas (set it to `"noline"` to hide them). By default, this option has the value of the global variable `ld.Hiddenlinestyle`, which is itself initialized with the value `"dotted"`.
  - `hiddencolor=edgecolor`, sets the color of hidden lines. By default, it is the same value as the `edgecolor` option.
  - `color=""`, when this option is an empty string (the default value), there is no fill; when it is a color (as a string), there is a fill with a "ball color".
  - `opacity=1`, defines the transparency of the drawing.
  - `edgecolor=<current color>`, sets the color of the lines.
  - `edgestyle=<currrent style>`, defines the line style for visible edges.
  - `edgewidth=<current width>`, sets the thickness of the visible edges in tenths of a point.

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

local g = ld.graph3d:new{ size={10,10} }
local dessin = function(args)
  g:Dsphere(M(-1,-2.5,1),2.5, args)
  g:Dcone(M(-1,2.5,5),-5*vecK,2, args)
  g:Dcylinder(M(3,-2,0),6*vecJ,1.5, args)
end
-- top left, default options
g:Saveattr(); g:Viewport(-5,0,0,5); g:Coordsystem(-5,5,-5,5,true); dessin(); g:Restoreattr()
g:Saveattr(); g:Viewport(0,5,0,5); g:Coordsystem(-5,5,-5,5,true)
-- top right
dessin({mode=ld.mGrid, hiddenstyle="solid", hiddencolor="LightGray"}); g:Restoreattr()
g:Saveattr(); g:Viewport(-5,0,-5,0); g:Coordsystem(-5,5,-5,5,true)
-- bottom left
dessin({mode=ld.mBorder, color="orange"}); g:Restoreattr()
g:Saveattr(); g:Viewport(0,5,-5,0); g:Coordsystem(-5,5,-5,5,true)
-- bottom right
dessin({mode=ld.mGrid,opacity=0.8,hiddenstyle="noline",color="LightBlue"}); g:Restoreattr()
g:Show()
\end{luadraw}
```

Figure 8: Cylinders, Cones, and Spheres



## IV Faceted Solids

### 1) Definition of a Solid

There are two ways to define a solid:

1. As a list (table) of facets. A facet is itself a list of 3D points (at least 3) that are coplanar and unaligned, which are the vertices. Facets are assumed to be convex and are oriented by the order of appearance of the vertices. That is, if  $A$ ,  $B$ , and  $C$  are the first three vertices of a facet  $F$ , then the facet is oriented with the normal vector  $\vec{AB} \wedge \vec{AC}$ . If this normal vector is directed toward the observer, then the facet is considered visible. The method **g:Cosine\_incidence(N [, A])** returns the cosine of the angle at point  $\langle A \rangle$  between the vector  $\langle N \rangle$  (which must be unit) and the unit vector directed towards the observer; if this cosine is positive then  $\langle N \rangle$  is directed towards the observer (point  $\langle A \rangle$  is optional EXCEPT in central projection).

In the definition of a solid, the normal vectors to the facets must be directed **outside** the solid for the orientation to be correct.

2. In the form of a **polyhedron**, that is to say a table with two fields, a first field called *vertices* which is the list of vertices of the polyhedron (3D points), and a second field called *facets* which is the list of facets, but here, in the definition of the facets, the vertices are replaced by their index in the *vertices* list. The facets are oriented in the same way as before. This definition corresponds to the *obj* format but without normal vectors.

For example, let's consider the four points  $A = M(-2, -2, 0)$ ,  $B = M(3, 0, 0)$ ,  $C = M(-2, 2, 0)$ , and  $D = M(0, 0, 4)$ . We can then define the tetrahedron constructed on these four points:

- either as a list of facets:  $T = \{\{A, B, D\}, \{B, C, D\}, \{C, A, D\}, \{A, C, B\}\}$  (pay attention to the orientation),
- or as a polyhedron:  $T = \{\text{vertices} = \{A, B, C, D\}, \text{facets} = \{\{1, 2, 4\}, \{2, 3, 4\}, \{3, 1, 4\}, \{1, 3, 2\}\}\}$ .

### Functions for converting between the two definitions

- The function **ld.poly2facet(P)** where  $\langle P \rangle$  is a polyhedron, returns this solid as a list of facets.
- The function **ld.facet2poly(L [, epsilon])** returns the list of facets  $\langle L \rangle$  as a polyhedron. The optional argument  $\langle \text{epsilon} \rangle$  defaults to  $10^{-8}$ , specifying the accuracy of the comparisons between 3D points. If there are many facets, the computation time can become significant.



## 2) Drawing a Polyhedron: Dpoly

The function **g:Dpoly(P, options)** allows you to represent the polyhedron  $\langle P \rangle$  (using the naive painter's algorithm). The argument  $\langle options \rangle$  is a table containing the options, these are, with their default value:

- **mode=ld.mShaded**: Sets the representation mode. There are six possible values:
  - **ld.mWireframe**: Wireframe mode, draws both visible and hidden edges.
  - **ld.mFlat**: Draws solid-color faces, as well as visible edges.
  - **ld.mFlatHidden**: Draws solid-color faces, visible edges, and hidden edges.
  - **ld.mShaded**: Draws the faces in shaded color based on their inclination, as well as the visible edges. This is the default mode.
  - **ld.mShadedHidden**: Draws the faces in shaded color based on their inclination, with both visible and hidden edges.
  - **ld.mShadedOnly**: Draws the faces in shaded color based on their inclination, but not the edges.
- **contrast=1**: This number allows you to accentuate or diminish the shade of the facet colors in the **ld.mShaded**, **ld.mShadedHidden**, and **ld.mShadedOnly** modes.
- **edgestyle=<current style>**: This is a string that defines the line style of the edges.
- **edgecolor=<current line color>**: This is a string that defines the edge color.
- **hiddenstyle=ld.Hiddenlinestyle**: This is a string that defines the line style of hidden edges. By default, this is the value contained in the global variable **ld.Hiddenlinestyle** (which itself is "dotted" by default).
- **hiddencolor=<current line color>**: This is a string that defines the color of hidden edges.
- **edgewidth=<current lien width>**: This is the line thickness of edges in tenths of a point.
- **opacity=1**: This is a number between 0 and 1 that allows you to set transparency or not on the facets.
- **backcull=false**: When this boolean is **true**, facets considered invisible (normal vectors not directed towards the observer) are not displayed. This option is useful for convex polyhedra because it reduces the number of facets to draw.
- **twoside=true**: Boolean that defaults to true, meaning that both sides of the facets (inner and outer) are distinguished; the two sides will not have exactly the same color.
- **color="white"**: String defining the fill color of the facets.
- **usepalette=nil**: this option allows you to specify a color palette for painting the facets as well as a calculation mode, the syntax is: **usepalette={palette,mode}**, where  $\langle palette \rangle$  designates a table of colors which are themselves tables of the form  $\{r, g, b\}$  where  $r$ ,  $g$  and  $b$  are numbers between 0 and 1. The argument  $\langle mode \rangle$  can be :
  - one of the strings: "x", "y", "z". In the first case, for example, facets with the minimum x-coordinate at their centroid have the first color in the palette, facets with the maximum x-coordinate at their centroid have the last color in the palette, and for the others, the color is calculated based on the x-coordinate of the centroid by linear interpolation.
  - a function:  $\langle mode \rangle: f \mapsto mode(f) \in \mathbb{R}$ , where  $f$  denotes a facet (a list of 3D points). Facets with the minimum value have the first color in the palette, those with the maximum value have the last color in the palette, and for the others, the color is calculated by linear interpolation.

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

local g = ld.graph3d:new{viewdir={10,60},bbox=false, size={10,10}, bg="gray!30"}
```

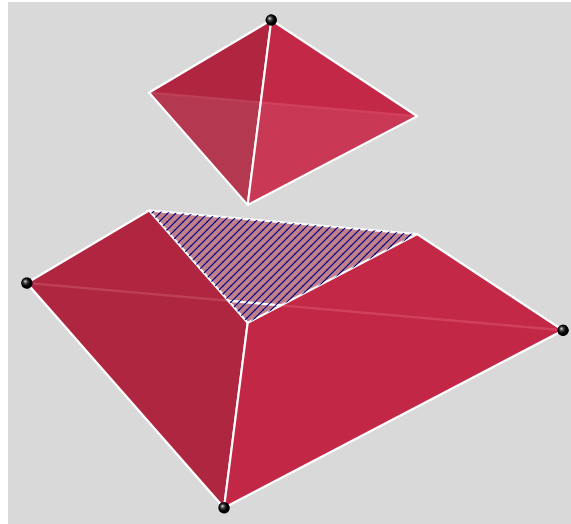


```

local A,B,C,D = M(-2,-4,-2),M(4,0,-2),M(-2,4,-2),M(0,0,2)
local T = ld.tetra(A,B-A,C-A,D-A) -- tetrahedron with vertices A, B, C, D
local plan = {Origin, -vecK} -- sectional plane
local T1, T2, section = ld.cutpoly(T,plan) -- we cut the tetrahedron
-- T1 is the resulting polyhedron in the half-space containing -vecK
-- T2 is the resulting polyhedron in the other half-space
-- section is a facet (it's the cut)
g:Dpoly(T1,{color="Crimson", edgecolor="white", opacity=0.8, edgewidth=8})
g:Filloptions("bdiag","Navy"); g:Dpolyline3d(section,true,"draw=none")
g:Dpoly(ld.shift3d(T2,2*vecK), {color="Crimson", edgecolor="white", opacity=0.8, edgewidth=8})
g:Dballdots3d({A,B,C,D+2*vecK}) -- we drew T2 translated with the vector 2*vecK
g:Show()
\end{luadraw}

```

Figure 9: Section of a tetrahedron by a plane



### 3) Displaying the Face and/or Vertex Numbers of a Polyhedron

The method **g:Dpolynames(P,option,opacity)** displays the polyhedron  $P$  with the option to add to each face its number (which is its position in the  $P$ .facets list) preceded by the letter  $F$ , and optionally the number of each vertex (which is its position in the  $P$ .vertices list) preceded by the letter  $V$ . The argument *option* can take the values: "facet", "vertex", or "both" (which is the default value). The argument *opacity* is a number between 0 and 1 which defaults to 0.6.

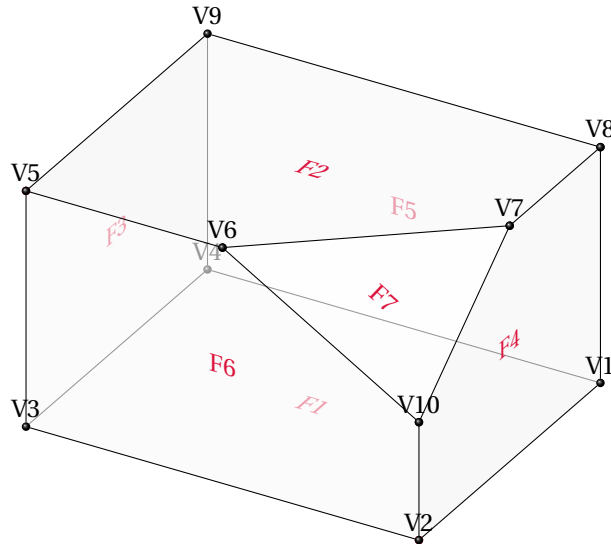
```

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

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

```

Figure 10: Visualize the faces and vertices of a polyhedron



#### 4) Polyhedron Construction Functions

The following functions return a polyhedron, that is, a table with two fields: a first field called *vertices*, which is the list of the polyhedron's vertices (3D points), and a second field called *facets*, which is the list of facets. However, in the definition of facets, the vertices are replaced by their index in the *vertices* list.

- **ld.tetra(S, v1, v2, v3)** returns the tetrahedron constructed from vertex  $\langle S \rangle$  (3D point) and the three vectors  $\langle v1 \rangle$ ,  $\langle v2 \rangle$ ,  $\langle v3 \rangle$  (3D points) assumed to be in the forward direction. The vertices of this tetrahedron are  $S$ ,  $S + v_1$ ,  $S + v_2$ , and  $S + v_3$ .
- **ld.parallelep(S, v1, v2, v3)** returns the parallelepiped constructed from vertex  $\langle S \rangle$  (3D point) and the three vectors  $\langle v1 \rangle$ ,  $\langle v2 \rangle$ ,  $\langle v3 \rangle$  (3D points) assumed to be in the forward direction.
- **ld.prism(base, vector [, open])** returns a prism. The argument  $\langle base \rangle$  is a list of 3D points (one of the two bases of the prism).  $\langle vector \rangle$  is the translation vector (3D point) used to obtain the second base. The optional argument  $\langle open \rangle$  is a Boolean indicating whether the prism is open or not (**false** by default). If it is open, only the lateral facets are returned. The  $\langle base \rangle$  must be oriented by the  $\langle vector \rangle$ .
- **ld.pyramid(base, vertex [, open])** returns a pyramid. The argument  $\langle base \rangle$  is a list of 3D points, and  $\langle vertex \rangle$  is the apex of the pyramid (3D point). The optional argument  $\langle open \rangle$  is a Boolean indicating whether the pyramid is open or not (**false** by default). If it is open, only the side facets are returned. The base must be oriented with the apex.
- **ld.regular\_pyramid(n, side, height [, open, center, axis])** returns a regular pyramid.  $\langle n \rangle$  is the number of sides of the base, the argument  $\langle side \rangle$  is the length of a side, and  $\langle height \rangle$  is the height of the pyramid. The optional argument  $\langle open \rangle$  is a Boolean indicating whether the pyramid is open or not (**false** by default). If it is open, only the lateral facets are returned. The optional argument  $\langle center \rangle$  is the center of the base (**Origin** by default), and the optional argument  $\langle axis \rangle$  is a direction vector of the pyramid axis (**vecK** by default).
- **ld.truncated\_pyramid(base, vertex, height [, open])** returns a truncated pyramid; the argument  $\langle base \rangle$  is a list of 3D points;  $\langle vertex \rangle$  is the apex of the pyramid (3D point). The argument  $\langle height \rangle$  is a number indicating the height from the base where the truncation occurs; this is parallel to the plane of the base. The optional argument  $\langle open \rangle$  is a boolean indicating whether the pyramid is open or not (**false** by default). If it is open, only the lateral facets are returned. The base must be oriented by the apex.
- **ld.cylinder(A, V, R [, nbfacet, open])** returns a right cylinder of radius  $\langle R \rangle$ ,  $\langle A \rangle$  (3D point) is the center of one of the circular bases, and  $\langle V \rangle$  is a non-zero 3D vector such that the center of the second base is the point  $A + V$ . The optional argument  $\langle nbfacet \rangle$  is 35 by default (number of lateral facets). The optional argument  $\langle open \rangle$  is a Boolean indicating whether the cylinder is open or not (**false** by default). If it is open, only the lateral facets are returned.

- **ld.cylinder(A, R, B [, nbfacet, open])** returns a right cylinder of radius  $\langle R \rangle$ ,  $\langle A \rangle$  (3D point) is the center of one of the circular bases, and  $\langle B \rangle$  the center of the second base. The cylinder is right. The optional argument  $\langle nbfacet \rangle$  is 35 by default (number of lateral facets). The optional argument  $\langle open \rangle$  is a Boolean indicating whether the cylinder is open or not (**false** by default). If it is open, only the lateral facets are returned.
- **ld.cylinder(A, R, V, B [, nbfacet, open])** returns a cylinder of radius  $\langle R \rangle$ ,  $\langle A \rangle$  (3D point) is the center of one of the circular bases, and  $\langle B \rangle$  the center of the second base, and  $\langle V \rangle$  is a 3D vector normal to the plane of the circular bases (the cylinder can therefore be tilted). The optional argument  $\langle nbfacet \rangle$  is 35 by default (number of lateral facets). The optional argument  $\langle open \rangle$  is a boolean indicating whether the cylinder is open or not (**false** by default). If it is open, only the lateral facets are returned.
- **ld.cone(A, V, R [, nbfacet, open])** returns a cone with vertex  $\langle A \rangle$  (3D point), axis directed by  $\langle V \rangle$  (3D point), and base the circle with center  $A + V$  and radius  $\langle R \rangle$  (in a plane orthogonal to  $\langle V \rangle$ ). The optional argument  $\langle nbfacet \rangle$  is 35 by default (number of lateral facets). The optional argument  $\langle open \rangle$  is a boolean indicating whether the cone is open or not (**false** by default). If it is open, only the lateral facets are returned.
- **ld.cone(C, R, A [, nbfacet, open])** returns a cone with vertex  $\langle A \rangle$  (3D point),  $\langle C \rangle$  is the circular base center, and  $\langle R \rangle$  is its radius (in a plane orthogonal to the  $(AC)$  axis). The optional argument  $\langle nbfacet \rangle$  is 35 by default (number of lateral facets). The optional argument  $\langle open \rangle$  is a Boolean indicating whether the cone is open or not (**false** by default). If it is open, only the lateral facets are returned.
- **ld.cone(C, R, V, A [, nbfacet, open])** returns a cone with vertex  $\langle A \rangle$  (3D point),  $\langle C \rangle$  is the circular base center,  $\langle R \rangle$  is its radius, and the base is in a plane orthogonal to  $\langle V \rangle$  (3D vector). The  $(AC)$  axis is therefore not necessarily orthogonal to the circular face (tilted cone). The optional argument  $\langle nbfacet \rangle$  is 35 by default (number of lateral facets). The optional argument  $\langle open \rangle$  is a Boolean indicating whether the cone is open or not (**false** by default). If it is open, only the lateral facets are returned.
- **ld.frustum(C, R, r, V [, nbfacet, open])** returns a right frustum.  $\langle C \rangle$  (3D point) is the center of the circular base of radius  $\langle R \rangle$ , and vector  $\langle V \rangle$  directs the axis of the frustum. The center of the other circular base is point  $C + V$ , and its radius is  $\langle r \rangle$  (the bases are orthogonal to  $\langle V \rangle$ ). The optional argument  $\langle nbfacet \rangle$  is 35 by default (number of lateral facets). The optional argument  $\langle open \rangle$  is a Boolean indicating whether the frustum is open or not (**false** by default). If it is open, only the lateral facets are returned.
- **ld.frustum(C, R, r, V, A [, nbfacet, open])** returns a frustum of a cone.  $\langle C \rangle$  (3D point) is the center of the circular base of radius  $\langle R \rangle$ , the center of the other circular base is point  $\langle A \rangle$ , and its radius is  $\langle r \rangle$ . The bases are orthogonal to vector  $\langle C \rangle$ , but not necessarily orthogonal to axis  $(AC)$ . The optional argument  $\langle nbfacet \rangle$  is 35 by default (number of lateral facets). The optional argument  $\langle open \rangle$  is a Boolean indicating whether the frustum is open or not (**false** by default). If it is open, only the lateral facets are returned.
- **ld.sphere(A, R [, nbu, nbv])** returns the sphere with center  $\langle A \rangle$  (3D point) and radius  $\langle R \rangle$ . The optional argument  $\langle nbu \rangle$  represents the number of spindles (36 by default) and the optional argument  $\langle nbv \rangle$  the number of parallels (20 by default).

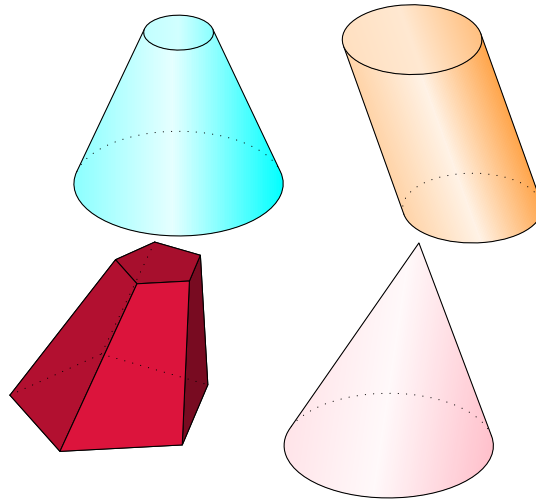
```

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

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

```

Figure 11: Truncated cone, truncated pyramid, oblique cylinder

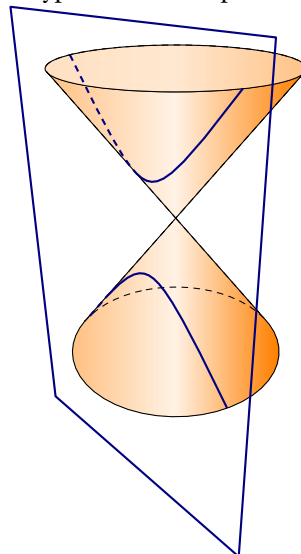


**Note** : We already have primitives for drawing cylinders, cones, and spheres without using facets. One of the advantages of defining these objects as polyhedra is that we can perform certain calculations on them, such as plane sections.

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

local g = ld.graph3d:new{window={-8,6,-9,9},bbox=false,
  viewdir={"central",45,65}, size={10,10}}
ld.Hiddenlinestyle = "dashed"; ld.Hiddenlines = true
local C1 = ld.cone(Origin,4*vecK,3,35,true)
local C2 = ld.cone(Origin, -4*vecK,3,35,true)
local P = {M(1,-1,-2),vecI} -- sectional plane
local I1 = g:Intersection3d(C1,P) -- intersection between cone C1 and plane P
local I2 = g:Intersection3d(C2,P) -- intersection between cone C2 and plane P
-- I1 and I2 are of the Edges type
g:Dcone(Origin,4*vecK,3,{color="orange"}); g:Dcone(Origin,-4*vecK,3,{color="orange"})
g:Lineoptions("solid","Navy",8)
g:Dedges(I1); g:Dedges(I2) -- drawing of edges I1 and I2
g:Dplane(P, vecK,14,9)
g:Show()
\end{luadraw}
```

Figure 12: Hyperbola: cone-plane intersection



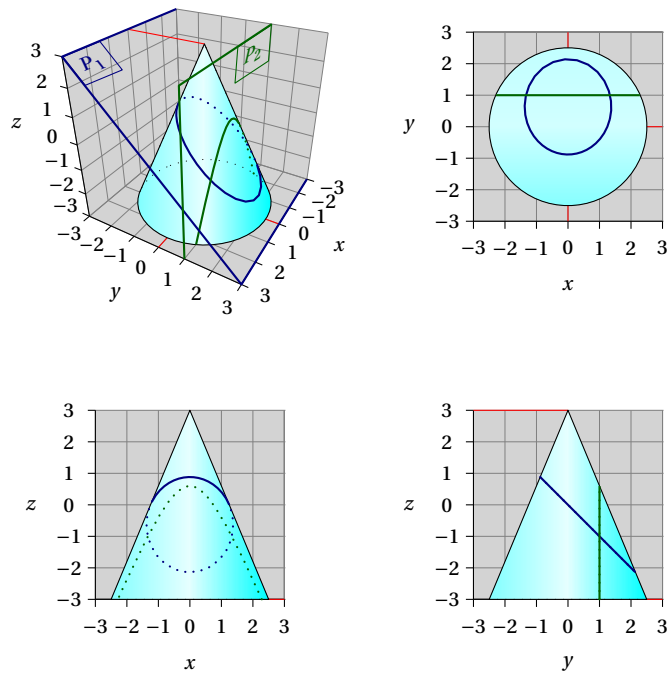
In this example, cones  $C_1$  and  $C_2$  are defined as polyhedra to determine their intersection with plane  $P$ , but not to draw them. The method `g:Intersection3d(C1, P)` returns the intersection of polyhedron  $C_1$  with plane  $P$  as a two-field table: one

field named *visible* that contains a 3D polygonal line representing the visible edges (segments) of the intersection (i.e., those that are on a visible facet of  $C_1$ ), and another field named *hidden* that contains a 3D polygonal line representing the hidden edges of the intersection (i.e., those that are on a non-visible facet of  $C_1$ ). The method **g:Dedges()** can be used to draw these types of objects.

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

local g = ld.graph3d:new{window3d={-3,3,-3,3,-3,3}, size={10,10}, margin={0,0,0,0}}
g:Labelsize("footnotesize")
local y0, R = 1, 2.5
local C = ld.cone(M(0,0,3),-6*vecK,R,35,true) -- open cone
local P1 = {M(0,0,0),vecK+vecJ} -- first plane
local P2 = {M(0,y0,0),vecJ} -- second plane
local I, I2
local dessin = function() -- one drawing per view
    g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
    I1 = g:Intersection3d(C,P1) -- intersection entre le cône C et les plans P1 et P2
    I2 = g:Intersection3d(C,P2) -- I1 and I2 are the Edges type
    g:Dpolyline3d( {M(0,-3,3),M(0,0,3),M(0,0,-3),M(3,0,-3)}, {M(0,0,-3),M(0,3,-3)}, "red,line width=0.4pt" )
    g:Dcone( M(0,0,3),-6*vecK,R, {color="cyan"})
    g:Dedges(I1, {hidden=true,color="Navy", width=8})
    g:Dedges(I2, {hidden=true,color="DarkGreen", width=8})
end
-- In the upper left corner, the view in space, we add the plans to the drawing
g:Saveattr(); g:Viewport(-5,0,0,5); g:Coordsystem(-7,6,-6,5,1); g:Setviewdir("central"); dessin()
g:Dpolyline3d( {M(-3,-3,3),M(3,-3,3),M(3,3,-3),M(-3,3,-3)}, "Navy,line width=0.8pt")
g:Dpolyline3d( {M(-3,y0,3),M(3,y0,3),M(3,y0,-3)}, "DarkGreen,line width=0.8pt")
g:Dlabel3d( "$P_1$",M(3,-3,3),{pos="SE",dir={-vecI,-vecJ+vecK},node_options="Navy, draw"})
g:Dlabel3d( "$P_2$",M(-3,y0,3),{pos="SW",dir={-vecI,vecK},node_options="DarkGreen,draw"})
g:Restoreattr()
-- top right, projection onto the xy-plane
g:Saveattr(); g:Viewport(0,5,0,5); g:Coordsystem(-6,6,-6,5,1); g:Setviewdir("x0y"); dessin()
g:Restoreattr()
-- bottom left, projection onto the xz-plane
g:Saveattr(); g:Viewport(-5,0,-5,0); g:Coordsystem(-6,6,-6,5,1); g:Setviewdir("x0z"); dessin()
g:Restoreattr()
-- bottom right, projection onto the yz-plane
g:Saveattr(); g:Viewport(0,5,-5,0); g:Coordsystem(-6,6,-6,5,1); g:Setviewdir("y0z"); dessin()
g:Restoreattr()
g:Show()
\end{luadraw}
```

Figure 13: Cone section with multiple views



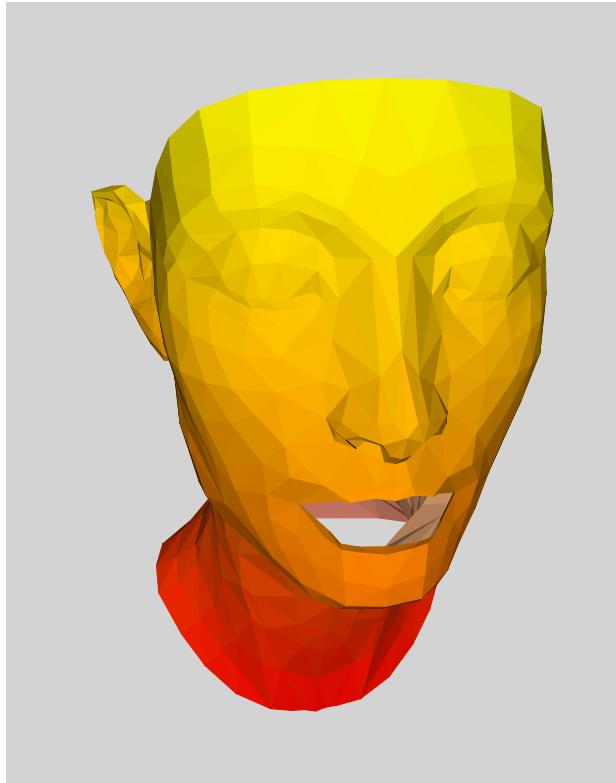
## 5) Reading from an obj file

The function `ld.read_obj_file(file)`<sup>1</sup> allows you to read the contents of the file *obj* designated by the string *(file)*. The function reads the vertex definitions (lines beginning with *v* ), and the lines defining the facets (lines beginning with *f* ). The other lines are ignored. The function returns a sequence consisting of the polyhedron, followed by a list of four real numbers  $\{x1, x2, y1, y2, z1, z2\}$  representing the 3D bounding box of the polyhedron.

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

<sup>1</sup>This function is a contribution by Christophe BAL.

Figure 14: Mask of Nefertiti



## 6) Surface in obj format

Two possible syntaxes:

1. **ld.obj\_surface(f, u1, u2, v1, v2 [, grid])** returns, in the format *obj*, the surface parameterized by the function  $\langle f \rangle: (u, v) \mapsto f(u, v) \in \mathbb{R}^3$ . The interval for the parameter  $u$  is given by  $\langle u1 \rangle$  and  $\langle u2 \rangle$ . The interval for the parameter  $v$  is given by  $\langle v1 \rangle$  and  $\langle v2 \rangle$ . The optional parameter  $\langle grid \rangle$  defaults to  $\{25, 25\}$ , and defines the number of points to calculate for the parameter  $u$  followed by the number of points to calculate for the parameter  $v$  (the values of  $u$  and  $v$  are equally distributed).
2. **ld.obj\_surface(f, mesh)** with  $\langle mesh \rangle = \{\{u_1, \dots, u_n\}, \{v_1, \dots, v_m\}\}$  returns the surface parameterized by the function  $\langle f \rangle: (u, v) \mapsto f(u, v) \in \mathbb{R}^3$ . The values of the parameters  $u$  and  $v$  are given by the argument  $\langle mesh \rangle$ ; they must be in strictly ascending order, but they are not necessarily equally distributed.

In both cases, the result is not a list of facets but a table with three fields:

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

- The *vertices* and *facets* fields are identical to the case of polyhedra: lists of vertices (3D points) and a list of facets with vertex numbers, except that here the facets are all **triangular**.
- The *normals* field is a list of unit 3D vectors representing normal vectors to the surface, one per vertex.

## 7) Drawing a List of Facets: Dfacet and Dmixfacet

There are two possible methods:

1. For a solid  $S$  in the form of a list of facets (with 3D points), the method is: `cmdlng:Dfacet(S, options)` where  $\langle S \rangle$  is the list of facets and  $\langle options \rangle$  is a table defining the options. These are:
  - **mode=ld.mShaded**: Sets the representation mode. The possible values are:
    - **ld.mWireframe**: Wireframe mode, draws only the edges.
    - **ld.mFlat** or **ld.mFlatHidden**: Draws the faces in a solid color, as well as the edges.
    - **ld.mShaded** or **ld.mShadedHidden**: The faces are drawn in shaded color based on their inclination, as well as the edges.

- `ld.mShadedOnly`: The faces are drawn in shaded color based on their inclination, but not the edges.
- `contrast=1`: This number allows you to accentuate or diminish the shade of the facet colors in the `ld.mShaded`, `ld.mShadedHidden`, and `ld.mShadedOnly` modes.
- `edgestyle=<current style>`: This is a string that defines the line style of the edges.
- `edgecolor=<current line color>`: This is a string that defines the edge color.
- `hiddenstyle=ld.Hiddenlinestyle`: This is a string that defines the line style of hidden edges. By default, this is the value contained in the global variable `ld.Hiddenlinestyle` (which itself is "dotted" by default).
- `hiddencolor=<current line color>`: This is a string that defines the color of hidden edges.
- `edgewidth=<current line width>`: This is the line thickness of edges in tenths of a point.
- `opacity=1`: This is a number between 0 and 1 that allows you to set transparency or not on the facets.
- `backcull=false`: When this boolean is `true`, facets considered invisible (normal vectors not directed towards the observer) are not displayed. This option is useful for convex polyhedra because it reduces the number of facets to draw.
- `twoside=true`: Boolean that defaults to true, meaning that both sides of the facets (inner and outer) are distinguished; the two sides will not have exactly the same color.
- `color="white"`: String defining the fill color of the facets.
- `usepalette=nil`: this option allows you to specify a color palette for painting the facets as well as a calculation mode, the syntax is: `usepalette={palette,mode}`, where *palette* designates a table of colors which are themselves tables of the form  $\{r, g, b\}$  where  $r$ ,  $g$  and  $b$  are numbers between 0 and 1. The argument *mode* can be :
  - one of the strings: "x", "y", "z". In the first case, for example, facets with the minimum x-coordinate at their centroid have the first color in the palette, facets with the maximum x-coordinate at their centroid have the last color in the palette, and for the others, the color is calculated based on the x-coordinate of the centroid by linear interpolation.
  - a function:  $\langle mode \rangle: f \rightarrow \text{mode}(f) \in \mathbb{R}$ , where  $f$  denotes a facet (a list of 3D points). Facets with the minimum value have the first color in the palette, those with the maximum value have the last color in the palette, and for the others, the color is calculated by linear interpolation.

2. For multiple facet lists in the same drawing, the method is:

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

where  $\langle S1 \rangle$ ,  $\langle S2 \rangle$ , ..., are facet lists, and  $\langle options1 \rangle$ ,  $\langle options2 \rangle$ , ..., are the corresponding options. The options in one facet list also apply to the following ones if they are not changed. These options are identical to the previous method.

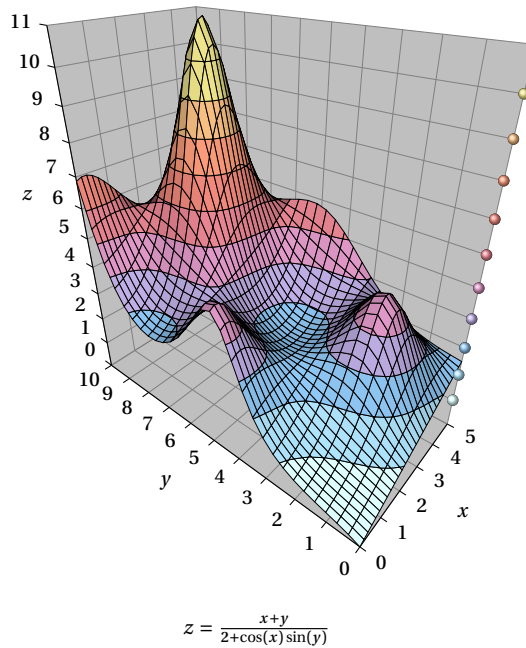
This method is useful for drawing multiple solids together, provided there are no intersections between the objects, as these are not handled here.

```
\begin{luadraw}{name=courbes_niv}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M, Z = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M, ld.cpx.Z
local cos, sin = math.cos, math.sin
local g = ld.graph3d:new{window3d={0,5,0,10,0,11}, adjust2d=true, size={10,10},
  viewdir={"central",220,60,15,M(2.5,5,5.5)}}
g:Labelsize("footnotesize")
local S = ld.cartesian3d(function(u,v) return (u+v)/(2+cos(u)*sin(v)) end,0,5,0,10,{30,30})
local n = 10 -- number of levels
local Colors = ld.getpalette(ld.palGasFlame,n,true) -- list of 10 colors in table format
local niv, S1 = {}
for k = 1, n do
  S1, S = ld.cutfacet(S,{M(0,0,k),-vecK}) -- section of S with the plane z=k
  ld.insert(niv,{S1, {color=Colors[k],mode=ld.mShaded,edgewidth=0.5}}) -- S1 is the part below the plane and S is
  -- above it
end
```



```
ld.insert(niv,{S, {color=Colors[n+1]}}) -- inserting the last level
-- niv is a list of the type {facets1, options1, facets2, options2, ...}
g:Dboxaxes3d({grid=true, gridcolor="gray",fillcolor="lightgray"})
g:Dmixfacet(table.unpack(niv))
for k = 1, n do
    g:Dballdots3d( M(5,0,k), ld.rgb(Colors[k]))
end
g:Dlabel("$z=\frac{x+y}{2+\cos(x)\sin(y)}$", Z((g:Xinf()+g:Xsup())/2, g:Yinf()), {pos="N"})
g:Show()
\end{luadraw}
```

Figure 15: Example of contour lines on a surface



## 8) Functions for Constructing Facet Lists

The following functions return a solid as a list of facets (with 3D points).

### surface()

Two possible syntaxes:

1. **ld.surface(f, u1, u2, v1, v2 [, grid])** returns the surface parameterized by the function  $\langle f \rangle: (u, v) \mapsto f(u, v) \in \mathbb{R}^3$ . The interval for the parameter  $u$  is given by  $\langle u1 \rangle$  and  $\langle u2 \rangle$ . The interval for the parameter  $v$  is given by  $\langle v1 \rangle$  and  $\langle v2 \rangle$ . The optional parameter  $\langle grid \rangle$  defaults to  $\{25, 25\}$ , and defines the number of points to calculate for the parameter  $u$  followed by the number of points to calculate for the parameter  $v$  (the values of  $u$  and  $v$  are equally distributed).
2. **ld.surface(f, mesh)** with  $\langle mesh \rangle = \{\{u_1, \dots, u_n\}, \{v_1, \dots, v_m\}\}$ , returns the surface parameterized by the function  $\langle f \rangle: (u, v) \mapsto f(u, v) \in \mathbb{R}^3$ . The values of the parameters  $u$  and  $v$  are given by the argument  $\langle mesh \rangle$ ; they must be in strictly ascending order, but they are not necessarily equally distributed.

There are two variants for surfaces:

### cartesian3d()

The function **ld.cartesian3d(f, x1, x2, y1, y2 [, grid, addwall])** returns the Cartesian surface with equation  $z = f(x, y)$  where  $\langle f \rangle: (x, y) \mapsto f(x, y) \in \mathbb{R}$ . The interval for  $x$  is given by  $\langle x1 \rangle$  and  $\langle x2 \rangle$ . The interval for  $y$  is given by  $\langle y1 \rangle$  and  $\langle y2 \rangle$ . The optional parameter  $\langle grid \rangle$  is  $\{25, 25\}$  by default; it defines the number of points to calculate for  $x$  followed by the number of points to

calculate for  $y$ . The parameter  $\langle addwall \rangle$  is 0 or "x", or "y", or "xy" (0 by default). When this option is set to "x" (or "xy"), the function returns, after the list of facets composing the surface, a list of separating facets (walls or partitions) between each "layer" of facets. A layer corresponds to two consecutive values of the parameter  $x^2$ . With the value "y" (or "xy"), it is a list of separating facets (walls) between each "layer" corresponding to two consecutive values of the parameter  $y^3$ . This option can be useful with the **g:Dscene3d()** method (only), because the separating partitions form a partition of space isolating the facets of the surface, which avoids unnecessary intersection calculations between them. This is particularly the case with non-convex surfaces.

For example, here is the code for figure 1:

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

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

### cylindrical\_surface()

The function **ld.cylindrical\_surface( $r, z, u1, u2, theta1, theta2$  [,  $grid, addwall$ ])** returns the surface parameterized as cylindrical by  $r(u, theta)$ ,  $theta$ ,  $z(u, theta)$ . The arguments  $\langle r \rangle$  and  $\langle z \rangle$  are therefore two real-valued functions of  $u$  and  $\theta$ . The interval for  $u$  is given by  $\langle u1 \rangle$  and  $\langle u2 \rangle$ . The interval for  $\theta$  is given by  $\langle theta1 \rangle$  and  $\langle theta2 \rangle$  (in radians). The optional parameter  $\langle grid \rangle$  is {25,25} by default; it defines the number of points to calculate for  $u$  followed by the number of points to calculate for  $v$ . The parameter  $\langle addwall \rangle$  is 0 or "v" or "z" or "vz" (0 by default). When this option is "v" ou "vz", the function returns, after the list of facets composing the surface, a list of separating facets (walls or partitions) between each "layer" of facets, a layer corresponds to two consecutive values of the angle  $\theta^4$ . When this option is set to "z" ou "vz", the function returns, after the list of facets composing the surface, a list of separating facets (walls or partitions) between each "layer" of facets. A layer corresponds to two consecutive values of the dimension  $z^5$ , the values of  $z$  are calculated from the values of the parameter  $u$  and with the value  $\langle theta1 \rangle$ . This is useful when  $z$  only depends on  $u$  (and therefore not on  $\theta$ ). This option can be useful with the **g:Dscene3d()** method (only), because the separating partitions form a partition of space isolating the surface facets, which avoids unnecessary intersection calculations between them. This is particularly the case with non-convex surfaces.

```
\begin{luadraw}{name=surface_with_addWall}
local ld = luadraw
local pi, ch, sh = math.pi, math.cosh, math.sinh
local O = ld.pt3d.Origin
local g = ld.graph3d:new{window3d={-4,4,-4,4,-5,5}, window={-10,10,-4,4}, size={10,10}, viewdir={60,60}}
g:Labelsize("footnotesize")
local S,wall = ld.cartesian3d(function(x,y) return x^2-y^2 end,-2,2,-2,2,nil,"xy")
g:Saveattr(); g:Viewport(-10,0,-4,4); g:Coordsystem(-4.5,4.5,-4.5,4.75)
g:Dscene3d(
  g:addWall(wall), -- 2 facet cutouts with this instruction, and 529 facet cutouts without it
  g:addFacet(S,{color="SteelBlue"}),
  g:addAxes(0,{arrows=1}) )
g:Restoreattr()
```

<sup>2</sup>These partitions are actually planes with the equation  $x = \text{constant}$

<sup>3</sup>These partitions are actually planes with the equation  $y = \text{constant}$

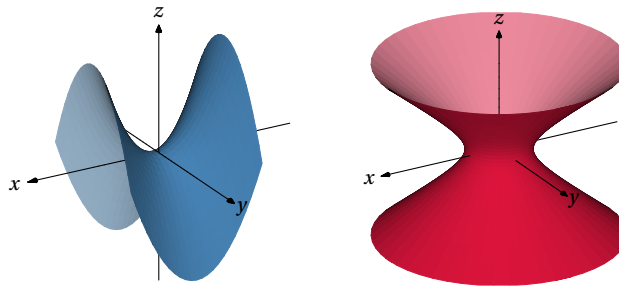
<sup>4</sup>These partitions are in fact planes of equation  $\theta = \text{constant}$

<sup>5</sup>These partitions are actually planes with the equation  $z = \text{constant}$

```

g:Saveattr(); g:Viewport(0,10,-4,4); g:Coordsystem(-5,5,-5,5)
local r = function(u,v) return ch(u) end
local z = function(u,v) return sh(u) end
S,wall = ld.cylindrical_surface(r,z,-2,2,-pi,pi,{25,51},"zv")
g:Dscene3d(
  g:addWall(wall), -- 13 facet cutouts with this instruction, and more than 17000 facet cutouts without it ...
  g:addFacet(S,{color="Crimson"}),
  g:addAxes(0,{arrows=1}) )
g:Restoreattr()
g:Show()
\end{luadraw}

```

Figure 16: Surfaces using the *addwall* option**curve2cone()**

The function **ld.curve2cone(f, t1, t2, S, options)** constructs a cone with vertex  $\langle S \rangle$  (3D point) and the base curve parametrized by  $\langle f \rangle: t \mapsto f(t) \in \mathbb{R}^3$  on the interval defined by  $\langle t1 \rangle$  et  $\langle t2 \rangle$ . The argument  $\langle options \rangle$  is an table whose fields define the options, which are (with thier default value):

- **nbdots=15**: minimum number of points on the curve to calculate.
- **ratio=0**: number representing the homothety ratio (centered at vertex  $\langle S \rangle$ ) to construct the other part of the cone, with the value 0 ther is no second part.
- **nbdiv=0**: positive integer indicating the number of times the interval between two consecutive values of the parameter  $t$  can be bisected (dichotomized) when the corresponding points are too far apart.
- **obj=false**: with the value **false** this function builds a list of facets, with the value **true** it builds a table with three fields:  $\{\text{vertices}=\{\text{3D points}\}, \text{facets}=\{\{\text{index1}, \dots\}, \dots\}, \text{normals}=\{\text{3D vectors}\}\}$ .

The function returns a sequence:

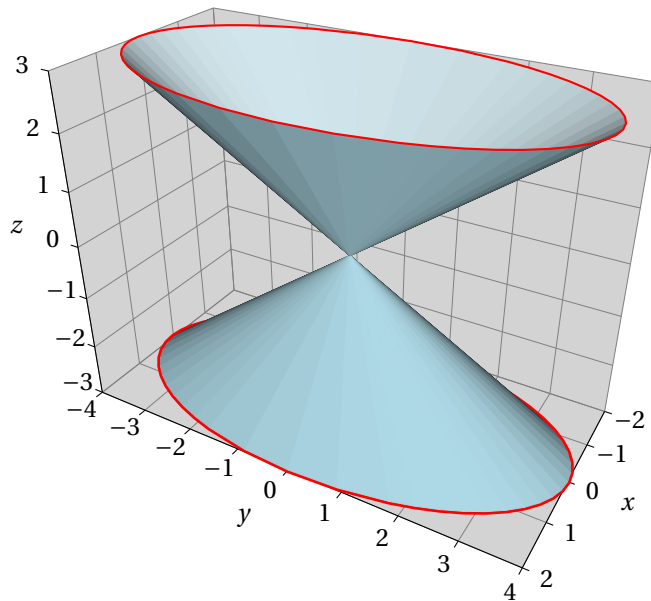
1. the list of facets or the table in *obj* format, followed by
2. a 3D polygonal line representing the cone's edges.

```

\begin{luadraw}{name=curve2cone}
local ld = luadraw
local O, M = ld.pt3d.Origin, ld.pt3d.M
local cos, sin, pi = math.cos, math.sin, math.pi
local g = ld.graph3d:new{ window3d={-2,2,-4,4,-3,3}, window={-5.5,5.5,-5.5,5},
  size={10,10}, viewdir="central"}
local f = function(t) return M(2*cos(t),4*sin(t),-3) end -- ellipse in the plane z=-3
local C, bord = ld.curve2cone(f,-pi,pi,0,{nbdiv=2, ratio=-1})
g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
g:Dpolyline3d(bord[1],"red,line width=2.4pt") -- bottom edge
g:Dfacet(C, {mode=ld.mShadedOnly,color="LightBlue"}) -- cone
g:Dpolyline3d(bord[2],"red,line width=0.8pt") -- upper edge
g:Show()
\end{luadraw}

```

Figure 17: Elliptical Cone Example

**curve2cylinder()**

The function **ld.curve2cylinder(f, t1, t2, V [, options])** constructs a cylinder with axis directed by the vector  $\langle V \rangle$  (3D point) and with a base parameterized by  $\langle f \rangle: t \mapsto f(t) \in \mathbb{R}^3$  on the interval defined by  $\langle t1 \rangle$  and  $\langle t2 \rangle$ . The second base is the translation of the first with the vector  $V$ . The argument  $\langle options \rangle$  is an table whose fields define the options, which are (with thier default value):

- **nbdots=15**: minimum number of points on the curve to calculate.
- **nbdiv=0**: positive integer indicating the number of times the interval between two consecutive values of the parameter  $t$  can be bisected (dichotomized) when the corresponding points are too far apart.
- **obj=false**: with the value **false** this function builds a list of facets, with the value **true** it builds a table with three fields: {vertices={3D points}, facets={{index1,...},...}, normals={3D vectors}}.

The function returns a sequence:

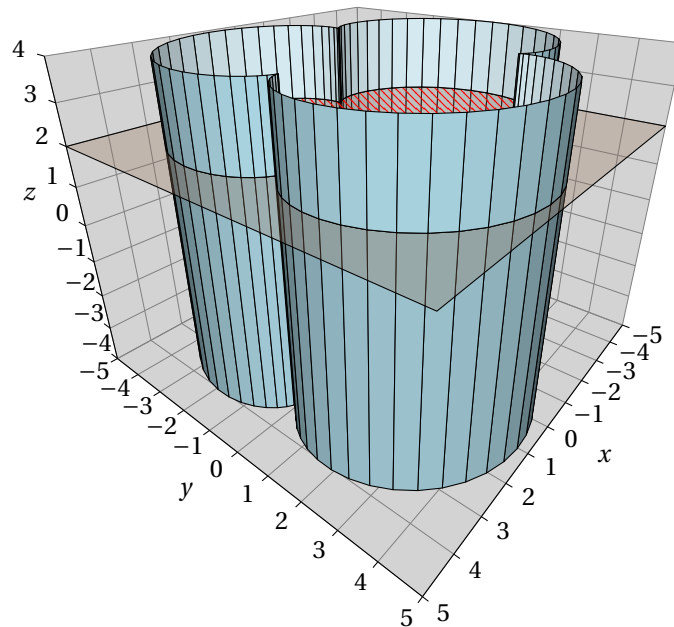
1. the list of facets or the table in *obj* format, followed by
2. a 3D polygonal line representing the cylinder's edges.

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

local g = ld.graph3d:new{ window3d={-5,5,-5,5,-4,4},window={-9,8,-10.5,5.5},
    viewdir={"central",39,64}, size={10,10}}
local f = function(t) return M(4*cos(t)-cos(4*t),4*sin(t)-sin(4*t),-4) end -- courbe dans le plan z=-3
local V = 8*vecK
local C = ld.curve2cylinder(f,-pi,pi,V,{nbdots=25,nbdiv=2})
local plan = {M(0,0,2), -vecK} -- cutting plane z=2
local C1, C2, section = ld.cutfacet(C,plan)
g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
g:Dfacet(C1, {mode=ld.mShaded,color="LightBlue"}) -- part below the plane
g:Dfacet(g:Plane2facet(plan), {opacity=0.3,color="Chocolate"}) -- draw the plane as a facet
g:Filloptions("fdiag","red"); g:Dpolyline3d(section) -- drawing of the section
g:Dfacet(C2, {mode=3,color="LightBlue"}) -- part of the cylinder above the plane
```

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

Figure 18: Section of a non-circular cylinder



### `line2tube(); section2tube()`

The function `ld.line2tube(L, r [, options])` constructs (as a list of facets) a tube centered on  $\langle L \rangle$ , which must be a 3D polygonal line (list of 3D points or list of lists of 3D points). The argument  $\langle r \rangle$  represents the radius of this tube. The argument  $\langle options \rangle$  is an table whose fields define the options, which are (with thier default value):

- `nbfacet=3`: number of lateral facets of the tube.
- `close=false`: boolean indicating whether the polygonal line should be closed.
- `hollow=false`: boolean indicating whether both ends of the tube should be open or not. When the `close` option is set to `true`, the `hollow` option is automatically set to `true`.
- `addwall=0`: number that is 0 or 1. When this option is 1, the function returns, after the tube, a list of separating facets (walls) between each "section" of the tube, which can be useful with the `g:Dscene3d()` method (only).
- `obj=false`: with the value `false` this function returns a list of facets, with the value `true` it returns a table with three fields: `{vertices={3D points}, facets={{index1,...},...}, normals={3D vectors}}`.

The function `ld.section2tube(section, L [, options])` also constructs a tube centered on  $\langle L \rangle$ , which must be a list of 3D points. The argument  $\langle section \rangle$  must be a facet centered on the first point of  $\langle L \rangle$ ; it represents a section of the tube to be constructed. The argument  $\langle options \rangle$  is an table whose fields define the options, which are (with thier default value):

- `nbfacet=3`: number of lateral facets of the tube.
- `close=false`: boolean indicating whether the polygonal line should be closed.
- `hollow=false`: boolean indicating whether both ends of the tube should be open or not. When the `close` option is set to `true`, the `hollow` option is automatically set to `true`.
- `addwall=0`: number that is 0 or 1. When this option is 1, the function returns, after the tube, a list of separating facets (walls) between each "section" of the tube, which can be useful with the `g:Dscene3d()` method (only).
- `obj=false`: with the value `false` this function returns a list of facets, with the value `true` it returns a table with three fields: `{vertices={3D points}, facets={{index1,...},...}, normals={3D vectors}}`.

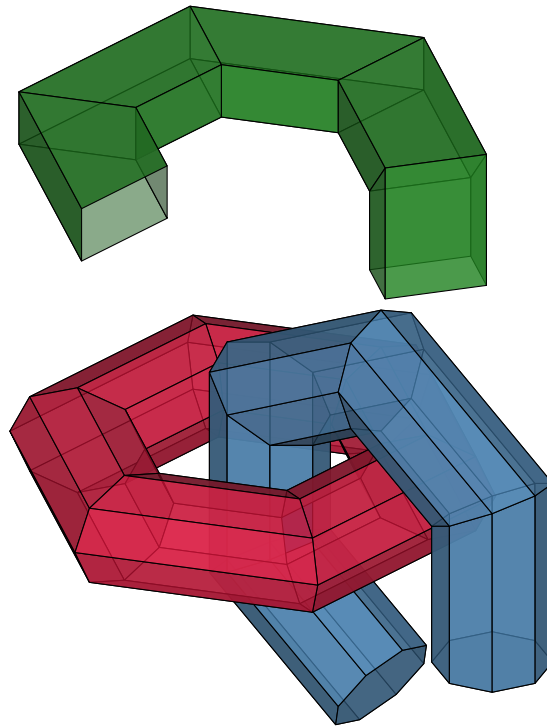
```

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

local g = ld.graph3d:new{window={-5,6,-4.5,8}, viewdir={45,60}, margin={0,0,0,0}, size={10,10}}
local L1 = ld.map(pt3d.toPoint3d,ld.polyreg(0,3,6)) -- regular hexagon in the xOy plane, with center O and vertex
-- M(3,0,0)
local L2 = ld.shift3d(ld.rotate3d(L1,90,{Origin,vecJ}),3*vecJ)
local L3 = ld.shift3d(ld.reverse(L1),6*vecK)
L3[6] = L3[5]-2*vecK -- modification of the last point
local section = ld.shift3d({M(2,0,0.5),M(4,0,0.5),M(4,0,-0.5),M(2,0,-0.5)},6*vecK)
local T1 = ld.line2tube(L1,1,{nbfacet=8,close=true}) -- tube 1 closed
local T2 = ld.line2tube(L2,1,{nbfacet=8}) -- tube 2 not closed
local T3 = ld.section2tube(section, L3,{hollow=true})
g:Dmixfacet( T1, {color="Crimson",opacity=0.8}, T2, {color="SteelBlue"}, T3, {color="ForestGreen"} )
g:Show()
\end{luadraw}

```

Figure 19: Example with line2tube and section2tube



### rotcurve()

The function **ld.rotcurve(p, t1, t2, axe, angle1, angle2 [, options])** constructs, as a list of facets, the surface swept by the curve parameterized by  $\langle p \rangle: t \mapsto p(t) \in \mathbf{R}^3$  over the interval defined by  $\langle t1 \rangle$  and  $\langle t2 \rangle$ , by rotating it around  $\langle axis \rangle$  (which is a table of the form {3D point, 3D vector} representing an oriented line in space), by an angle ranging from  $\langle angle1 \rangle$  (in degrees) to  $\langle angle2 \rangle$ . The argument  $\langle options \rangle$  is an table whose fields define the options, which are (with thier default value):

- **grid={25,25}**: table consisting of two numbers, the first being the number of points calculated for the  $t$  parameter, and the second being the number of points calculated for the angular parameter.
- **addwall=0**: number equal to 0, 1, or 2. When this option is set to 1, the function returns, after the surface, a list of separating facets (walls) between each "layer" of facets (a layer corresponds to two consecutive values of the  $t$  parameter), and with a value of 2, it is a list of separating facets (walls) between each rotation "slice" (a layer corresponds to two consecutive values of the angular parameter; this is useful when the curve is in the same plane as the rotation axis). This option can be useful with the **g:Dscene3d()** method (only).

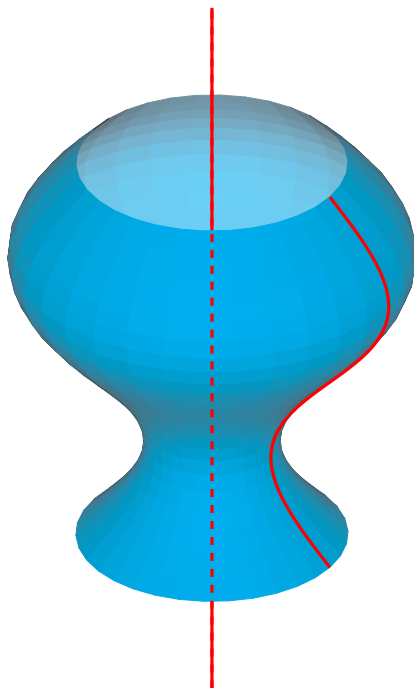
- **obj=false** : with the value **false** this function returns a list of facets, with the value **true** it returns a table with three fields: {vertices={3D points}, facets={{index1,...},...}, normals={3D vectors}}.

```

\begin{luadraw}{name=rotcurve}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M
local cos, sin, pi = math.cos, math.sin, math.pi
local g = ld.graph3d:new{viewdir={30,60},size={10,10}}
local p = function(t) return M(0,sin(t)+2,t) end -- curve in the plane yOz
local axe = {Origin,vecK}
local S = ld.rotcurve(p,pi,-pi,axe,0,360,{grid={25,35}})
local visible, hidden = g:Classifyfacet(S)
g:Dfacet(hidden, {mode=ld.mShadedOnly,color="cyan"})
g:Dline3d(axe,"red,line width=1.2pt")
g:Dfacet(visible, {mode=5,color="cyan"})
g:Dline3d(axe,"red,line width=1.2pt,dashed")
g:Dparametric3d(p,{t={-pi,pi}},draw_options="red,line width=1.2pt")
g:Show()
\end{luadraw}

```

Figure 20: Example with rotcurve



**Note** : If the surface orientation does not seem correct, simply swap the parameters  $\langle t1 \rangle$  and  $\langle t2 \rangle$ , or  $\langle angle1 \rangle$  and  $\langle angle2 \rangle$ .

### rotline()

The function **ld.rotline(L, axe, angle1, angle2 [, options])** constructs, as a list of facets, the surface swept by the list of 3D points  $\langle L \rangle$  by rotating it around  $\langle axis \rangle$  (which is a table of the form {3D point, 3D vector} representing an oriented line in space), through an angle ranging from  $\langle angle1 \rangle$  (in degrees) to  $\langle angle2 \rangle$ . The argument  $\langle options \rangle$  is an table whose fields define the options, which are (with thier default value):

- **nbdots=25**: number of points calculated for the angular parameter.
- **close=false**: boolean indicating whether  $\langle L \rangle$  should be closed.
- **addwall=0**: number equal to 0, 1, or 2. When this option is set to 1, the function returns, after the surface, a list of separating facets (walls) between each "layer" of facets (a layer corresponds to two consecutive values of the

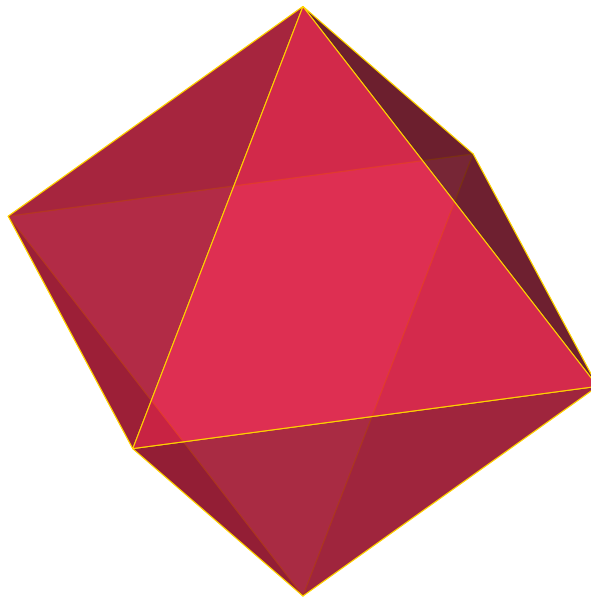


$t$  parameter), and with a value of 2, it is a list of separating facets (walls) between each rotation "slice" (a layer corresponds to two consecutive values of the angular parameter; this is useful when the curve is in the same plane as the rotation axis). This option can be useful with the **g:Dscene3d()** method (only).

- **obj=false** : with the value **false** this function returns a list of facets, with the value **true** it returns a table with three fields: {vertices={3D points}, facets={{index1,...},...}, normals={3D vectors}}.

```
\begin{luadraw}{name=rotline}
local ld = luadraw
local pt3d = ld.pt3d
local M = pt3d.M
local g = ld.graph3d:new{window={-4,4,-4,4},size={10,10}}
local L = {M(0,0,4),M(0,4,0),M(0,0,-4)} -- list of points in the yz plane
local axe = {pt3d.Origin, pt3d.vecK}
local S = ld.rotline(L,axe,0,360,{nbdots=5}) -- point 1 and point 5 are confused
g:Dfacet(S,{color="Crimson",edgecolor="Gold",opacity=0.8})
g:Show()
\end{luadraw}
```

Figure 21: Example with rotline



## 9) Edges of a solid

An "edge" object is a table with two fields: one field named *visible* that contains a 3D polygonal line corresponding to the visible edges, and another field named *hidden* that contains a 3D polygonal line corresponding to the hidden edges.

- The method **g:Edges(P)**, where  $\langle P \rangle$  is a polyhedron, returns the edges of  $\langle P \rangle$  as an "edge" object. An edge of  $\langle P \rangle$  is visible when it belongs to at least one visible face.
- The method **g:Intersection3d(P, plane)**, where  $\langle P \rangle$  is a polyhedron or a list of facets, returns as an "edge" object the intersection between  $\langle P \rangle$  and the plane represented by  $\langle plane \rangle$  (it is a table of the form  $\{A,u\}$  where  $A$  is a point on the plane and  $u$  is a normal vector, so they are two 3D points).
- The method **g:Dedges(edges, options)** allows you to draw  $\langle edges \rangle$ , which must be an "edge" object. The argument  $\langle options \rangle$  is a table whose fields define the options, which are (with thier default value):
  - **hidden=false**: Boolean indicating whether hidden edges should be drawn.
  - **visible=true**: Boolean indicating whether visible edges should be drawn.



- `clip=false`: Boolean indicating whether edges should be clipped by the 3D window.
- `hiddenstyle=ld.Hiddenlinestyle`: String defining the line style of hidden edges. By default, this option contains the value of the global variable `ld.Hiddenlinestyle` (which defaults to `"dotted"`).
- `hiddencolor=color`: String defining the color of hidden edges. By default, this option contains the same color as the `color` option.
- `style=<current line style>`: String defining the line style of visible edges.
- `color=<current line color>`: String defining the color of the visible edges.
- `width=<current width>`: Number representing the line thickness of the edges (in tenths of a point).

- **Complement:**

- The function `ld.facetedges(F)`, where  $\langle F \rangle$  is a list of facets or a polyhedron, returns a list of 3D segments representing all the edges of  $\langle F \rangle$ . The result is not an "edge" object, and is drawn with the `g:Dpolyline3d()` method. Of course, each edge only appears once in the list.
- The function `ld.facetvertices(F)`, where  $\langle F \rangle$  is a list of facets or a polyhedron, returns the list of all vertices of  $\langle F \rangle$  (3D points).

## 10) Methods and functions applying to facets or polyhedra

- The method `g:Isvisible(F)`, where  $\langle F \rangle$  denotes a facet (list of at least 3 coplanar and non-aligned 3D points), returns true if facet  $\langle F \rangle$  is visible (normal vector directed towards the observer). If  $A$ ,  $B$ , and  $C$  are the first three points of  $\langle F \rangle$ , the normal vector is calculated by performing the vector product  $\vec{AB} \wedge \vec{AC}$ .
- The method `g:Classifyfacet(F)`, where  $\langle F \rangle$  is a list of facets or a polyhedron, returns **two** lists of facets: the first is the list of visible facets, and the second, the list of invisible facets.
- The method `g:Sortfacet(F [, backcull])`, where  $\langle F \rangle$  is a list of facets, returns this list of facets sorted from furthest to closest to the observer. The optional argument  $\langle backcull \rangle$  is a boolean that defaults to `false`; when it is `true`, non-visible facets are excluded from the result (only visible facets are then returned after being sorted). The calculation of a facet's distance is based on its center of gravity. The so-called "painter" technique consists of displaying the facets from furthest to closest, therefore in the order of the list returned by this function (the displayed result, however, is not always correct depending on the size and shape of the facets).
- The method `g:Sortpolyfacet(P [, backcull])`, where  $\langle P \rangle$  is a polyhedron, returns the list of facets of  $\langle P \rangle$  (facets with 3D points) sorted from furthest to closest to the observer. The optional argument  $\langle backcull \rangle$  is a boolean that defaults to `false`; when it is `true`, invisible facets are excluded from the result, as in the previous method. These two sorting methods are used by the methods for drawing polyhedrons or facets (`g:Dpoly()`, `g:Dfacet()` and `g:Dmixfacet()`).
- The method `g:Outline(P)`, where  $\langle P \rangle$  is a polyhedron, returns the "outline" of  $\langle P \rangle$  as a two-field table. One field, named *visible*, contains a 3D polygonal line representing the "edges" (segments) belonging to a single facet, which is visible, or to two facets, one visible and one hidden; the other field, named *hidden*, contains a 3D polygonal line representing the "edges" belonging to a single facet, which is hidden.
- The function `ld.border(P)`, where  $\langle P \rangle$  is a polyhedron or a list of facets, returns a 3D polygonal line corresponding to the edges belonging to a single facet of  $\langle P \rangle$  (these edges are placed "end to end" to form a polygonal line).
- The function `ld.getfacet(P, list)`, where  $\langle P \rangle$  is a polyhedron, returns the list of facets of  $\langle P \rangle$  (with 3D points) whose number appears in the table  $\langle list \rangle$ . If the argument  $\langle list \rangle$  is not specified, the list of all facets of  $\langle P \rangle$  is returned (in this case, it is the same as `ld.poly2facet(P)`).
- The function `ld.facet2plane(L)`, where  $\langle L \rangle$  is either a facet or a list of facets, returns either the plane containing the facet or the list of planes containing each of the facets of  $\langle L \rangle$ . A plane is a table of the type  $\{A, u\}$  where  $A$  is a point on the plane and  $u$  is a normal vector to the plane (i.e., two 3D points).

- The function `ld.reverse_face_orientation(F)` where  $\langle F \rangle$  is either a facet, a list of facets, or a polyhedron, returns a result of the same nature as  $\langle F \rangle$  but in which the order of the vertices of each facet has been reversed. This can be useful when the orientation of space has been changed.

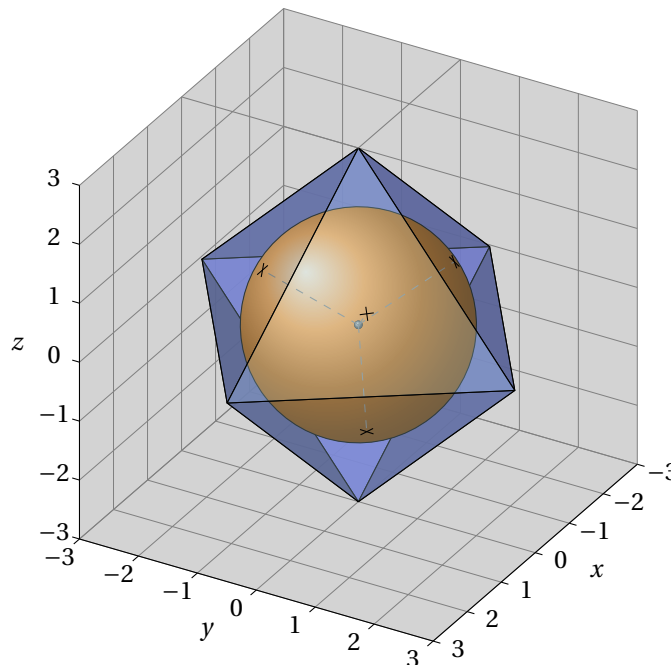
```

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

local poly = require "luadraw_polyhedrons"
local g = ld.graph3d:new{ window3d={-3,3,-3,3,-3,3}, size={10,10}}
local P = poly.octahedron(Origin,M(0,0,3)) -- polyhedron defined in the luadraw_polyhedrons module
P = ld.rotate3d(P,-10,{Origin,vecK}) -- rotate3d on a polyhedron returns a polyhedron
local V, H = g.Classifyfacet(P) -- V for visible facets, H for hidden
local S = ld.map(function(p) return {ld.proj3d(Origin,p),p[2]} end, ld.facet2plane(V) )
-- S contains the list of: {projected, normal vector} (projected from Origin onto the visible faces)
local R = pt3d.abs(S[1][1]) -- sphere radius
g:Dboxaxes3d({grid=true, gridcolor="gray", fillcolor="LightGray"})
g:Dfacet(H, {color="blue",opacity=0.9}) -- drawing of non-visible facets
g:Dsphere(Origin,R,{mode=ld.mBorder,color="orange"}) -- drawing of the sphere
g:Dballdots3d(Origin,"gray",0.75) -- center of the sphere
for _,D in ipairs(S) do -- segments connecting the origin to the projected
    g:Dpolyline3d( {Origin,D[1]}, "dashed,gray")
end
g:Dfacet(V,{opacity=0.4, color="LightBlue"}) -- visible facets of the octahedron
g:Dcrossdots3d(S,nil,0.75) -- drawing of the projections on the faces
g:Dpolyline3d( {M(0,-3,3), M(0,0,3), M(-3,0,3)}, "gray")
g:Show()
\end{luadraw}

```

Figure 22: Sphere inscribed in an octahedron with the center projected onto the faces



## 11) Cutting a solid: cutpoly and cutfacet

- The function `ld.cutpoly(P, plane [, close])` cuts the polyhedron  $\langle P \rangle$  with the plane  $\langle plane \rangle$  (a table of type  $\{A,n\}$  where  $A$  is a point on the plane and  $n$  is a vector normal to the plane). The function returns three things: the part located in the half-space containing the vector  $n$  (in the form of a polyhedron), followed by the part located in the other half-space (still in the form of a polyhedron), followed by the section in the form of a facet oriented by  $-n$ . When the

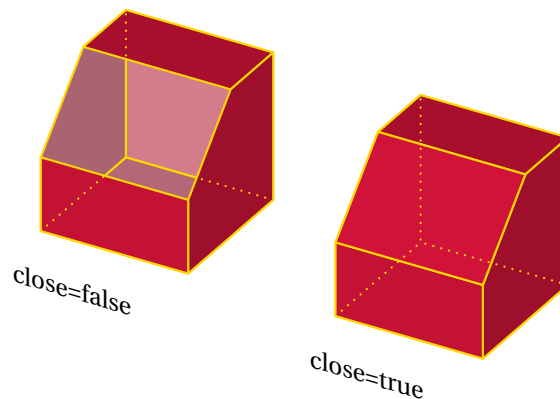
optional argument  $\langle close \rangle$  is **true**, the section is added to both resulting polyhedra, which closes them (**false** by default).

**Note:** When the polyhedron  $\langle P \rangle$  is not convex, the section result is not always correct.

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

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

Figure 23: Cube cut by a plane (cutpoly), with  $close=false$  and with  $close=true$



- The function **ld.cutfacet(F, plane [, close])**, where  $\langle F \rangle$  is a facet, a list of facets, or a polyhedron, does the same thing as the previous function except that this function returns lists of facets and not polyhedra. This function was used in the contour line example in Figure 15.

## 12) Clipping Facets with a Convex Polyhedron: clip3d

The function **ld.clip3d(S, P [, exterior])** clips the solid  $\langle S \rangle$  (which is a list of facets or a polyhedron) with the convex solid  $\langle P \rangle$  (which is a list of facets or a polyhedron) and returns the resulting list of facets. The optional argument  $\langle exterior \rangle$  is a boolean that defaults to **false**. In this case, the part of  $\langle S \rangle$  that is interior to  $\langle P \rangle$  is returned; otherwise, the part of  $\langle S \rangle$  that is exterior to  $\langle P \rangle$  is returned.

**Note:** The result is not always satisfactory for the exterior part.

**Special case** : Clipping a list of facets  $S$  (or polyhedron) with the current 3D window can be done with this function as follows:

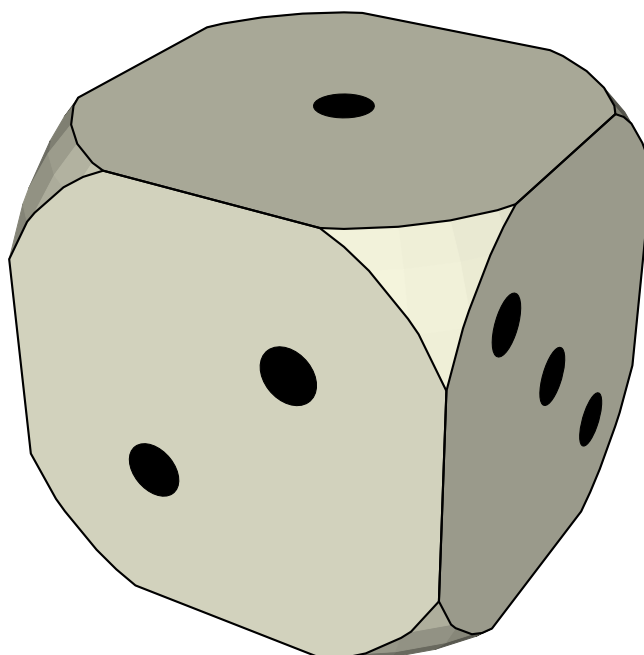
$$S = \text{ld.clip3d}(S, \text{g:Box3d}())$$

Indeed, the `g:Box3d()` method returns the current 3D window as a parallelepiped.

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

local g = ld.graph3d:new(window={-3,3,-3,3},size={10,10}, viewdir="central")
local S = ld.sphere(Origin,3)
local C = ld.parallelelep(M(-2,-2,-2),4*vecI,4*vecJ,4*vecK)
local C1 = ld.clip3d(S,C) -- sphere clipped by the cube
local C2 = ld.clip3d(C,S) -- cube clipped by the sphere
local V = g:Classifyfacet(C2) -- visible facets of C2
g:Dfacet( ld.concat(C1,C2), {color="Beige",mode=ld.mShadedOnly,backcull=true} ) -- only visible faces
g:Dpolyline3d(V,true,"line width=0.8pt") -- outline of the visible faces of C2
local A, B, C, D = M(2,-2,-2), M(2,2,2), M(-2,2,-2), M(0,0,2) -- drawing black dots
g:Filloptions("full","black")
g:Dcircle3d( D,0.25,vecK); g:Dcircle3d( (2*A+B)/3,0.25,vecI)
g:Dcircle3d( (A+2*B)/3,0.25,vecI); g:Dcircle3d( (3*B+C)/4,0.25,vecJ)
g:Dcircle3d( (B+C)/2,0.25,vecJ); g:Dcircle3d( (B+3*C)/4,0.25,vecJ)
g:Show()
\end{luadraw}
```

Figure 24: Example with clip3d: constructing a die from a cube and a sphere



### 13) Clip a plane with a convex polyhedron: clipplane

The function `ld.clipplane(plane, P)`, where the argument  $\langle plane \rangle$  is a table of the form  $\{A, n\}$  representing the plane passing through  $A$  (3D point) and normal vector  $n$  (non-zero 3D point), and  $\langle P \rangle$  is a convex polyhedron, returns the section of the polyhedron through the plane, if it exists, in the form of a facet (list of 3D points) oriented by  $n$ .

## V The Dscene3d Method

### 1) The Principle, the Limitations

The major flaw of the `g:Dpoly()`, `g:Dfacet()`, and `g:Dmixfacet()` methods is that they do not handle possible intersections between facets of different solids. Not to mention that sometimes, even for a simple convex polyhedron, the painter's

algorithm does not always produce the correct result (because the facets are sorted only by their center of gravity). Furthermore, these methods only allow you to draw facets.

The principle of the **g:Dscene3d()** method is to classify the 3D objects to be drawn (facets, polygonal lines, points, labels, etc.) in a tree (which represents the scene). At each node of the tree, there is a 3D object, let's call it *A*, and two descendants. One of the descendants will contain the 3D objects that are in front of object *A* (i.e., closer to the observer than *A*), and the other descendant will contain the 3D objects that are behind object *A* (i.e., further from the observer than *A*).

In particular, to classify a facet *B* with respect to a facet *A* that is already in the tree, we proceed as follows: we split facet *B* with the plane containing facet *A*, which generally results in two "half" facets, one that will be in front of *A* (the one in the half-space "containing" the observer), and the other that will therefore be behind *A*.

This method is effective but has limitations because it can cause the number of facets in the tree to explode, thus increasing its size exponentially. This can make it prohibitive to use this method when there are many facets (long computation time<sup>6</sup>, excessively large \*.tkz file size, excessively long drawing time per TikZ). However, it is very effective when there are few facets, and therefore few facet intersections (convex objects with few facets). Furthermore, it is possible to draw below and above the 3D scene, i.e., before using the **g:Dscene3d()** method, and after its use.

This method should therefore be reserved for very simple scenes. For complex 3D scenes, the vector format is not suitable, so it is better to turn to other tools like POV-Ray, Blender, or WebGL.

## 2) Construction of a 3D scene

The **g:Dscene3d(...)** method allows this construction. It takes as arguments the 3D objects that will constitute this scene one after the other. These 3D objects are themselves created using dedicated methods that will be detailed later. In the current version, these 3D objects can be:

- polyhedra,
- lists of facets (with 3D points),
- 3D polygonal lines,
- 3D points,
- labels,
- axes,
- planes, lines,
- angles,
- circles, and arcs.

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

local g = ld.graph3d:new{viewdir={"central",-10,60}, window={-5,6.5,-6.5,6},bg="gray", size={10,10}}
local P1 = ld.planeEq(1,1,1,-2) -- plane of equation x+y+z-2=0
local P2 = {Origin, vecK-vecJ} -- plane passing through O and normal to (1,1,1)
local D = ld.interPP(P1,P2) -- line of intersection between P1 and P2 (D = {A,u})
local posD = D[1]+1.85*D[2] -- to place the label
ld.Hiddenlines = true; ld.Hiddenlinestyle = "dotted" -- display of hidden lines as dotted lines
g:Dscene3d(
  g:addPlane(P1, {color="Crimson",edge=true,edgecolor="Pink",edgewidth=8}), --addition of plane P1
  g:addPlane(P2, {color="ForestGreen",edge=true,edgecolor="Pink",edgewidth=8}), --addition of plane P2
  g:addLine(D, {color="Navy",edgewidth=12}), --addition of line D
  g:addAxes(Origin, {arrows=1, color="Gold",width=8}), -- addition of arrowed axes
  g:addLabel( -- adding labels; these could have been added on top of the scene.
    "$D=P_1\cap P_2$",posD,{color="Navy"},
```

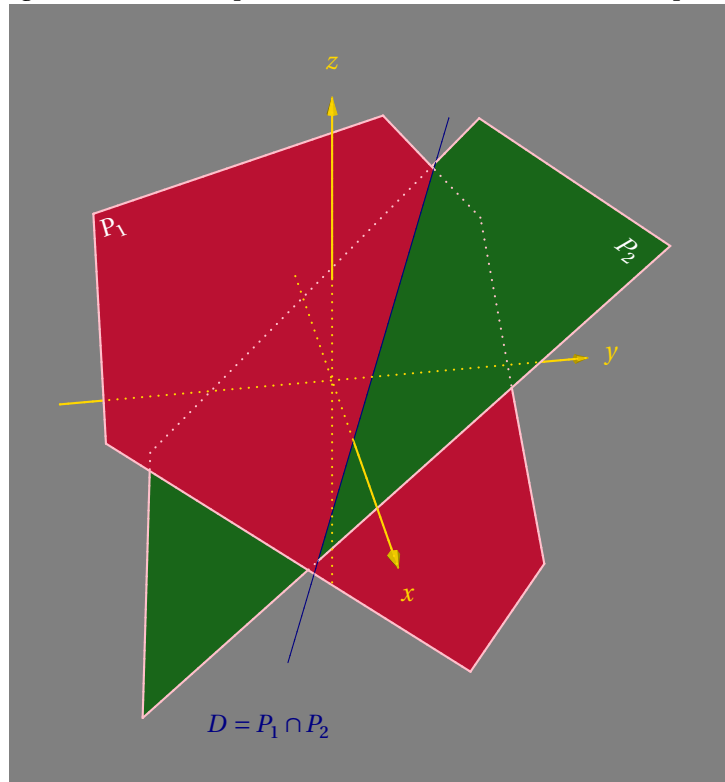
<sup>6</sup>Lua is an interpreted language, so execution is generally slower than with a compiled language.

```

"$P_2$", M(3,0,0)+3.5*M(0,1,1)+0.2*vecI,{color="white",dir={vecI,vecJ+vecK}},
"$P_1$",M(2,0,0)+1.85*M(-1,-1,2)-1.5*M(-1,1,0), {dir={M(-1,1,0),M(-1,-1,2)}} )
)
g:Show()
\end{luadraw}

```

Figure 25: First example with Dscene3d: intersection of two planes



### 3) Methods for adding an object to the 3D scene

These methods are to be used as arguments to the **g:Dscene3d(...)** method, as in the example above.

#### Adding facets: **g:addFacet** and **g:addPoly**

The **g:addFacet(F, options)** method, where  $\langle F \rangle$  is a facet or a list of facets (with 3D points), allows you to add these facets to the scene.

The **g:addPoly(P, options)** method allows you to add the polyhedron  $\langle P \rangle$  to the scene.

In both cases, the  $\langle options \rangle$  argument is a table whose fields define these options (with their default values) are:

- **color="white"**: Sets the fill color of the facets. This color will be shaded depending on their inclination. By default, the edges of the facets are not drawn (only the fill).
- **usepalette=nil**, this option allows you to specify a color palette for painting the facets as well as a calculation mode, the syntax is: **usepalette={palette,mode}**, where  $\langle palette \rangle$  designates a table of colors which are themselves tables of the form  $\{r, g, b\}$  where  $r$ ,  $g$  and  $b$  are numbers between 0 and 1, and  $\langle mode \rangle$  which is a string which can be either "x", or "y", or "z". In the first case for example, the facets at the center of gravity of minimum abscissa have the first color of the palette, the facets at the center of gravity of maximum abscissa have the last color of the palette, for the others, the color is calculated according to the abscissa of the center of gravity by linear interpolation.
- **opacity=1**: Number between 0 and 1 to define the opacity of the facets (1 means no transparency).
- **backcull=false**: Boolean indicating whether invisible facets should be excluded from the scene. By default, they are present.
- **clip=false**: Boolean indicating whether the facets should be clipped by the 3D window.

- **contrast=1**: Numerical value to increase or decrease the color contrast between the facets. With a value of 0, all facets have the same color.
- **twoside=true**: Boolean indicating whether the inner and outer sides of the facets are distinguished. The color of the inner side is slightly lighter than that of the outer side.
- **edge=false**: Boolean indicating whether edges should be added to the scene.
- **edgecolor=<current line color>**: Indicates the color of the edges when they are drawn.
- **edgewidth=<current line width>**: Indicates the line thickness (in tenths of a point) of the edges.
- **hidden=ld.Hiddenlines**: Boolean indicating whether hidden edges should be drawn. `ld.Hiddenlines` is a global variable that defaults to `false`.
- **hiddenstyle=ld.Hiddenlinestyle**: String defining the line style of the hidden edges. `ld.Hiddenlinestyle` is a global variable that defaults to `"dotted"`.
- **matrix=ld.ID3d**: 3D facet transformation matrix, by default this is the 3D identity matrix, i.e. the table `{M(0,0,0), vecI, vecJ, vecK}`.

### Adding a plane: `g:addPlane` and `g:addPlaneEq`

The `g:addPlane(P, options)` method adds the plane  $\langle P \rangle$  to the 3D scene. This plane is defined as a table  $\{A, u\}$  where  $A$  is a point on the plane (a 3D point) and  $u$  is a normal vector to the plane (a non-zero 3D point). This function determines the intersection between this plane and the parallelepiped given by the `window3d` argument (itself defined when the graph is created), which results in a facet, which is added to the scene. This method uses `g:addFacet()`.

The method `g:addPlaneEq(coef, options)`, where  $\langle coef \rangle$  is a table consisting of four real numbers  $\{a, b, c, d\}$ , allows you to add the plane with the equation  $ax + by + cz + d = 0$  to the scene (this method uses the previous one).

In both cases, the optional argument  $\langle options \rangle$  is a table whose fields define the options. These options are those of the `g:addFacet()` method, plus the options:

- **rectangle=nil**: With the value `nil`, the plane is intersected with the 3D window, and the resulting facet will be drawn. It is not necessarily rectangular. With **rectangle={V, L1, L2}**, the drawn facet will be rectangular with sides of lengths  $\langle L1 \rangle$  and  $\langle L2 \rangle$ , and the vector  $\langle V \rangle$  (which must belong to the plane) will determine the direction of one side of the rectangle. This vector  $\langle V \rangle$  is optional; if it is omitted, the function will choose it itself. The argument  $\langle L2 \rangle$  is also optional; if omitted, it implicitly has the same value as  $\langle L1 \rangle$ .
- **scale=1**: This number is a scaling ratio; it is only taken into account when **rectangle=nil**. In this case, the facet is scaled with a scale ratio of `scale`. This allows you to adjust the size of the plane in its representation.

**scale=1**: this number is a homothety ratio. We apply to the facet the homothety with center the centroid of the facet and ratio `scale`. This allows you to adjust the size of the plane in its representation.

### Add a polygonal line: `g:addPolyline`

The method `g:addPolyline(L, options)`, where  $\langle L \rangle$  is a list of 3D points, or a list of lists of 3D points, adds  $\langle L \rangle$  to the scene. The  $\langle options \rangle$  argument is a table whose fields define these options (with their default values) are:

- **style=<current line style>**: to set the line style; this is the current style by default.
- **color=**: line color.
- **close=false**: indicates whether the line  $\langle L \rangle$  (or each component of  $\langle L \rangle$ ) should be closed.
- **clip=false**: Indicates whether the line  $\langle L \rangle$  (or each component of  $\langle L \rangle$ ) should be clipped by the 3D window.
- **width=<current line width>**: line thickness in tenths of a point.
- **opacity=1**: opacity of the line drawing (1 means no transparency).



- `hidden=ld.Hiddenlines`: boolean indicating whether the hidden parts of the line should be represented. `ld.Hiddenlines` is a global variable that defaults to `false`.
- `hiddenstyle=ld.Hiddenlinestyle`: string defining the line style of the hidden parts. `ld.Hiddenlinestyle` is a global variable that defaults to `"dotted"`.
- `arrows=0`: this option can be 0 (no arrow added to the line), 1 (an arrow added at the end of the line), or 2 (an arrow at the beginning and end of the line). The arrows are small cones.
- `arrowscale=1`: Allows you to reduce or increase the size of the arrows.
- `matrix=ld.ID3d`: 3D transformation matrix (of the line). By default, this is the 3D identity matrix, i.e., the table `{M(0,0,0),vecI,vecJ,vecK}`.

### Add Axes: `g:addAxes`

The `g:addAxes(O, options)` method adds the axes  $(O, \text{vecI})$ ,  $(O, \text{vecJ})$ , and  $(O, \text{vecK})$  to the 3D scene, where the argument  $\langle O \rangle$  is a 3D point. The options are those of the `g:addPolyline()` method, plus the `legend=true` option, which allows you to automatically add a legend to the end of each axis; these legends are managed by the option `labels={"$x$", "$y$", "$z$"}.` The axes are not graduated.

### Add a line: `g:addLine`

The `g:addLine(d, options)` method adds the line  $\langle d \rangle$  to the scene. This line is a table of the form  $\{A, u\}$  where  $A$  is a point on the line (3D point) and  $u$  is a direction vector (non-zero 3D point). The optional argument  $\langle options \rangle$  is a table whose fields define the options, these are those of the `g:addPolyline()` method, plus the `scale=1` option: this number is a homothety ratio. The homothety is applied, with the center being the midpoint of the segment representing the line, and the ratio `scale`. This allows you to adjust the size of the segment in its representation. This segment is the line clipped by the polyhedron given by the `window3d` option (itself defined when the graph is created), which results in a segment (possibly empty).

### Adding a "right" angle: `g:addAngle`

The `g:addAngle(B, A, C [, r, options])` method allows you to add the angle  $\widehat{BAC}$  in the form of a parallelogram with side  $\langle r \rangle$  (0.25 by default). Only two sides are represented. The arguments  $\langle B \rangle$ ,  $\langle A \rangle$  and  $\langle C \rangle$  are 3D points. The options are those of the `g:addPolyline()` method.

### Add a circular arc: `g:addArc`

The `g:addArc(B, A, C, r, direction [, normal, options])` method adds the arc of a circle centered at  $\langle A \rangle$  (3D point), with radius  $\langle r \rangle$ , extending from  $\langle B \rangle$  to  $\langle C \rangle$  (3D points) in the direct direction if  $\langle direction \rangle$  is 1 (indirect otherwise). The arc is drawn in the plane passing through  $\langle A \rangle$  and orthogonal to the  $\langle normal \rangle$  vector (non-zero 3D point); this same vector orients the plane. If the vector  $\langle normal \rangle$  is not specified, then by default it will be the cross product  $\vec{AB} \wedge \vec{AC}$ . The options are those of the `g:addPolyline()` method.

### Add a circle: `g:addCircle`

The `g:addCircle(A, r, normal, options)` method adds the circle with center  $A$  (3D point) and radius  $r$  in the plane passing through  $A$  and orthogonal to the  $normal$  vector (non-zero 3D point). The options are those of the `g:addPolyline` method.

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

local g = ld.graph3d:new{window={-5,5,-7,5}, viewdir={"central",30,65,20},
  size={10,10},margin={0,0,0,0}}
ld.Hiddenlines = false
local R, r, A, B = 3, 1.5
```

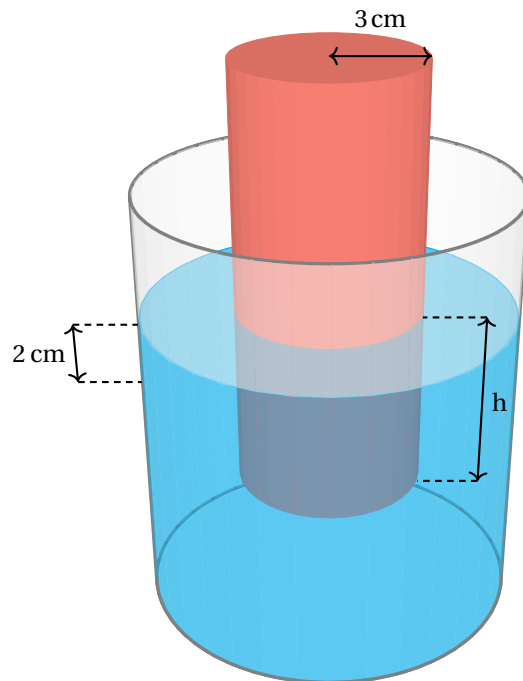


```

local C1 = ld.cylinder(M(0,0,-5),5*vecK,R) -- to model water
local C2 = ld.cylinder(Origin,2*vecK,R,35,true) -- part of the container above the water (open cylinder)
local C3 = ld.cylinder(M(0,0,-3),7*vecK,r) -- small cylinder submerged in water
-- under the 3D scene
g:Lineoptions(nil,"gray",12)
g:Dcylinder(M(0,0,-5),7*vecK,R,{hiddenstyle="noline"}) -- outline of the container (large cylinder)
-- scène 3D
g:Dscene3d(
    g:addPoly(C1,{contrast=0.125,color="cyan",opacity=0.5}), -- water
    g:addPoly(C2,{contrast=0.125,color="WhiteSmoke", opacity=0.5}), -- part of the container above the water
    g:addPoly(C3,{contrast=0.25,color="Salmon",backcull=true}), -- small cylinder in the water
    g:addCircle(M(0,0,2),R,vecK,{color="gray"}), -- upper edge of the container
    g:addCircle(M(0,0,-5),R,vecK,{color="gray"}), -- lower edge of the container
    g:addCircle(Origin,R-0.025,vecK, {width=2,color="cyan"}) -- upper edge water
)
-- over the 3D scene
g:Lineoptions(nil,"black",8); A = 4*vecK; B = A+r*g:ScreenX()
g:Dpolyline3d( {A,B}, "<->"); g:Dlabel3d("$3\\$,cm", (A+B)/2,{pos="N",dist=0.25})
A = Origin+(r+1)*g:ScreenX(); A = ld.rotate3d(A,-10,{Origin,vecK})
B = A-3*vecK
g:Dpolyline3d( {A,B}, "<->"); g:Dlabel3d("h", (A+B)/2,{pos="E"})
g:Lineoptions("dashed")
g:Dpolyline3d({{A,A-g:ScreenX()}},{B,B-g:ScreenX()})
A = Origin-(R+1)*g:ScreenX(); A = ld.rotate3d(A,10,{Origin,vecK})
B = A-vecK
g:Dpolyline3d({{A,A+g:ScreenX()}},{B,B+g:ScreenX()})
g:Linestyle("solid")
g:Dpolyline3d( {A,B}, "<->"); g:Dlabel3d("$2\\$,cm", (A+B)/2,{pos="W"})
g:Show()
\end{luadraw}

```

Figure 26: Solid cylinder immersed in water

**Notes :**

- The **g:ScreenX()** method returns the space vector (3D point) corresponding to the vector with affix 1 in the screen plane, and the **g:ScreenY()** method returns the space vector (3D point) corresponding to the vector with affix  $i$  in the screen plane.
- For the small cylinder (C3), we use the **backcull=true** option to reduce the number of facets; however, we do not do

this for the other two cylinders (C1 and C2) because they are transparent.

### Adding points: `g:addDots`

The `g:addDots(dots, options)` method allows you to add 3D points to the scene. The argument  $\langle dots \rangle$  is either a 3D point or a list of 3D points. The optional argument  $\langle options \rangle$  is a table whose fields define the options, these options are (with the default value):

- `style="ball"`: String defining the dot style. These are all 2D point styles, plus the `"ball"` (sphere) style, which is the default.
- `color="black"`: String defining the dot color.
- `scale=1`: Number allowing you to adjust the size of the points.
- `matrix=ld.ID3d`: 3D transformation matrix. By default, this is the 3D identity matrix, i.e., the table  $\{M(0,0,0), \text{vecI}, \text{vecJ}, \text{vecK}\}$ .

### Adding Labels: `g:addLabels`

The `g:addLabel(text1, anchor1, options1, text2, anchor2, options2, ...)` method allows you to add the labels  $\langle text1 \rangle$ ,  $\langle text2 \rangle$ , etc. The (required) arguments  $\langle anchor1 \rangle$ ,  $\langle anchor2 \rangle$ , etc., are 3D points representing the anchor points of the labels. The (required) arguments  $\langle options1 \rangle$ ,  $\langle options2 \rangle$ , etc., tables whose fields define the options. These options are (with the default value):

- `color=<current color>`: String defining the label color.
- `pos=<current style>`: A string defining the position of the label relative to the anchor point (as in 2D: `"N"`, `"NW"`, `"W"`, ...).
- `dist=0`: Expresses the distance between the label and its anchor point (in the screen plane).
- `size=<current size>`: String defining the label size.
- `dir={}`: Table defining the writing direction in space (usual direction by default). In general, `dir={dirX, dirY, dep}`, and the three values  $\langle dirX \rangle$ ,  $\langle dirY \rangle$ , and  $\langle dep \rangle$  are three 3D points representing three vectors: the first two indicate the writing direction, the third a displacement (translation) of the label relative to the anchor point.
- `showdot=false`: Boolean indicating whether a (2D) point should be drawn at the anchor point.
- `matrix=ld.ID3d`: 3D transformation matrix; by default, this is the 3D identity matrix, i.e., the table  $\{M(0,0,0), \text{vecI}, \text{vecJ}, \text{vecK}\}$ .

**Note:** As in 2D, the options of a label apply to subsequent labels as long as they are not modified.

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

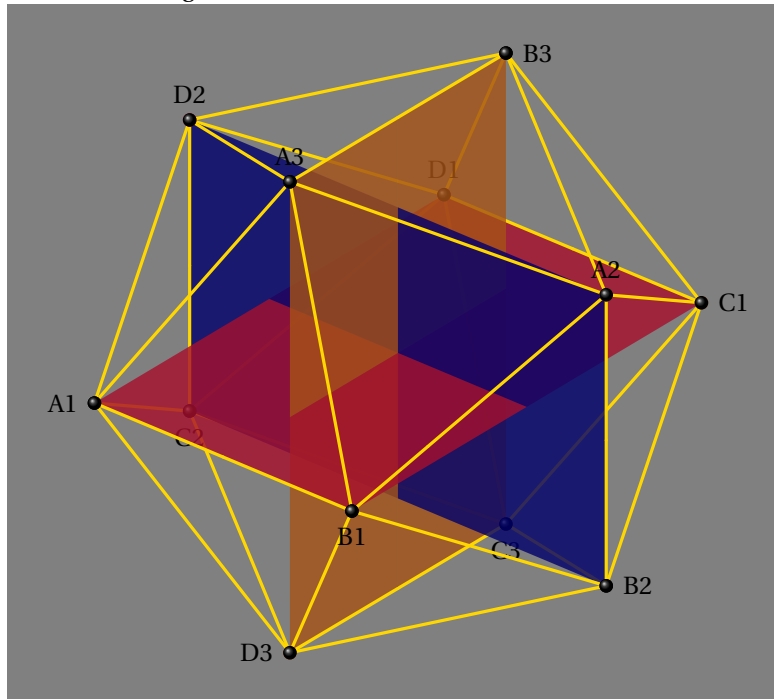
local g = ld.graph3d:new{window={-2.25,2.25,-2,2}, viewdir={40,60},bg="gray",size={10,10},margin={0,0,0,0}}
ld.Hiddenlines = false
local phi = (1+math.sqrt(5))/2 -- golden ratio
local A1, B1, C1, D1 = M(phi,-1,0), M(phi,1,0), M(-phi,1,0), M(-phi,-1,0) -- in the plane z=0
local A2, B2, C2, D2 = M(0,phi,1), M(0,phi,-1), M(0,-phi,-1), M(0,-phi,1) -- in the plane x=0
local A3, B3, C3, D3 = M(1,0,phi), M(-1,0,phi), M(-1,0,-phi), M(1,0,-phi) -- in the plane y=0
local ico = {
{A1,B1,A3}, {B1,A1,D3}, {D1,C1,C3}, {C1,D1,B3},
{B2,A2,B1}, {A2,B2,C1}, {D2,C2,A1}, {C2,D2,D1},
{B3,A3,A2}, {A3,B3,D2}, {D3,C3,B2}, {C3,D3,C2},
{A1,A3,D2}, {B1,A2,A3}, {A2,C1,B3}, {D1,D2,B3},
{B2,B1,D3}, {A1,C2,D3}, {B2,C3,C1}, {C2,D1,C3} }
g:Dscene3d(
  g:addFacet({A2,B2,C2,D2},{color="Navy",twoside=false,opacity=0.8}),
  g:addFacet({A1,B1,C1,D1},{color="Crimson",twoside=false,opacity=0.8}),
  g:addFacet({A3,B3,C3,D3},{color="Chocolate",twoside=false,opacity=0.8}),
)
```

```

g:addPolyline(ld.facetedges(ico), {color="Gold",width=12}), -- drawing edges only
g:addDots({A1,B1,C1,D1,A2,B2,C2,D2,A3,B3,C3,D3}, {color="black",scale=1.2}),
g:addLabel("A1",A1,{style="W",dist=0.1}, "B1",B1,{style="S"}, "C2",C2,{}, "C3",C3,{},
" A3",A3,{style="N"}, "D1",D1,{}, "A2",A2,{}, "D2",D2,{}, "B3",B3,{style="E"},
"C1",C1,{}, "B2",B2,{}, "D3",D3,{style="W"} )
)
g:Show()
\end{luadraw}

```

Figure 27: Construction of an icosahedron



### Adding dividing walls: g:addWall

Dividing walls are 3D objects that are inserted first into the scene tree. These objects are not drawn (therefore invisible); their role is to partition the space because a facet on one side of a dividing wall cannot be cut by the plane of a facet on the other side of the wall. This allows, in some cases, to significantly reduce the number of facet (or polygonal line) cuts during scene construction. A dividing wall can be an entire plane (i.e., a table of two 3D points of the form  $\{A, n\}$ , i.e., a point and a normal vector), or just a facet.

The syntax is: **g:addWall(C, options)** where  $\langle C \rangle$  is either a plane, a list of planes, a facet, or a list of facets. The  $\langle options \rangle$  argument is a table with only one option:

- **matrix=ld.ID3d**: 3D transformation matrix. By default, this is the 3D identity matrix, i.e., the table  $\{M(0,0,0), \text{vecI}, \text{vecJ}, \text{vecK}\}$ .

In the following example, the two dividing walls have been drawn for visualization, but they are normally invisible:

```

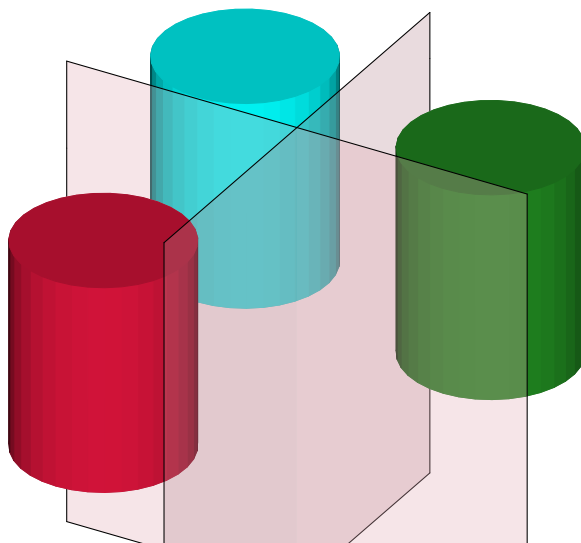
\begin{luadraw}{name=addWall}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

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

```

```
\end{luadraw}
```

Figure 28: Example with addWall (the two transparent pink facets are normally invisible)

**Notes on this example :**

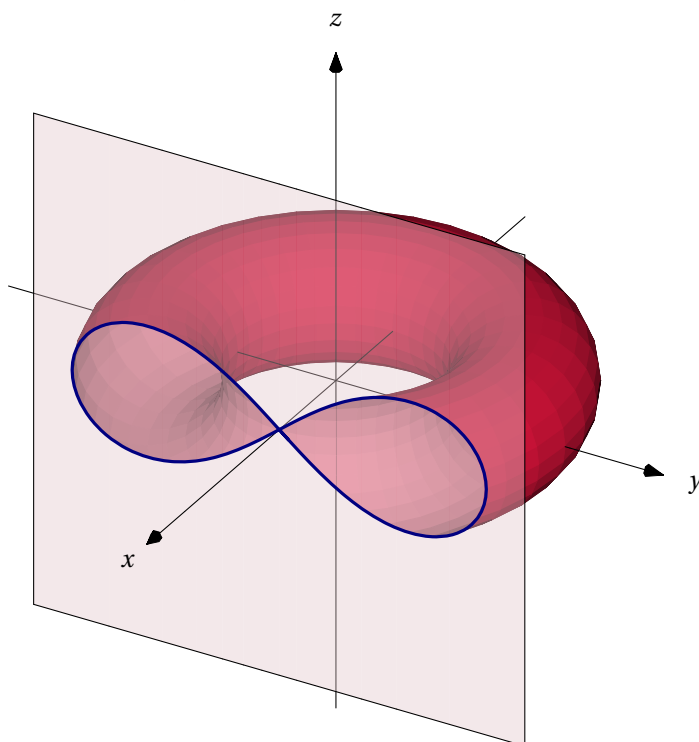
- with the two dividing walls, there are no cut facets, and the scene contains exactly 111 (37 per cylinder).
- without the dividing walls, there are 117 (useless) facet cuts, bringing their number to 228 in the scene.
- with the two dividing walls, and the `backcull=true` option for each cylinder, there are no cut facets, and the scene contains only 57.

Here is another, much more convincing example where the use of dividing walls is essential to have a drawing of reasonable size. It involves obtaining a lemniscate as the intersection of a torus with a certain plane. Since the torus is non-convex, the number of unnecessary facet cuts can be very high.

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

local g = ld.graph3d:new{size={10,10}, margin={0,0,0,0}}
local cos, sin, pi = math.cos, math.sin, math.pi
local R, r = 2.5, 1
local x0 = R-r
local f = function(t) return M(0,R+r*cos(t),r*sin(t)) end
local plan = {M(x0,0,0),-vecI} -- plane whose section with the torus gives the lemniscate
local C, wall = ld.rotcurve(f,-pi,pi,{Origin,vecK},360,0,{grid={25,37},addwall=2})
local C1 = ld.cutfacet(C,plan) -- part of the torus in the half-space containing -vecI
g:Dscene3d(
  g:addWall(plan), g:addWall(wall), -- addition of partition walls
  g:addFacet( C1, {color="Crimson", backcull=false}),
  g:addPlane(plan, {color="Pink",opacity=0.4,edge=true}), -- sectional plane
  g:addAxes( Origin, {arrows=1})
)
-- Cartesian equation of the torus : (x^2+y^2+z^2+R^2-r^2)^2-4*R^2*(x^2+y^2) = 0
-- the lemniscate therefore has the equation (x0^2+y^2+z^2+R^2-r^2)^2-4*R^2*(x0^2+y^2)=0 (implicit curve)
local h = function(y,z) return (x0^2+y^2+z^2+R^2-r^2)^2-4*R^2*(x0^2+y^2) end
local I = ld.implicit(h,-4,4,-3,3,{50,50}) -- 2D polygonal line (list of lists of complex numbers)
local lemniscate = ld.map(function(z) return M(x0,z.re,z.im) end, I[1]) -- conversion to 3D coordinates
g:Dpolyline3d(lemniscate,"Navy,line width=1.2pt")
g:Show()
\end{luadraw}
```

Figure 29: Torus and lemniscate

**Notes on this example :**

- With the dividing walls, we have 30 facets that are cut and a tkz file of approximately 140 KB.
- Without the dividing walls, we have 2068 facet cuts (!) and a tkz file of approximately 550 KB.
- We could have used the cut section returned by the **ld.cutfacet()** function, but the result is not very satisfactory (this is because the torus is non-convex).
- If we hadn't wanted the axes passing through the torus and the cutting plane, we could have drawn the drawing with the **g:Dfacet()** method, replacing the **g:Dscene3d()** instruction with:

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

We get exactly the same thing but without the axes (and without facet cutting, of course).

**To conclude this section :** we use the **g:Dscene3d()** method when it is not possible to do otherwise, for example when there are intersections (few) that cannot be handled "by hand". But this isn't the case for all intersections! In the following example, we represent a section of a sphere using a plane, but without using the **g:Dscene3d()** method, as this would require drawing a faceted sphere, which isn't very attractive. The trick here is to draw the sphere using the **g:Dsphere()** method, then draw a previously perforated facet over the plane, the hole corresponding to the outline (3D path) of the part of the sphere located above the plane:

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

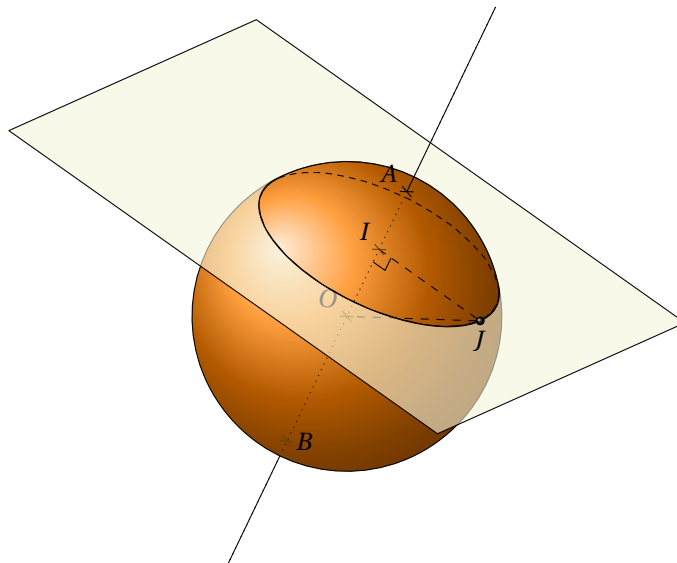
local g = ld.graph3d:new{ window3d={-4,4,-4,4,-4,4}, window={-5.5,5.5,-4,5}, viewdir={30,75}, size={10,10}}
local O, R = Origin, 2.5 -- center and radius
local S, P = ld.sphere(O,R), {M(0,0,1.5),vecK+vecJ/2} -- the sphere and the section plane
local w, n = pt3d.normalize(P[2]), g.Normal -- unit vectors normal to P for w and to the screen for n
local I, r = ld.interPS(P,{O,R}) -- center and radius of the small circle (intersection between the plane and the
-- sphere)
local C = g:Intersection3d(S,P) -- It is a list of edges
```

```

local N = I-O
local J = I+r*pt3d.normalize(vecJ-vecK/2) -- a point on the small circle
local a = R/pt3d.abs(N)
local A, B = O+a*N, O-a*N -- points of intersection of the axis (O,I) with the sphere
local c1, alpha = ld.Orange, 0.4
local coul = {c1[1]*alpha, c1[2]*alpha, c1[3]*alpha} --to simulate transparency
g:Dhline( g:Proj3d({B,-N})) -- half-line (point B is not visible)
g:Dsphere(O,R,{mode=ld.mBorder,color="orange"})
g:Dline3d(A,B,"dotted") -- dotted line (A,B)
g:Dedges(C, {hidden=true,hiddenstyle="dashed"}) -- drawing of the intersection
g:Dpolyline3d({I,J,O},"dashed")
g:Dangle3d(O,I,J) -- right angle
g:Dcrossdots3d({{B,N},{I,N},{O,N}},ld.rgb(coul),0.75) -- points in the sphere
g:Dlabel3d("$O$", O, {pos="NW"})
local L = C.visible[1] -- visible part of the intersection (arc of a circle)
A1 = L[1]; A2 = L[#L] -- ends of L
local F = g:Plane2facet(P) -- plan converted to facet
-- hole plane as 3D path, the hole is the outline of the part of the sphere above the plane
ld.insert(F,{"l","c1",A1,"m",I,A2,r,-1,w,"ca",Origin,A1,R,-1,n,"ca"})
g:Dpath3d( F,"fill=Beige,fill opacity=0.6") -- drawing of the perforated plan
g:Dhline( g:Proj3d({A,N})) -- half-line, upper part of the axis (AB)
g:Dcrossdots3d({A,N},"black",0.75); g:Dballdots3d(J,"black",0.75)
g:Dlabel3d("$A$", A, {pos="NW"}, "$I$", I, {}, "$B$", B, {pos="E"}, "$J$", J, {pos="S"})
g:Show()
\end{luadraw}

```

Figure 30: Section of a sphere without Dscene3d()



## VI Geometric Constructions

This section groups together functions that construct geometric figures without dedicated graphics methods.

### 1) Circumscribed circle, incircle: `circumcircle3d()`, `incircle3d()`

- The function `ld.circumcircle3d(A, B, C)`, where  $\langle A \rangle$ ,  $\langle B \rangle$  and  $\langle C \rangle$  are three non-aligned 3D points, returns the circumcircle of the triangle formed by these three points, in the form of a sequence:  $O, R, n$ , where  $O$  is the center of the circle,  $R$  its radius, and  $n$  a normal vector to the plane of the circle.
- The function `ld.incircle3d(A, B, C)`, where  $\langle A \rangle$ ,  $\langle B \rangle$  and  $\langle C \rangle$  are three non-aligned 3D points, returns the circle inscribed in the triangle formed by these three points, as a sequence:  $I, R, n$ , where  $I$  is the center of the circle,  $R$  its radius, and  $n$  a normal vector to the plane of the circle.

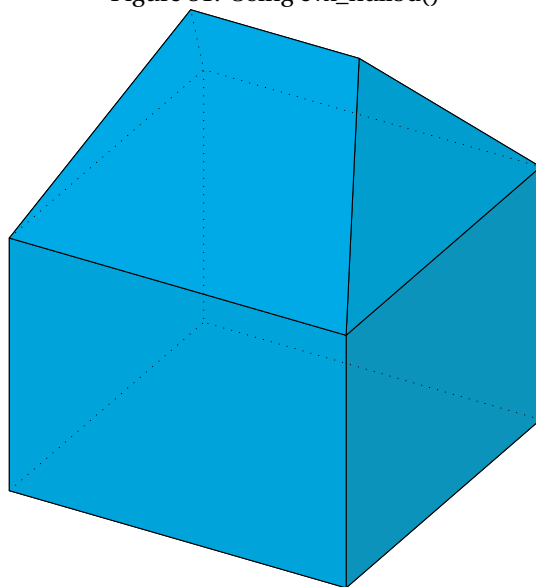
## 2) Convex Hull: `cvx_hull3d()`

The function `ld.cvx_hull3d(L)`, where  $\langle L \rangle$  is a list of **distinct** 3D points, calculates and returns the convex hull of  $\langle L \rangle$  as a list of facets.

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

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

Figure 31: Using `cvx_hull3d()`



**Special case** : when all points of  $\langle L \rangle$  are in the same plane, we can use the function `ld.cvx_hull3dcoplanar(L, n)` where  $\langle n \rangle$  is a vector orthogonal to the plane. This function returns a facet (list of 3D points).

## 3) Planes: `plane()`, `planeEq()`, `orthoframe()`, `plane2ABC()`

A plane in space is a table of the form  $\{A, n\}$  where  $A$  is a point in the plane (3d point) and  $n$  is a normal vector to the plane (non-zero 3D point).

- The function `ld.plane(A, B, C)` returns the plane passing through the three 3D points  $\langle A \rangle$ ,  $\langle B \rangle$  and  $\langle C \rangle$  (if they are not aligned, otherwise the result is `nil`).
- The function `ld.planeEq(a, b, c, d)` returns the plane whose Cartesian equation is  $ax + by + cz + d = 0$  (if the coefficients  $\langle a \rangle$ ,  $\langle b \rangle$  and  $\langle c \rangle$  are not all zero, otherwise the result is `nil`).
- The function `ld.plane2ABC(P)`, where  $\langle P \rangle = \{A, n\}$  denotes a plane, returns a sequence of three 3D points  $A, B, C$ , belonging to the plane, and such that  $(A, \vec{AB}, \vec{AC})$  is a direct orthonormal frame of this plane.
- The function `ld.orthoframe(P)`, where  $\langle P \rangle = \{A, n\}$  denotes a plane, returns a sequence of three 3D points  $A, u, v$ , such that  $(A, u, v)$  is a direct orthonormal frame of this plane.

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

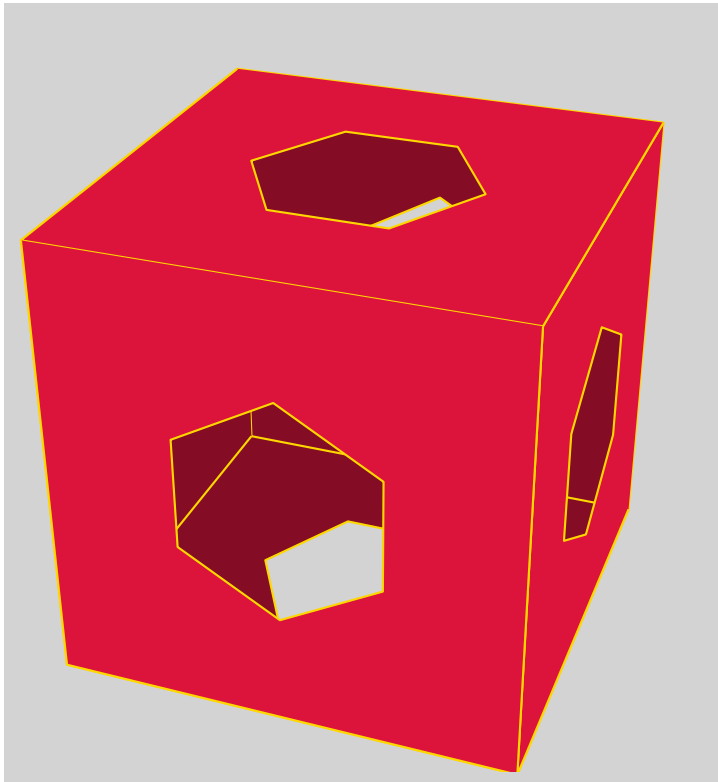
```

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

local g = ld.graph3d:new{window={-3,3,-3.25,3.25}, margin={0,0,0,0},
    viewdir={"central",20,60}, bg="LightGray", size={10,10}}
ld.Hiddenlines = true; ld.Hiddenlinestyle = "dashed"
local p = ld.polyreg(0,1,6)
local P = ld.parallelelep(M(-2,-2,-2),4*vecI,4*vecJ,4*vecK)
local V = g:Sortpolyfacet(P)
local list = {}
g:Filloptions("full","Crimson",1,true); -- true for even odd rule
g:Lineoptions("solid","Gold",8)
for _, F in ipairs(V) do
    local P1 = ld.plane(pt3d.isobar3d(F),F[1],F[2]) -- plane of facet F
    local A, u, v = ld.orthoframe(P1) -- orthonormal coordinate system on the facet with center of gravity as the
    -- origin
    local p1 = ld.map(function(z) return A+z.re*u+z.im*v end,p) -- hexagon reproduced on the facet
    table.insert(p1,2,"m")
    local color = "Crimson"
    if not g:Isvisible(F) then color = "Crimson!60!black" end
    g:Dpath3d( ld.concat(F,{"1"},p1,{"1","c1"}),"fill"..color ) -- drawing of the "perforated" facet with the hexagon
end
g:Show()
\end{luadraw}

```

Figure 32: Faces of a cube with holes in it and a regular hexagon



#### 4) Circumscribed Sphere, Inscribed Sphere: `circumsphere()`, `insphere()`

- The function `ld.circumsphere(A, B, C, D)`, where  $\langle A \rangle$ ,  $\langle B \rangle$ ,  $\langle C \rangle$  and  $\langle D \rangle$  are four non-coplanar 3D points, returns the sphere circumscribed within the tetrahedron formed by these four points, as a sequence:  $I, R$ , where  $I$  is the center of the sphere, and  $R$  its radius.
- The function `ld.insphere(A, B, C, D)`, where  $\langle A \rangle$ ,  $\langle B \rangle$ ,  $\langle C \rangle$  and  $\langle D \rangle$  are four non-coplanar 3D points, returns the sphere inscribed within the tetrahedron formed by these four points, as a sequence:  $I, R$ , where  $I$  is the center of the sphere, and  $R$  its radius.

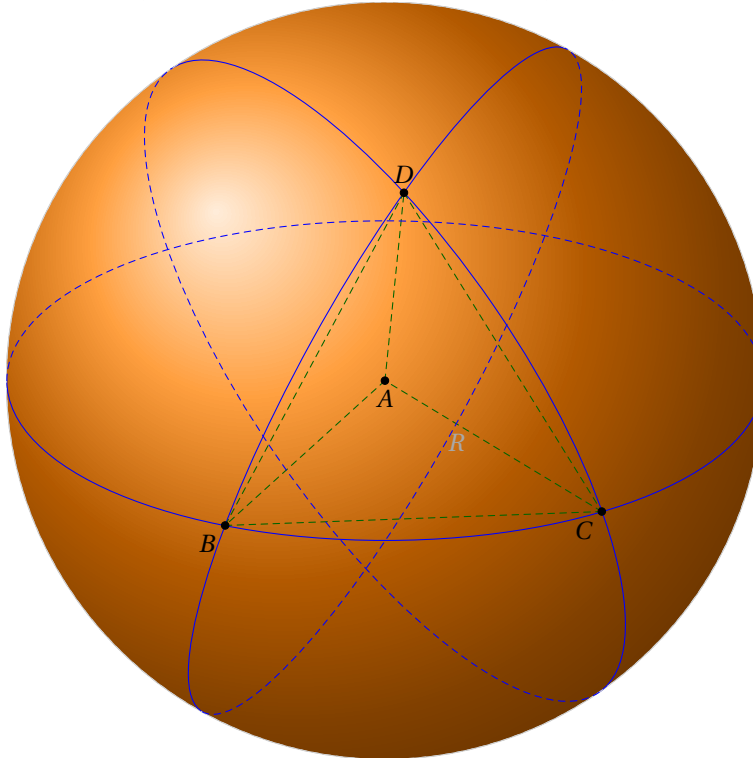


### 5) Fixed-length tetrahedron: `tetra_len()`

The function `ld.tetra_len(ab, ac, ad, bc, bd, cd)` calculates the vertices  $A, B, C, D$  of a tetrahedron whose edge lengths are given, i.e., such that  $AB = \langle ab \rangle$ ,  $AC = \langle ac \rangle$ ,  $AD = \langle ad \rangle$ ,  $BC = \langle bc \rangle$ ,  $BD = \langle bd \rangle$  and  $CD = \langle cd \rangle$ . The function returns the sequence of four points  $A, B, C, D$ . Vertex  $A$  is always the point  $M(0, 0, 0)$  (`pt3d.Origin`) and vertex  $B$  is always the point `ab*pt3d.vecI` and vertex  $C$  in the  $xy$  plane. The tetrahedron as a polyhedron can then be constructed with the function `tetra(A, B-A, C-A, D-A)`.

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

Figure 33: A tetrahedron with fixed edge lengths



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

These functions are the 3D version of the `sss_triangle()`, `sas_triangle()`, and `asa_triangle()` functions already described.

- The function `ld.sss_triangle3d(ab, bc, ca)`, where  $\langle ab \rangle$ ,  $\langle bc \rangle$ , and  $\langle ca \rangle$  are three lengths, computes and returns a list of three 3D points  $\{A, B, C\}$  forming the vertices of a direct triangle in the  $xOy$  plane, whose side lengths are the arguments, i.e.,  $AB = \langle ab \rangle$ ,  $BC = \langle bc \rangle$  and  $CA = \langle ca \rangle$ , when possible. Vertex  $A$  is always point  $M(0, 0, 0)$  (`pt3d.Origin`) and vertex  $B$  is always point `ab*pt3d.vecI`. This triangle can be drawn with the method `g:Dpolyline3d()`.

- The function **ld.sas\_triangle3d(ab, alpha, ca)** where  $\langle ab \rangle$  and  $\langle ca \rangle$  are two lengths,  $\langle alpha \rangle$  an angle in degrees, computes and returns a list of three 3D points  $\{A, B, C\}$  forming the vertices of a triangle in the plane  $xOy$  such that  $AB = \langle ab \rangle$ ,  $CA = \langle ca \rangle$ , and such that the angle  $(\vec{AB}, \vec{AC})$  has measure  $\langle alpha \rangle$ , when possible. Vertex  $A$  is always point  $M(0, 0, 0)$  (`pt3d.Origin`) and vertex  $B$  is always point  $\mathbf{ab} * \mathbf{pt3d.vecI}$ . This triangle can be drawn with the method **g:Dpolyline3d()**.
- The function **ld.asa\_triangle3d(alpha, ab, beta)** where  $\langle ab \rangle$  is a length,  $\langle alpha \rangle$  and  $\langle beta \rangle$  are two angles in degrees, computes and returns a list of three 3D points  $\{A, B, C\}$  forming the vertices of a triangle in the  $xOy$  plane such that  $AB = \langle ab \rangle$ , such that angle  $(\vec{AB}, \vec{AC})$  has measure  $\langle alpha \rangle$ , and such that angle  $(\vec{BA}, \vec{BC})$  has measure  $\langle beta \rangle$ , when possible. Vertex  $A$  is always point  $M(0, 0, 0)$  (`pt3d.Origin`) and vertex  $B$  is always point  $\mathbf{ab} * \mathbf{pt3d.vecI}$ . This triangle can be drawn with the method **g:Dpolyline3d()**.

## VII Matrix Calculus Transformations and Some Mathematical Functions

### 1) 3D Transformations

In the following functions:

- the argument  $\langle L \rangle$  is either a 3D point, a polyhedron, a list of 3D points (facet), or a list of lists of 3D points (facet list),
- a line  $\langle d \rangle$  is a list of two 3D points  $\{A, u\}$ : a point on the line ( $A$ ) and a direction vector ( $u$ ),
- a plane  $\langle P \rangle$  is a list of two 3D points  $\{A, n\}$ : a point on the plane ( $A$ ) and a normal vector to the plane ( $n$ ).

The returned result is of the same type as  $\langle L \rangle$ .

#### Apply a transformation function: **ftransform3d**

The function **ld.ftransform3d(L, f)** returns the image of  $\langle L \rangle$  by the function  $\langle f \rangle$ ; this must be a function from  $\mathbf{R}^3$  to  $\mathbf{R}^3$  (to a 3D point it associates a 3D point).

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

- The function **ld.proj3d(L, P)** returns the image of  $\langle L \rangle$  by the orthogonal projection onto the plane  $\langle P \rangle$ .
- The function **ld.proj3dO(L, P, v)** returns the image of  $\langle L \rangle$  by the projection onto the plane  $\langle P \rangle$  parallel to the direction of the vector  $\langle v \rangle$  (non-zero 3D point).
- The function **ld.dproj3d(L, d)** returns the image of  $\langle L \rangle$  by the projection onto the line  $\langle d \rangle$ .

#### Projections onto axes or planes related to axes

- The function **ld.pxy(L [, z0])** returns the image of  $\langle L \rangle$  by the orthogonal projection onto the  $z = \langle z0 \rangle$  plane (by default  $z0 = 0$ ).
- The function **ld.pyz(L [, x0])** returns the image of  $\langle L \rangle$  by the orthogonal projection onto the  $x = \langle x0 \rangle$  plane (by default  $x0 = 0$ ).
- The function **ld.pxz(L [, y0])** returns the image of  $\langle L \rangle$  by the orthogonal projection onto the  $y = \langle y0 \rangle$  plane (by default  $y0 = 0$ ).
- The function **ld.px(L)** returns the image of  $\langle L \rangle$  by the orthogonal projection onto the  $x$ -axis.
- The function **ld.py(L)** returns the image of  $\langle L \rangle$  by the orthogonal projection onto the  $y$ -axis.
- The function **ld.pz(L)** returns the image of  $\langle L \rangle$  by the orthogonal projection onto the  $z$ -axis.

**Symmetries: sym3d, sym3dO, dsym3d, psym3d**

- The function **ld.sym3d(L, P)** returns the image of  $\langle L \rangle$  by the orthogonal symmetry about the  $\langle P \rangle$  plane.
- The function **ld.sym3dO(L, P, v)** returns the image of  $\langle L \rangle$  by the symmetry about the  $\langle P \rangle$  plane and parallel to the direction of the  $\langle v \rangle$  vector (non-zero 3D point).
- The function **ld.dsym3d(L, d)** returns the image of  $\langle L \rangle$  by the orthogonal symmetry with respect to the line  $\langle d \rangle$ .
- The function **ld.psym3d(L, point)** returns the image of  $\langle L \rangle$  by the symmetry with respect to  $\langle point \rangle$  (3d point).

**Rotation: rotate3d, rotateaxe3d**

- The function **ld.rotate3d(L, angle, d)** returns the image of  $\langle L \rangle$  rotated along axis  $\langle d \rangle$  (oriented by the direction vector, which is  $d[2]$ ), and by  $angle$  degrees.
- The function **ld.rotateaxe3d(L, v1, v2 [, center])** returns the image of  $\langle L \rangle$  rotated along axis passing through the 3D point  $\langle center \rangle$ , which transforms the vector  $\langle v1 \rangle$  into the vector  $\langle v2 \rangle$ . These vectors are normalized by the function. The argument  $\langle center \rangle$  is optional and defaults to the point `pt3d.Origin`.

**Scaling: scale3d**

The function **ld.scale3d(L, k [, center])** returns the image of  $\langle L \rangle$  by the scaling with center at the 3D point  $\langle center \rangle$ , and ratio  $\langle k \rangle$ . The argument  $\langle center \rangle$  is optional and defaults to the point `pt3d.Origin`.

**Inversion: inv3d**

The function **ld.inv3d(L, radius [, center])** returns the image of  $\langle L \rangle$  by the inversion with respect to the sphere with center  $\langle center \rangle$ , and radius  $\langle radius \rangle$ . The argument  $\langle center \rangle$  is optional and defaults to the point `pt3d.Origin`.

**Stereography: projstereo and inv\_projstereo**

Function **ld.projstereo(L, S, N, h)**: the argument  $\langle L \rangle$  denotes a 3D point or a list of 3D points or a list of lists of 3D points, all belonging to the sphere  $\langle S \rangle$ , where  $\langle S \rangle = \{C, r\}$  ( $C$  is the center of the sphere, and  $r$  the radius). The argument  $\langle N \rangle$  denotes a point on the sphere that will be the pole of the projection. The argument  $\langle h \rangle$  is a real number that defines the projection plane. This plane is perpendicular to the axis  $(CN)$ , and passes through the point  $I = C + h \frac{\vec{CN}}{CN}$  (with  $h = 0$  it is the equatorial plane, with  $h = -r$  it is the plane tangent to the sphere at the opposite pole). The function returns the image of  $\langle L \rangle$  by the stereographic projection with respect to the sphere  $\langle S \rangle$  with  $\langle N \rangle$  as pole, and on the plane  $\{I, N - C\}$ .

Inverse function **ld.inv\_projstereo(L, S, N)**:  $\langle S \rangle = \{C, r\}$  is the sphere with center  $C$  and radius  $r$ ,  $\langle N \rangle$  is a point on the sphere  $\langle S \rangle$  (pole), and  $\langle L \rangle$  is a 3D point or a list of 3D points or a list of lists of 3D points all belonging to the same plane orthogonal to the  $(CN)$  axis. The function returns the image of  $\langle L \rangle$  by the inverse of the stereographic projection with respect to  $\langle S \rangle$  and with pole  $\langle N \rangle$ .

**Translation: shift3d**

The function **ld.shift3d(L, v)** returns the image of  $\langle L \rangle$  by the translation of vector  $\langle v \rangle$  (3D point).

**2) Matrix Calculus**

If  $f$  is an affine mapping of the space  $\mathbf{R}^3$ , we will call the list (table) of  $f$  a matrix:

$$\{f(\text{pt3d.Origin}), Lf(\text{pt3d.vecI}), Lf(\text{pt3d.vecJ}), Lf(\text{pt3d.vecK})\}$$

where  $Lf$  denotes the linear part of  $f$  (we have  $Lf(\text{pt3d.vecI}) = f(\text{pt3d.vecI}) - f(\text{pt3d.Origin})$ , etc.). The identity matrix is denoted `ld.ID3d` in *luadraw*; it simply corresponds to the list:

$$\{\text{pt3d.Origin}, \text{pt3d.vecI}, \text{pt3d.vecJ}, \text{pt3d.vecK}\}$$

**applymatrix3d and applyLmatrix3d**

- The function **ld.applymatrix3d(A, M)** applies the matrix  $\langle M \rangle$  to the 3D point  $\langle A \rangle$  and returns the result (which is equivalent to calculating  $f(A)$  if  $M$  is the matrix of  $f$ ). If  $\langle A \rangle$  is not a 3D point, the function returns  $\langle A \rangle$ .
- The function **ld.applyLmatrix3d(A, M)** applies the linear part of the matrix  $\langle M \rangle$  to the 3D point  $\langle A \rangle$  and returns the result (which is equivalent to calculating  $Lf(A)$  if  $M$  is the matrix of  $f$ ). If  $\langle A \rangle$  is not a 3D point, the function returns  $\langle A \rangle$ .

**composematrix3d**

The function **ld.composematrix3d(M1, M2)** performs the matrix product  $\langle M1 \rangle \times \langle M2 \rangle$  and returns the result.

**invmatrix3d**

The function **ld.invmatrix3d(M)** calculates and returns the inverse of the matrix  $\langle M \rangle$  when possible.

**matrix3dof**

The function **ld.matrix3dof(f)** calculates and returns the matrix of  $\langle f \rangle$  which must be an affine mapping of the space  $\mathbf{R}^3$  (to a 3D point it associates a 3D point).

**mtransform3d and mLtransform3d**

- The function **ld.mtransform3d(L, M)** applies the matrix  $\langle M \rangle$  to the list  $\langle L \rangle$  and returns the result.  $\langle L \rangle$  must be a list of 3D points (a facet) or a list of lists of 3D points (a list of facets).
- The function **ld.mLtransform3d(L, M)** applies the linear part of the matrix  $\langle M \rangle$  to the list  $\langle L \rangle$  and returns the result.  $\langle L \rangle$  must be a list of 3D points (a facet) or a list of lists of 3D points (a list of facets).

**3) Matrix associated with the 3D graph**

When creating a graph in the *luadraw* environment, for example:

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

The created *g* object has a 3D transformation matrix that is initially the identity. All graphics methods automatically apply the graph's 3D transformation matrix. To manipulate this matrix, the following methods are available.

**g:Composematrix3d()**

The **g:Composematrix3d(M)** method multiplies the 3D matrix of the graph *g* by the matrix  $\langle M \rangle$  (with  $\langle M \rangle$  on the right), and the result is assigned to the graph's 3D matrix. The argument  $\langle M \rangle$  must therefore be a 3D matrix.

**g:Det3d()**

The **g:Det3d()** method returns 1 when the 3D transformation matrix has a positive determinant, and  $-1$  otherwise. This information is useful when we need to know whether the orientation of space has been changed or not.

**g:IDmatrix3d()**

The **g:IDmatrix3d()** method reassigns the identity to the 3D matrix of the graph *g*.

**g:Mtransform3d()**

The **g:Mtransform3d(L)** method applies the 3D graph matrix of *g* to *L* and returns the result. The argument *L* must be a list of 3D points (a facet) or a list of lists of 3D points (a list of facets).

**g:MLtransform3d()**

The **g:MLtransform3d(L)** method applies the linear part of the 3D matrix of the graph *g* to  $\langle L \rangle$  and returns the result. The argument  $\langle L \rangle$  must be a list of 3D points (a facet) or a list of lists of 3D points (a list of facets).

**g:Rotate3d()**

The method **g:Rotate3d(angle, axis)** modifies the 3D transformation matrix of the graph *g* by composing it with the rotation matrix of angle  $\langle angle \rangle$  (in degrees) and axis  $\langle axis \rangle$ .

**g:Scale3d()**

The method **g:Scale3d(factor [, center])** modifies the 3D transformation matrix of the graph *g* by composing it with the homothety matrix of ratio  $\langle factor \rangle$  and center  $\langle center \rangle$ . The argument  $\langle center \rangle$  is a 3D point that defaults to `pt3d.Origin`.

**g:Setmatrix3d()**

The **g:Setmatrix3d(M)** method assigns the matrix  $\langle M \rangle$  to the 3D transformation matrix of the graph *g*.

**g:Shift3d()**

The **g:Shift3d(v)** method modifies the 3D transformation matrix of the graph *g* by composing it with the translation matrix of vector  $\langle v \rangle$ , which must be a 3D point.

**4) Additional Mathematical Functions****clippolyline3d()**

The function **ld.clippolyline3d(L, poly [, exterior, close])** clips the 3D polygonal line  $\langle L \rangle$  to the **convex** polyhedron  $\langle poly \rangle$ . If the optional argument  $\langle exterior \rangle$  is `true`, then the part outside the polyhedron is returned (`false` by default). If the optional argument  $\langle close \rangle$  is `true`, then the polygonal line is closed (`false` by default).  $\langle L \rangle$  is a list of 3D points or a list of lists of 3D points.

**Note:** The result is not always satisfactory for the exterior part.

**Special case** : Clipping a 3D polygonal line  $\langle L \rangle$  with the current 3D window can be done with this function as follows:

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

Indeed, the **g:Box3d()** method returns the current 3D window as a parallelepiped.

**clipline3d()**

The function **ld.clipline3d(line, poly)** clips the line  $\langle line \rangle$  with the **convex** polyhedron  $\langle poly \rangle$ ; the function returns the part of the line inside the polyhedron. The argument  $\langle line \rangle$  is a table of the form  $\{A, u\}$  where *A* is a point on the line and *u* is a direction vector (two 3D points).

**Special case** : Clipping a line  $\langle d \rangle$  with the current 3D window can be done with this function as follows:

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

Indeed, the **g:Box3d()** method returns the current 3D window as a parallelepiped ( $\langle d \rangle$  then becomes a segment).

**cutpolyline3d()**

The function **ld.cutpolyline3d(L, plane [, close])** intersects the 3D polygonal line  $\langle L \rangle$  with the plane  $\langle plane \rangle$ . If the optional argument  $\langle close \rangle$  is `true`, then the line is closed (`false` by default).  $\langle L \rangle$  is a list of 3D points or a list of lists of 3D points,  $\langle plane \rangle$  is a table of the form  $\{A, n\}$  where *A* is a point in the plane and *n* is a normal vector (two 3D points).

The function returns three things:

- the part of  $\langle L \rangle$  that is in the half-space containing the vector *n*,

- followed by the part of  $\langle L \rangle$  that is in the other half-space,
- followed by the list of intersection points.

### **getbounds3d()**

The function **ld.getbounds3d(L)** returns the bounds  $xmin, xmax, ymin, ymax, zmin, zmax$  (sequence) of the 3D polygonal line  $\langle L \rangle$  (list of 3D points or a list of lists of 3D points).

### **interDP()**

The function **ld.interDP(d, P)** calculates and returns (if it exists) the intersection between line  $\langle d \rangle$  and plane  $\langle P \rangle$ .

### **interPP()**

The function **ld.interPP(P1, P2)** calculates and returns (if it exists) the intersection between planes  $\langle P1 \rangle$  and  $\langle P2 \rangle$ .

### **interDD()**

The function **ld.interDD(D1, D2 [, epsilon])** calculates and returns (if it exists) the intersection between lines  $\langle D1 \rangle$  and  $\langle D2 \rangle$ . The argument  $\langle epsilon \rangle$  is  $10^{-10}$  by default (used to test whether a given float is zero).

### **interCS()**

The function **ld.interCS(C, S)** calculates and returns (if it exists) the intersection between the circle  $\langle C \rangle = \{A, r, n\}$  ( $A$  is the center of the circle,  $r$  the radius, and  $n$  a normal vector to the plane of the circle), and the sphere  $\langle S \rangle = \{B, R\}$  ( $B$  is the center of the sphere and  $R$  the radius). The function returns either **nil** (empty intersection), a single point, or two points (sequence).

### **interDS()**

The function **ld.interDS(d, S)** calculates and returns (if it exists) the intersection between the line  $\langle d \rangle$  and the sphere  $\langle S \rangle$  which is a table of the form  $\{C, r\}$  with  $C$  as the center (3D point) and  $r$  as the radius of the sphere. The function returns either **nil** (empty intersection), a single point, or two points.

### **interPS()**

The function **ld.interPS(P, S)** calculates and returns (if it exists) the intersection between the plane  $\langle P \rangle$  and the sphere  $\langle S \rangle = \{C, r\}$  with  $C$  as the center (3d point) and  $r$  as the radius of the sphere. The function returns either **nil** (empty intersection) or a sequence of the form  $I, r, n$ , where  $I$  is a 3D point representing the center of a circle,  $r$  its radius, and  $n$  a normal vector to the plane of the circle. This circle is the desired intersection.

### **interSS()**

The function **ld.interSS(S1, S2)** calculates and returns (if it exists) the intersection between the sphere  $\langle S1 \rangle = \{C1, r1\}$  and  $\langle S2 \rangle = \{C2, r2\}$ . The function returns either **nil** (empty intersection) or a sequence of the form  $I, r, n$ , where  $I$  is a 3D point representing the center of a circle,  $r$  its radius, and  $n$  a normal vector to the plane of the circle. This circle is the desired intersection.

### **interSSS()**

The function **ld.interSSS(S1, S2, S3)** calculates and returns (if it exists) the intersection between the spheres  $\langle S1 \rangle = \{C1, r1\}$ ,  $\langle S2 \rangle = \{C2, r2\}$  and  $\langle S3 \rangle = \{C3, r3\}$ . The function returns either **nil** (empty intersection), a single point, or two points (sequence).

**merge3d()**

The function **ld.merge3d(L [, epsilon])** combines, if possible, the connected components of  $\langle L \rangle$ , which must be a list of lists of 3D points. The function returns the result. The argument  $\langle epsilon \rangle$  defaults to  $10^{-10}$ , it is used during comparisons.

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

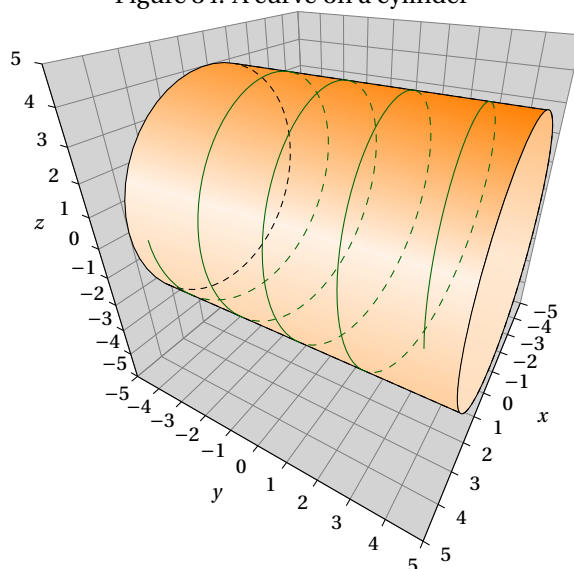
The function **ld.split\_points\_by\_visibility(L, visible\_function)**, where  $\langle L \rangle$  is a list of 3D points, or a list of lists of 3D points, and where  $\langle visible\_function \rangle$  is a function such that **visible\_function(A)** returns **true** if the 3D point  $A$  is visible, **false** otherwise, sorts the points of  $\langle L \rangle$  according to whether they are visible or not. The function returns a sequence of two tables: *visible\_points*, *hidden\_points*.

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

local g = ld.graph3d:new{adjust2d=true, bbox=false, size={10,10}, viewdir="central"}
g:Labelsize("footnotesize")
ld.Hiddenlines = true; ld.Hiddenlinestyle = "dashed"
local curve_on_cylinder = function(curve, cylinder)
-- curve is a 3D polyline on a cylinder,
-- cylinder = {A,r,V,B}
local A,r,V,B = table.unpack(cylinder)
if B == nil then B = V; V = B-A end
local U = B-A
local visible_function
if ld.projection_mode == "central" then
visible_function = function(N)
local I = ld.dproj3d(N,{A,U})
local M1, M2 = ld.interCS({I,r,U},{ (I+ld.camera)/2, pt3d.abs(I-ld.camera)/2})
local alpha = pt3d.angle3d(M1-I,ld.camera-I)
return pt3d.angle3d(N-I,ld.camera-I) <= alpha
end
else
visible_function = function(N)
local I = ld.dproj3d(N,{A,U})
return (pt3d.dot(N-I,g.Normal) >= 0)
end
end
return ld.split_points_by_visibility(curve,visible_function)
end
-- test
local A, r, B = -5*vecJ, 4, 5*vecJ -- cylinder
local p = function(t) return Mc(r,t,t/3) end
local Curve = ld.rotate3d( ld.parametric3d(p,-4*math.pi,4*math.pi),90,{Origin,vecI})
local Vi, Hi = curve_on_cylinder(Curve,{A,r,B})
local curve_color = "DarkGreen"
g:Dboxaxes3d({grid=true,gridcolor="gray",fillcolor="LightGray"})
g:Dcylinder(A,r,B,{color="orange"})
g:Dpolyline3d(Vi,curve_color)
g:Dpolyline3d(Hi,curve_color.."", "..ld.Hiddenlinestyle)
g:Show()
\end{luadraw}
```



Figure 34: A curve on a cylinder



## VIII More Advanced Examples

### 1) The Box of Sugars

The problem<sup>7</sup> is to draw sugars in a box. You need to be able to position the desired number of pieces, and wherever you want them in the box<sup>8</sup> without having to rewrite the entire code. Another constraint: to keep the figure as light as possible, only the facets actually seen should be displayed. In the code below, we keep the default viewing angles, and:

- the sugars are cubes of side 1 (we then modify the 3D matrix of the graph to "elongate" them),
- each piece is identified by the coordinates  $(x, y, z)$  of the upper right corner of the front face, with  $x$  an integer between 1 and  $Lg$ ,  $y$  an integer between 1 and  $lg$ , and  $z$  an integer between 1 and  $ht$ .
- to store the positions of the pieces, we use a three-dimensional matrix *positions*, one for  $x$ , one for  $y$ , and one for  $z$ , with the convention that *positions*[ $x$ ][ $y$ ][ $z$ ] is 1 if there is a sugar at position  $(x, y, z)$ , and 0 otherwise.
- For each piece, there are at most three visible faces: the top one, the right one, and the front one.<sup>9</sup>, but we only draw the top face if there isn't another sugar cube above it, we only draw the right face if there isn't another sugar cube to the right, and we only draw the front face if there isn't another sugar cube in front. This builds the list of facets actually seen.
- In the scene display, you must **put the box first**, otherwise its facets will be cut off by the planes of the sugar cube facets. The sugar cube facets cannot be cut off by the box because they are all inside.

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

local g = ld.graph3d:new{window={-9,8,-10,4},size={10,10}}
ld.Hiddenlines = false
local Lg, lg, ht = 5, 4, 3 -- length, width, height (box size)
local positions = {} -- 3-dimensional matrix initialized with 0s
for L = 1, Lg do
  local X = {}
  for l = 1, lg do
    local Y = {}
    for h = 1, ht do table.insert(Y,0) end
    table.insert(X,Y)
  end
end
```

<sup>7</sup>Problem posed in a forum, the objective being to use it as counting exercises for students.

<sup>8</sup>A piece must rest either on the bottom of the box or on top of another piece

<sup>9</sup>Provided you don't change the viewing angles!

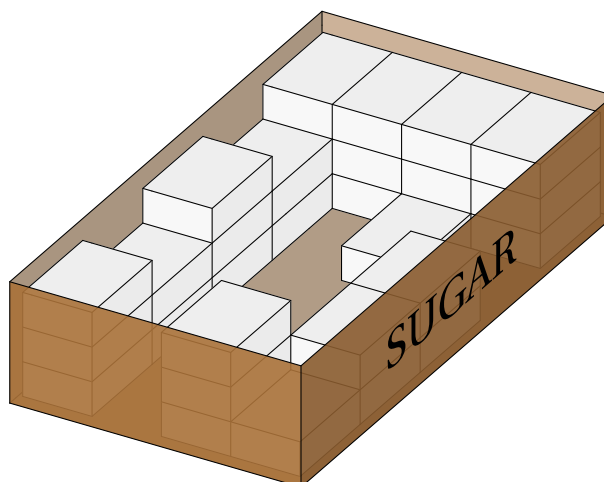


```

    end
    table.insert(positions,X)
end
local facetList = function() -- returns the list of facets to draw (pay attention to the orientation)
    local facet = {}
    for x = 1, Lg do -- loop over the positions matrix
        for y = 1, lg do
            for z = 1, ht do
                if positions[x][y][z] == 1 then -- there is a sugar in (x,y,z)
                    if (z == ht) or (positions[x][y][z+1] == 0) then -- no sugar on top so top side visible
                        table.insert(facet, {M(x,y,z),M(x-1,y,z),M(x-1,y-1,z),M(x,y-1,z)}) -- insert top face
                    end
                    if (y == lg) or (positions[x][y+1][z] == 0) then -- no sugar on the right so right side visible
                        table.insert(facet, {M(x,y,z),M(x,y,z-1),M(x-1,y,z-1),M(x-1,y,z)}) -- insert right face
                    end
                    if (x == Lg) or (positions[x+1][y][z] == 0) then -- no sugar in front so front side visible
                        table.insert(facet, {M(x,y,z),M(x,y-1,z),M(x,y-1,z-1),M(x,y,z-1)}) -- insert front face
                    end
                end
            end
        end
    end
    return facet
end
-- creation of the box (parallelepiped)
local O = Origin -0.1*M(1,1,1) -- so that the box does not stick to the sugars
local boite = ld.parallelep(O, (Lg+0.2)*vecI, (lg+0.2)*vecJ, (ht+0.5)*vecK)
table.remove(boite.facets,2) -- we remove the top of the box, this is facet number 2
-- sugars are positioned
for y = 1, 4 do for z = 1, 3 do positions[1][y][z] = 1 end end
for x = 2, 5 do for z = 1, 2 do positions[x][1][z] = 1 end end
for z = 1, 3 do positions[5][3][z] = 1 end
for z = 1, 2 do positions[4][4][z] = 1 end
for z = 1, 2 do positions[3][4][z] = 1 end
positions[5][1][3] = 1; positions[3][1][3] = 1; positions[5][4][1] = 1; positions[2][3][1] = 1
g:Setmatrix3d({Origin,3*vecI,2*vecJ,vecK}) -- expansion on Ox and Oy to "lengthen" the cubes...
g:Dscene3d( -- drawing
    g:addPoly(boite,{color="brown",edge=true,opacity=0.9}),
    g:addFacet(facetList(), {backcull=true,contrast=0.25,edge=true})
)
g:Labelsize("huge"); g:Dlabel3d( "SUGAR", M(Lg/2+0.1,lg+0.1,ht/2+0.1), {dir={-vecI,vecK}})
g:Show()
\end{luadraw}

```

Figure 35: Box of Sugar Cubes



## 2) Stack of Cubes

We can modify the previous example to draw a stack of randomly positioned cubes, with four views. We'll position the cubes by placing a random number per column, starting from the bottom. We'll create four views of the stack, adding axes to help us navigate between these different views. This slightly changes the search for potentially visible facets; there are five cases per cube, not just three (front, back, left, right, and top; we don't create bottom views). To make the stack more readable, we use three colors to paint the cube faces (two opposite faces have the same color).

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

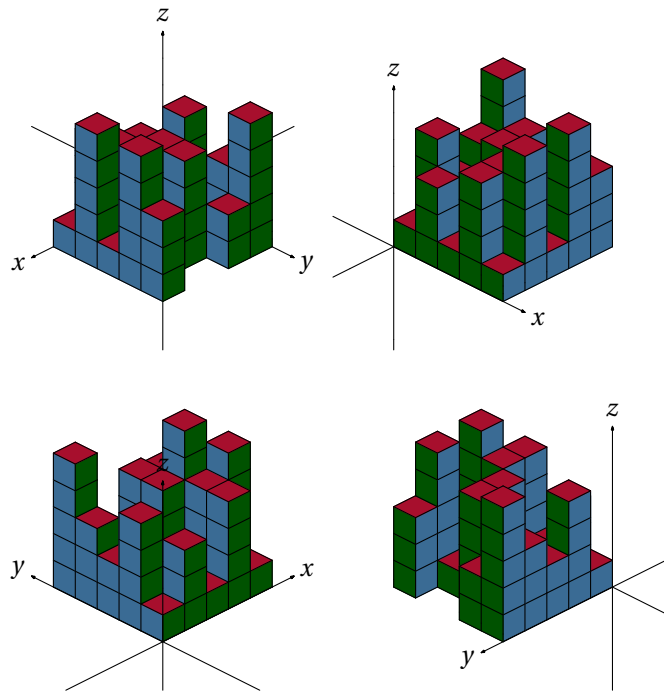
local g = ld.graph3d:new{window3d={-6,6,-6,6},size={10,10}}
ld.Hiddenlines = false
local Lg, lg, ht, a = 5, 5, 5, 2 -- length, width, height of the space to fill, size of a cube
local positions = {} -- 3-dimensional matrix initialized with 0s
for L = 1, Lg do
    local X = {}
    for l = 1, lg do
        local Y = {}
        for h = 1, ht do table.insert(Y,0) end
        table.insert(X,Y)
    end
    table.insert(positions,X)
end
for x = 1, Lg do -- random positioning of cubes
    for y = 1, lg do
        local nb = math.random(0,ht) -- we put number of cubes in the column (x,y,*) starting from the bottom
        for z = 1, nb do positions[x][y][z] = 1 end
    end
end
local dessus,gauche,devant = {},{},{},{} -- to memorize the facets
for x = 1, Lg do -- loop over the positions matrix to determine the facets to draw
    for y = 1, lg do
        for z = 1, ht do
            if positions[x][y][z] == 1 then -- there is a cube in (x,y,z)
                if (z == ht) or (positions[x][y][z+1] == 0) then -- no cube above so face up
                    table.insert(dessus,{M(x,y,z),M(x-1,y,z),M(x-1,y-1,z),M(x,y-1,z)}) -- insert top face
                end
                if (y == lg) or (positions[x][y+1][z] == 0) then -- no cube on the right so face up
                    table.insert(gauche,{M(x,y,z),M(x,y,z-1),M(x-1,y,z-1),M(x-1,y,z)}) -- insert right face
                end
                if (y == 1) or (positions[x][y-1][z] == 0) then -- no cube on the left so face up
                    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)}) -- insert left face
                end
                if (x == Lg) or (positions[x+1][y][z] == 0) then -- no cube in front so face up
                    table.insert(devant,{M(x,y,z),M(x,y-1,z),M(x,y-1,z-1),M(x,y,z-1)}) -- insert front face
                end
                if (x == 1) or (positions[x-1][y][z] == 0) then -- no cube behind so back face visible
                    table.insert(devant,{M(x-1,y,z),M(x-1,y,z-1),M(x-1,y-1,z-1),M(x-1,y-1,z)}) -- back face
                end
            end
        end
    end
end
g:Setmatrix3d({M(-a*Lg/2,-a*lg/2,-a*ht/2),a*vecI,a*vecJ,a*vecK}) -- to center the figure and have cubes of side a
local dessin = function()
    g:Dscene3d(
        g:addFacet(dessus, {backcull=true,color="Crimson"}), g:addFacet(gauche, {backcull=true,color="DarkGreen"}),
        g:addFacet(devant, {backcull=true,color="SteelBlue"}),
        g:addPolyline(ld.facetedges(ld.concat(dessus,gauche,devant))), -- drawing edges
        g:addAxes(Origin,{arrows=1})
    end
end
```

```

g:Saveattr(); g:Viewport(-5,0,0,5); g:Coordsystem(-11,11,-11,11); g:Setviewdir(45,60) -- top left
dessin(); g:Restoreattr()
g:Saveattr(); g:Viewport(0,5,0,5);g:Coordsystem(-11,11,-11,11); g:Setviewdir(-45,60) -- top right
dessin(); g:Restoreattr()
g:Saveattr(); g:Viewport(-5,0,-5,0);g:Coordsystem(-11,11,-11,11); g:Setviewdir(-135,60) -- bottom left
dessin(); g:Restoreattr()
g:Saveattr(); g:Viewport(0,5,-5,0);g:Coordsystem(-11,11,-11,11); g:Setviewdir(135,60) -- bottom right
dessin(); g:Restoreattr()
g:Show()
\end{luadraw}

```

Figure 36: Stack of Cubes



### 3) Illustration of Dandelin's Theorem

```

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

local g = ld.graph3d:new{window3d={-5,5,-5,5,-5,5}, window={-5,5,-5,6}, bg="lightgray",viewdir={-10,85}}
g:Linewidth(8)
local sqrt = math.sqrt
local sqr = function(x) return x*x end
local L, a = 4.5, 2
local R = (a+5)*L/sqrt(100+L^2) --large sphere center=M(0,0,a) radius=R
local S2 = ld.sphere(M(0,0,a),R,45,45)
local k = 0.35 --homothety ratio
local b, r = (a+5)*k-5, k*R -- small sphere center = M(0,0,b) radius = r
local S1 = ld.sphere(M(0,0,b),r,45,45)
local c = (b+k*a)/(1+k) --second center of homothety
local z = a+sqrt(R)/(c-a) --image of c under inversion with respect to the large sphere
local M1 = M(0,sqrt(sqr(R)-sqr(z-a)),z)--point of the large sphere and the tangent plane
local N = M1-M(0,0,a) --normal vector to the tangent plane
local plane = {M(0,0,c),-N} --tangent plane
local z2 = a+sqrt(R)/(-5-a) --image of the vertex under inversion with respect to the large sphere
local z1 = b+sqrt(r)/(-5-b) --image of the vertex by the inversion with respect to the small sphere
local P2 = M(sqrt(R^2-(z2-a)^2),0,z2)
local P1 = M(sqrt(r^2-(z1-b)^2),0,z1)
local S = M(0,0,-5)

```

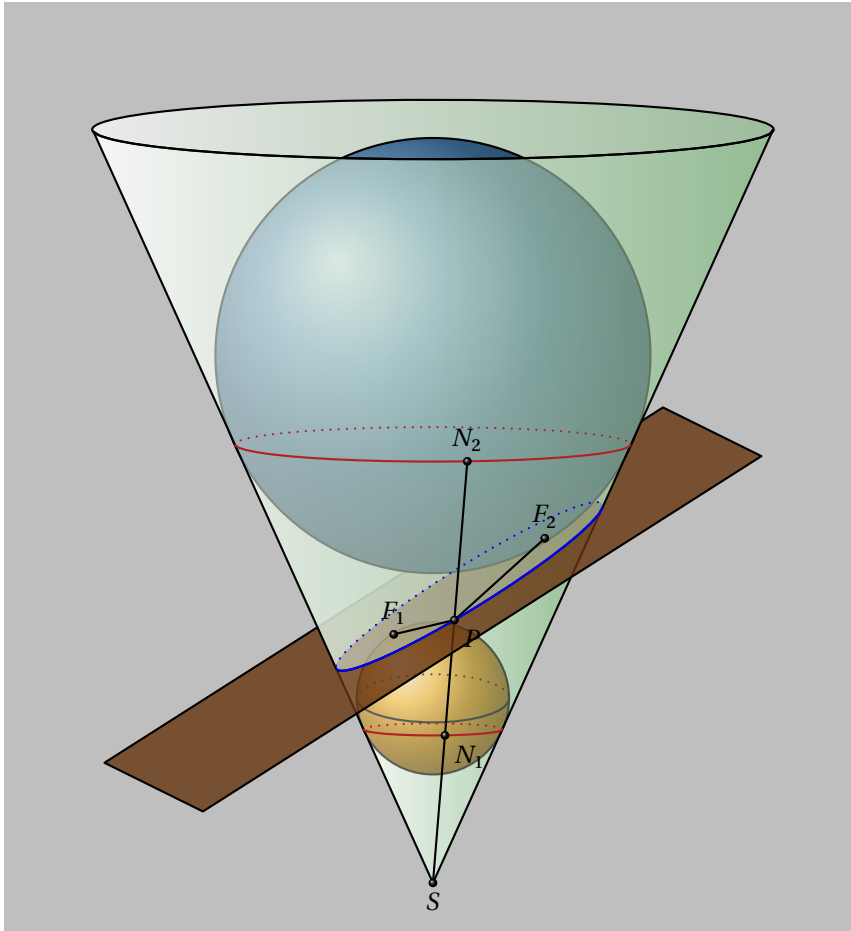
```

local P = ld.interDP({P1,P2-P1},plane)
local C = ld.cone(M(0,0,-5),10*vecK,L,45,true)
local ellips = g:Intersection3d(C,plane)
local plane1 = {M(0,0,z1),vecK}
local plane2 = {M(0,0,z2),vecK}
local L1, L2 = g:Intersection3d(S1,plane1), g:Intersection3d(S2,plane2)
local F1, F2 = ld.proj3d(M(0,0,b), plane), ld.proj3d(M(0,0,a), plane) --focal points
local s1, s2 = g:Proj3d(M(0,0,a)), g:Proj3d(M(0,0,b))
local V, H = g:Classifyfacet(C) -- separate visible and invisible facets
local V1, V2 = ld.cutfacet(V,plane)
local H1, H2 = ld.cutfacet(H,plane)

-- Drawing
-- faces not visible under the plane, fill only
g:Dpolyline3d( ld.border(H2),"left color=white, right color=DarkSeaGreen, draw=none")
g:Dsphere( M(0,0,b), r, {mode=mBorder,color="Orange"}) -- small sphere
-- visible faces below the plane
g:Dpolyline3d( ld.border(V2),"left color=white, right color=DarkSeaGreen, fill opacity=0.4")
g:Dpolyline3d({S,P}) -- segment [S,P] which is partially below the plane
g:Dfacet( g:Plane2facet(plane,0.75), {color="Chocolate", opacity=0.8}) -- the plane
-- outline of faces not visible above the plane, fill only
g:Dpolyline3d( ld.border(H1),"left color=white, right color=DarkSeaGreen,draw=none,fill opacity=0.7")
g:Dsphere( M(0,0,a),R, {mode=2,color="SteelBlue"}) -- large sphere
-- outline of faces visible above the plane
g:Dpolyline3d( ld.border(V1),"left color=white, right color=DarkSeaGreen, fill opacity=0.6")
g:Dcircle3d(M(0,0,5),L,vecK) -- opening of the cone
g:Dpolyline3d({{P,F1},{F2,P,P2}})
g:Dedges(L1,{hidden=true,color="FireBrick"})
g:Dedges(L2,{hidden=true,color="FireBrick"})
g:Dedges(ellips,{hidden=true, color="blue"})
g:Dballdots3d({F1,F2,S,P1,P,P2},nil,0.75)
g:Dlabel3d( "$F_1$",F1,{pos="N"}, "$F_2$",F2,{}, "$N_2$",P2,{}, "$S$",S,{pos="S"},
"$N_1$",P1,{pos="SE"}, "$P$",P,{pos="SE"} )
g:Show()
\end{luadraw}

```

Figure 37: Illustration of Dandelin's Theorem



We want to draw a cone with a section through a plane and two spheres inside this cone (and tangent to the plane), but without drawing any spheres or faceted cones. The starting point, however, is the creation of these faceted solids, the spheres  $S1$  and  $S2$  (lines 11 and 8 of the listing) as well as the cone  $C$  in line 23. The drawing principle is as follows:

1. We separate the facets of the cone into two categories: the visible facets (facing the observer) and the others (variables  $V$  and  $H$  in line 30), which actually correspond to the front of the cone and the back of the cone.
2. We divide the two lists of facets with the plane (lines 31 and 32). Thus,  $V1$  corresponds to the front facets located above the plane and  $V2$  corresponds to the front facets located below the plane (same thing with  $H1$  and  $H2$  for the back).
3. We then draw the outline of  $H2$  with a gradient fill (only) (line 34).
4. We draw the small sphere (in orange, line 35).
5. We draw the outline of  $V2$  with a gradient fill and transparency to see the small sphere (line 36).
6. We draw the segment  $[S, P]$  (line 37) then the plane as a transparent facet (line 38).
7. We draw the outline of  $H1$  with a gradient fill (line 39). This is the back part above the plane.
8. We draw the large sphere (line 40).
9. Finally, we draw the outline of  $V1$  with a gradient fill (line 41) and transparency to see the sphere (this is the front part of the cone above the plane), then the opening of the cone (line 42).
10. We draw the intersections between the cone and the spheres (lines 44 and 45) as well as between the cone and the plane (line 46).

#### 4) Volume defined by a double integral

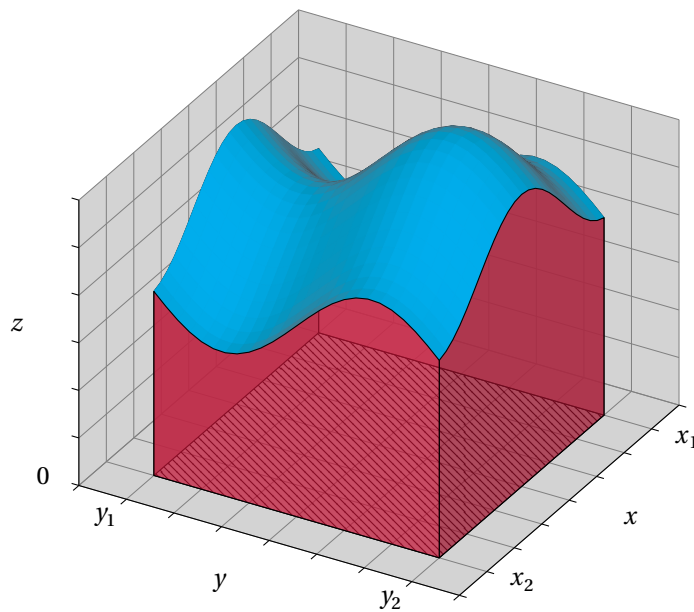
```
\begin{luadraw}{name=volume_integrale}
local ld = luadraw
local M = ld.pt3d.M
```

```

local pi, sin, cos = math.pi, math.sin, math.cos

local g = ld.graph3d:new{window3d={-4,4,-4,4,0,6},adjust2d=true,margin={0,0,0,0},size={10,10}}
local x1, x2, y1, y2 = -3,3,-3,3 -- bornes
local f = function(x,y) return cos(x)+sin(y)+5 end -- function to integrate
local p = function(u,v) return M(u,v,f(u,v)) end -- surface parameterization z=f(x,y)
local Fx1 = ld.concat({ld.pxy(p(x1,y2)), ld.pxy(p(x1,y1))}, ld.parametric3d(function(t) return p(x1,t)
  ↪ end,y1,y2,25,false,0)[1])
local Fx2 = ld.concat({ld.pxy(p(x2,y1)), ld.pxy(p(x2,y2))}, ld.parametric3d(function(t) return p(x2,t)
  ↪ end,y2,y1,25,false,0)[1])
local Fy1 = ld.concat({ld.pxy(p(x1,y1)), ld.pxy(p(x2,y1))}, ld.parametric3d(function(t) return p(t,y1)
  ↪ end,x2,x1,25,false,0)[1])
local Fy2 = ld.concat({ld.pxy(p(x2,y2)), ld.pxy(p(x1,y2))}, ld.parametric3d(function(t) return p(t,y2)
  ↪ end,x1,x2,25,false,0)[1])
g:Dboxaxes3d({grid=true, gridcolor="gray",fillcolor="LightGray",labels=false})
g:Filloptions("fdiag", "black"); g:Dpolyline3d( {M(x1,y1,0),M(x1,y2,0),M(x2,y2,0),M(x2,y1,0)} ) -- below
g:Dfacet( {Fx1,Fy1},{mode=ld.mShaded,opacity=0.7,color="Crimson"} )
g:Dfacet(ld.surface(p,x1,x2,y1,y2), {mode=ld.mShadedOnly,color="cyan"})
g:Dfacet( {Fx2,Fy2},{mode=ld.mShaded,opacity=0.7,color="Crimson"} )
g:Dlabel3d("$x_1$", M(x1,4.75,0),{},{}, "$x_2$", M(x2,4.75,0),{},{},
"$y_1$", M(4.75,y1,0),{},{}, "$y_2$", M(4.75,y2,0),{},{}, "$0$",M(4,-4.75,0),{},{})
g:Show()
\end{luadraw}

```

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

Here, the solid represented has lateral faces ( $Fx1$ ,  $Fx2$ ,  $Fy1$  and  $Fy2$ ) with one side being a parametric curve. We therefore take the points of this parametric curve (its first connected component) and add the projections of the two ends onto the  $xy$ -plane. Care must be taken with the direction of travel so that the faces are correctly oriented (normal outward). This normal is calculated from the first three points of the face; it is best to start the face with the two projections onto the plane to be sure of the orientation. We draw the bottom first, then the lateral faces, and finish with the surface.

## 5) Volume defined on something other than a block

```

\begin{luadraw}{name=volume2}
local ld = luadraw
local M = ld.pt3d.M
local pi, sin, cos = math.pi, math.sin, math.cos

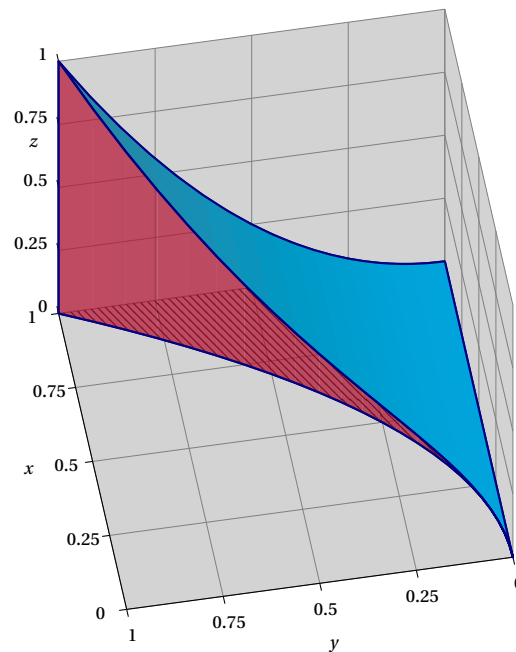
local g = ld.graph3d:new{window3d={0,1,0,1,0,1}, margin={0,0,0,0},adjust2d=true,viewdir={170,40}, size={10,10}}

```

```

g:Labelsize("scriptsize")
local f = function(t) return M(t,t^2,0) end
local h = function(t) return M(1,t,t^2) end
local C = ld.parametric3d(f,0,1,25,false,0)[1] -- curve y=x^2 in the plane z=0 (first connected component)
local D = ld.parametric3d(h,1,0,25,false,0)[1] -- curve z=y^2 in the plane x=1, in the opposite direction
local dessous = ld.concat({M(1,0,0)},C) -- forms the underside
local arriere = ld.concat({M(1,1,0)},D) -- forms the back face
local avant, dessus, A, B = {}, {}, nil, C[1]
for k = 2, #C do --We construct the front and top faces facet by facet, starting from the points of C
    A = B; B = C[k]
    table.insert(avant, {B,A,M(A.x,A.y,A.y^2),M(B.x,B.y,B.y^2)})
    table.insert(dessus, {M(B.x,B.y,B.y^2),M(A.x,A.y,A.y^2),M(1,A.y,A.y^2),M(1,B.y,B.y^2)})
end
g:Dboxaxes3d({grid=true, gridcolor="gray",fillcolor="LightGray", drawbox=false,
    xyzstep=0.25, xlabelstyle="W",ylabelstyle="W",zlabelstyle="W",zlabelsep=0})
g:Lineoptions(nil,"Navy",8)
g:Dpolyline3d(arriere,close,"fill=Crimson", fill_opacity=0.6) -- rear face (flat)
g:Filloptions("fdiag","black"); g:Dpolyline3d(dessous,close) -- below
g:Dmixfacet(avant,{color="Crimson",opacity=0.7,mode=ld.mShadedOnly}, dessous,{color="cyan",opacity=1})
g:Filloptions("none"); g:Dpolyline3d(ld.concat(ld.border(avant),ld.border(dessus)))
g:Show()
\end{luadraw}

```

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

In this example, the surface has the equation  $z = y^2$  (parabolic cylinder), but we are no longer on a block. The front face is not flat; we construct it like a cylinder (line 14) with vertical facets resting on curve  $C$  at the bottom, and on curve  $t \mapsto M(t, t^2, t^4)$  at the top.

Similarly, the top face (the surface) is constructed like a horizontal cylinder resting on curves  $D$  and  $t \mapsto M(t, t^2, t^4)$ .

We could not construct the surface by hand (called *dessus* in the code), and instead draw the following surface (after the front face):

```

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

```

but it has many more facets ( $25 \times 25$ ) than the cylinder-shaped construction (21 facets), which is less interesting.

# Appendices

## I Extensions

### 1) The *luadraw\_polyhedrons* module

This module is still in draft form and is expected to be expanded in the future. As its name suggests, it contains the definition of polyhedra. All numerical data comes from the [Visual Polyhedra](#) website.

**Usage:** This module does not add new graphics methods, but it returns a table of functions for constructing polyhedra. Therefore, it is used as follows (example):

```
local poly = require 'luadraw_polyhedrons'
local M = pt3d.M
local T = poly.tetrahedron( M(0,0,0), M(1,1,1) )
```

All functions follow the same model: **<name>(C, S [, all])** where *C* is the center of the polyhedron (3D point) and *S* is a vertex of the polyhedron (3D point). When *C* or *S* have the value *nil*, the untransformed polyhedron (centered at the origin) is returned. The optional argument *all* is a boolean. When it has the value *true*, the function returns four things: *P*, *V*, *E*, *F* where:

- *P* is the solid as a polyhedron,
- *V* the list (table) of vertices,
- *E* the list (table) of edges (with 3D points),
- *F* the list of facets (with 3D points). Some polyhedra have multiple facet types; in this case, the returned result is of the form: *P*, *V*, *E*, *F1*, *F2*, ..., where *F1*, *F2*, ... are lists of facets. This can allow them to be drawn with different colors, for example.

The argument *all* is set to *false*, which is the default value; the function only returns the polyhedron.

Here are the solids currently contained in this module:

- The Platonic solids, these solids have only one face type:
  - The function **tetrahedron(C, S [, all])** allows the construction of a regular tetrahedron with center *C* (3d point) and one vertex at *S* (3d point).
  - The function **octahedron(C, S [, all])** allows the construction of an octahedron with center *C* (3d point) and one vertex at *S* (3d point).
  - The function **cube(C, S [, all])** allows the construction of a cube with center *C* (3d point) and one vertex at *S* (3d point).
  - The function **icosahedron(C, S [, all])** allows the construction of an icosahedron with center *C* (3d point) and one vertex at *S* (3d point).
  - The function **dodecahedron(C, S [, all])** allows the construction of a dodecahedron with center *C* (3d point) and one vertex at *S* (3d point).



- The Archimedean Solids:

- The function **cuboctahedron**(*C*, *S* [, *all*]) allows the construction of a cuboctahedron with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **icosidodecahedron**(*C*, *S* [, *all*]) allows the construction of an icosidodecahedron with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **lsnubcube**(*C*, *S* [, *all*]) allows the construction of a snub cube (form 1) with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **lsnubdodecahedron**(*C*, *S* [, *all*]) allows the construction of a snub dodecahedron (form 1) with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **rhombicosidodecahedron**(*C*, *S* [, *all*]) allows the construction of a rhombicosidodecahedron with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has three types of faces.
- The function **rhombicuboctahedron**(*C*, *S* [, *all*]) allows the construction of a rhombicuboctahedron with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **rlnubcube**(*C*, *S* [, *all*]) allows the construction of a snub cube (shape 2) with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **rlnubdodecahedron**(*C*, *S* [, *all*]) allows the construction of a snub dodecahedron (shape 2) with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **truncatedcube**(*C*, *S* [, *all*]) allows the construction of a truncated cube with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **truncatedcuboctahedron**(*C*, *S* [, *all*]) allows the construction of a truncated cuboctahedron with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has three types of faces.
- The function **truncateddodecahedron**(*C*, *S* [, *all*]) allows the construction of a truncated dodecahedron with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **truncatedicosahedron**(*C*, *S* [, *all*]) allows the construction of a truncated icosahedron with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **truncatedicosidodecahedron**(*C*, *S* [, *all*]) allows the construction of a truncated icosidodecahedron with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **truncatedoctahedron**(*C*, *S* [, *all*]) allows the construction of a truncated octahedron with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **truncatedtetrahedron**(*C*, *S* [, *all*]) allows the construction of a truncated tetrahedron with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.

- Other solids:

- The function **octahemioctahedron**(*C*, *S* [, *all*]) allows the construction of an octahemioctahedron with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has two types of faces.
- The function **small\_stellated\_dodecahedron**(*C*, *S* [, *all*]) allows the construction of a small stellated dodecahedron with center  $\langle C \rangle$  (3d point) and one vertex at  $\langle S \rangle$  (3d point). This solid has only one type of face.

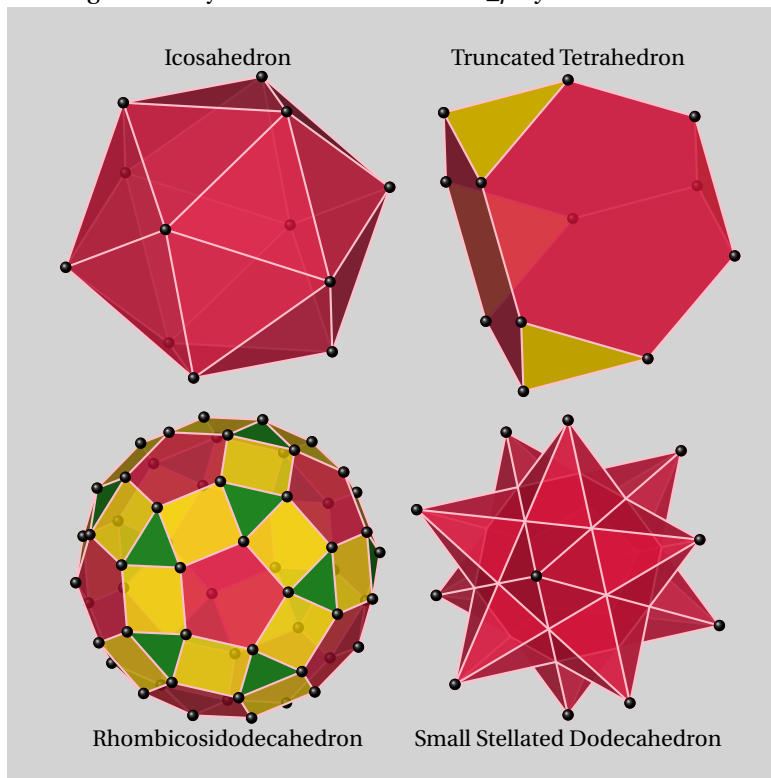
```
\begin{luadraw}{name=polyhedrons}
local ld = luadraw
local M, Origin = ld.pt3d.M, ld.pt3d.Origin
local i = ld.cpx.I
local poly = require 'luadraw_polyhedrons' -- chargement du module

local g = ld.graph3d:new{bg="LightGray", size={10,10}}
g:Labelsize("small"); ld.Hiddenlines = false
-- en haut à gauche
g:Saveattr(); g:Viewport(-5,0,0,5); g:Coordsystem(-5,5,-5,5,true)
local T,S,A,F = poly.icosahedron(Origin,M(0,2,4.5),true)
g:Dscene3d(
```

```

g:addFacet(F, {color="Crimson",opacity=0.8}),
g:addPolyline(A, {color="Pink", width=8}),
g:addDots(S) )
g:Dlabel("Icosahedron",5*i,{})
g:Restoreattr()
-- en haut à droite
g:Saveattr()
g:Viewport(0,5,0,5); g:Coordsystem(-5,5,-5,5,true)
local T,S,A,F1,F2 = poly.truncatedtetrahedron(Origin,M(0,0,5),true) -- Full output, display in a 3D scene
g:Dscene3d(
  g:addFacet(F1, {color="Crimson",opacity=0.8}),
  g:addFacet(F2, {color="Gold"}),
  g:addPolyline(A, {color="Pink", width=8}),
  g:addDots(S) )
g:Dlabel("Truncated Tetrahedron",5*i,{})
g:Restoreattr()
-- en bas à gauche
g:Saveattr(); g:Viewport(-5,0,-5,0); g:Coordsystem(-5,5,-5,5,true)
local T,S,A,F1,F2,F3 = poly.rhombicosidodecahedron(Origin,M(0,0,4.5),true)
g:Dscene3d(
  g:addFacet(F1, {color="Crimson",opacity=0.8}),
  g:addFacet(F2, {color="Gold",opacity=0.8}), g:addFacet(F3, {color="ForestGreen"}),
  g:addPolyline(A, {color="Pink", width=8}), g:addDots(S) )
g:Dlabel("Rhombicosidodecahedron",-5*i,{})
g:Restoreattr()
-- en bas à droite
g:Saveattr(); g:Viewport(0,5,-5,0); g:Coordsystem(-5,5,-5,5,true)
local T,S,A,F1 = poly.small_stellated_dodecahedron(Origin,M(0,0,5),true)
g:Dscene3d(
  g:addFacet(F1, {color="Crimson",opacity=0.8}),
  g:addPolyline(A, {color="Pink", width=8}),
  g:addDots(S) )
g:Dlabel("Small Stellated Dodecahedron",-5*i,{})
g:Restoreattr()
g:Show()
\end{luadraw}

```

Figure 1: Polyhedra from the *luadraw\_polyhedrons* module

## 2) The *luadraw\_spherical* Module

This module allows you to draw a number of objects on a sphere (such as circles, spherical triangles, etc) without having to manually manage the visible or invisible parts. Drawing is done in three steps:

1. We define the characteristics of the sphere (center, radius, color, etc.)
2. We define the objects to be added to the scene using dedicated methods.
3. We display everything with the **g:Dspherical()** method.

Of course, all 2D and 3D drawing methods remain usable.

**Usage:** This module adds new graphics methods to the *ld.graph3d* class; it does not return anything. The variables introduced by this module go in the *luadraw* namespace.

### Global Module Functions

- **ld.sM(x, y, z):** returns a point on the sphere; this is the point  $I$  on the sphere such that the half-line  $[O, I)$  ( $O$  being the center of the sphere) passes through the point  $A$  with Cartesian coordinates  $(x, y, z)$ . It is the projection of the point  $M(x, y, z)$  onto the sphere from the center.
- **ld.sM(theta, phi):** where  $\langle theta \rangle$  and  $\langle phi \rangle$  are angles in degrees, returns a point on the sphere, whose spherical coordinates are  $(R, theta, phi)$  where  $R$  is the radius of the sphere.
- **ld.toSphere(A):** returns the projection of point  $\langle A \rangle$  onto the sphere from the center.

### Sphere Definition

The sphere is defined using the method **g:Define\_sphere( options )**, where  $\langle options \rangle$  is a table allowing adjustment of each parameter. These are as follows (with their default values):

- **center=pt3d.Origin,**
- **radius=3,**
- **color="orange",**
- **opacity=1,**
- **mode=ld.mBorder,** sphere display mode (possible values: **ld.mWireframe** or **ld.mGrid** or **ld.mBorder**),
- **edgecolor="lightgray",**
- **edgestyle="solid",**
- **hiddenstyle=ld.Hiddenlinestyle,**
- **hiddencolor="gray",**
- **edgewidth=4,**
- **show=true,** to show or hide the sphere,
- **insidelabelcolor="darkgray":** Defines the color of labels whose anchor point is inside the sphere,
- **arrowBstyle="->":** Type of arrow at the end of the line,
- **arrowAstyle="<-":** Type of arrow at the beginning of the line,
- **arrowABstyle="<->":** Very rarely used because most of the time the lines drawn on the sphere must be cut,
- **hiddendelayed=false:** with the value **false** the hidden parts are drawn at the end of the instruction **g:Dspherical()**, with the value **true** they are drawn at the very end of the current graph which can be useful if you have added elements after the sphere which hide part of it (However, this behavior can be modified locally with the option **hidden=true/false**).

The **g:Clear\_spherical()** method allows you to remove objects that have been added to the scene, and resets the values to default.

**Add a circle: g:DScircle**

The **g:DScircle(P, options)** method allows you to add a circle to the sphere. The argument  $P$  is a table of the form  $\{A, n\}$  that represents a plane (passing through  $A$  and normal to  $n$ , two 3D points). The circle is then defined as the intersection of this plane with the sphere. The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `style=<current line style>`,
- `color=<current line color>`,
- `width=<current line thickness in tenths of a point>`,
- `opacity=<current line opacity>`,
- `hidden=ld.Hiddenlines`,
- `out=nil`, if we assign a list variable to this  $\langle out \rangle$  parameter, then the function adds to this list the two points corresponding to the ends of the hidden arc, if any, which allows us to retrieve them without having to calculate them.

**Add a great circle: g:DSbigcircle**

The method **g:DSbigcircle(AB, options)** adds a great circle to the sphere. The argument  $\langle AB \rangle$  is a table of the form  $\{A, B\}$  where  $A$  and  $B$  are two distinct points on the sphere. The great circle is then the circle centered at the center of the sphere, and passing through  $A$  and  $B$ . The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `style=<current line style>`,
- `color=<current line color>`,
- `width=<current line thickness in tenths of a point>`,
- `opacity=<current line opacity>`,
- `hidden=ld.Hiddenlines`,
- `out=nil`, if we assign a list variable to this  $\langle out \rangle$  parameter, then the function adds to this list the two points corresponding to the ends of the hidden arc, if any, which allows us to retrieve them without having to calculate them.

**Add a great circle arc: g:DSarc**

The method **g:DSarc(AB, sens, options)** allows you to add a great circle arc to the sphere. The argument  $\langle AB \rangle$  is a table of the form  $\{A, B\}$  where  $A$  and  $B$  are two distinct points on the sphere. The great circle arc is then drawn from  $A$  to  $B$ . The argument  $\langle sens \rangle$  is equal to 1 or  $-1$  to indicate the direction of the arc. When  $A$  and  $B$  are not diametrically opposed, the plane  $OAB$  (where  $O$  is the center of the sphere) is oriented with  $\vec{OA} \wedge \vec{OB}$ . The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `style=<current line style>`,
- `color=<current line color>`,
- `width=<current line thickness in tenths of a point>`,
- `opacity=<current line opacity>`,
- `hidden=ld.Hiddenlines`,
- `arrows=0`, three possible values: 0 (no arrow), 1 (one arrow at  $B$ ), 2 (arrow at  $A$  and  $B$ ).
- `normal=nil`, allows you to specify a normal vector to the  $OAB$  plane when these three points are aligned.

**Add an angle: `g:DSangle`**

The method **`g:DSangle(B, A, C, r, sens, options)`**, where  $\langle A \rangle$ ,  $\langle B \rangle$ , and  $\langle C \rangle$  are three points on the sphere, allows you to draw a great circle arc on the sphere to represent the angle  $(\vec{AB}, \vec{AC})$  with a radius of  $\langle r \rangle$ . The argument  $\langle sens \rangle$  is 1 or  $-1$  to indicate the direction of the arc; the plane  $ABC$  is oriented with  $\vec{AB} \wedge \vec{AC}$ . The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `style=<current line style>`,
- `color=<current line color>`,
- `width=<current line thickness in tenths of a point>`,
- `opacity=<current line opacity>`,
- `hidden=ld.Hiddenlines`,
- `arrows=0`, three possible values: 0 (no arrow), 1 (one arrow at  $B$ ), 2 (arrow at  $A$  and  $B$ ).
- `normal=nil`, allows you to specify a normal vector to the  $OAB$  plane when these three points are aligned.

**Add a spherical facet: `g:DSfacet`**

The method **`g:DSfacet(F, options)`**, where  $\langle F \rangle$  is a list of points on the sphere, allows you to draw the facet represented by  $\langle F \rangle$ , the edges being great circle arcs. The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `style=<current line style>`,
- `color=<current line color>`,
- `width=<current line thickness in tenths of a point>`,
- `opacity=<current line opacity>`,
- `hidden=ld.Hiddenlines`,
- `fill=""`, string representing the fill color (none by default),
- `fillopacity=0.3`, opacity of the fill color.

**Add a spherical curve: `g:DScurve`**

The method **`g:DScurve(L, options)`**, where  $\langle L \rangle$  is a list of points on the sphere, allows you to draw the curve represented by  $\langle L \rangle$ . The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `style=<current line style>`,
- `color=<current line color>`,
- `width=<current line thickness in tenths of a point>`,
- `opacity=<current line opacity>`,
- `hidden=ld.Hiddenlines`,
- `out=nil`. If we assign a table-type variable to this `out` option, then the function adds the points corresponding to the ends of the hidden parts to this list.

We will now deal with objects that are not necessarily on the sphere, but that may pass through it, or be inside it, or outside it.

**Add a segment: `g:DSseg`**

The **`g:DSseg(AB, options)`** method allows you to add a segment. The argument  $\langle AB \rangle$  is a table of the form  $\{A, B\}$  where  $A$  and  $B$  are two points in space. The function handles interactions with the sphere. The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `style=<current line style>`,
- `color=<current line color>`,
- `width=<current line thickness in tenths of a point>`,
- `opacity=<current line opacity>`,
- `hidden=ld.Hiddenlines`,
- `arrows=0`, three possible values: 0 (no arrow), 1 (one arrow in  $B$ ), 2 (arrow in  $A$  and  $B$ ).

**Add a line: `g:DSline`**

The **`g:DSline(d, options)`** method allows you to add a line. The argument  $\langle d \rangle$  is a table of the form  $\{A, u\}$  where  $A$  is a point on the line and  $u$  is a direction vector (two 3D points). The function handles interactions with the sphere. The drawn segment is obtained by intersecting the line with the 3D window; it may be empty if the window is too narrow. The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `style=<current line style>`,
- `color=<current line color>`,
- `width=<current line thickness in tenths of a point>`,
- `opacity=<current line opacity>`,
- `hidden=ld.Hiddenlines`,
- `arrows=0`, three possible values: 0 (no arrow), 1 (one arrow in  $B$ ), 2 (arrow in  $A$  and  $B$ ).
- `scale=1`, allows you to change the size of the plotted segment.

**Add a polygonal line: `g:DSpolyline`**

The **`g:DSpolyline(L, options)`** method allows you to add a polygonal line. The argument  $\langle L \rangle$  is a list of points in space, or a list of lists of points in space. The function handles interactions with the sphere. The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `style=<current line style>`,
- `color=<current line color>`,
- `width=<current line thickness in tenths of a point>`,
- `opacity=<current line opacity>`,
- `hidden=ld.Hiddenlines`,
- `arrows=0`, three possible values: 0 (no arrow), 1 (one arrow in  $B$ ), 2 (arrow in  $A$  and  $B$ ).
- `close=false`, indicates whether the line should be closed.

**Add a plane: `g:DSplane`**

The **`g:DSplane(P, options)`** method allows you to add the contour of a plane. The argument  $\langle P \rangle$  is a table of the form  $\{A, n\}$ , where  $A$  is a point on the plane and  $n$  is a normal vector. The function draws a parallelogram representing the plane  $\langle P \rangle$ , processing the interactions with the sphere. The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `style=<current line style>`,
- `color=<current line color>`,
- `width=<current line thickness in tenths of a point>`,
- `opacity=<current line opacity>`,
- `hidden=ld.Hiddenlines`,
- `scale=1`, allows you to change the size of the parallelogram,
- `angle=0`, angle in degrees, allows you to rotate the parallelogram around the perpendicular line passing through the center of the sphere.
- `trace=true`, allows you to draw, or not, the intersection of the plane with the sphere when it is not empty.

**Add a label: `g:DLabel`**

The **`g:DLabel(text1, anchor1, options1, text2, anchor2, options2, ...)`** method allows you to add one or more labels using the same principle as the **`g:Dlabel3d()`** method, except that here the function handles cases where the anchor point is inside the sphere, behind the sphere, or in front of the sphere. When it is inside, the label color is given by the sphere option `insidelabelcolor`, which defaults to `"darkgray"`.

**Add points: `g:DSdots` and `g:DSstars`**

The **`g:DSdots(dots, options)`** method allows you to add points to the scene. The  $\langle dots \rangle$  argument is a list of 3D points. The function draws points by managing interactions with the sphere. The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `hidden=ld.Hiddenlines`,
- `mark_options=""`, a string that will be passed directly to the `\draw` instruction.

If a point is inside the sphere, or on the hidden face, the point's color is given by sphere option `insidelabelcolor`, which defaults to `"darkgray"`.

The **`g:DSstars(dots, options)`** method allows you to add points **onto** the sphere. The  $\langle dots \rangle$  argument is a list of 3D points that will be projected onto the sphere. The function draws these points as an asterisk. The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `style=<current line style>`,
- `color=<current line color>`,
- `width=<current line thickness in tenths of a point>`,
- `opacity=<current line opacity>`,
- `hidden=ld.Hiddenlines`,
- `scale=1`, allows you to change the size,
- `circled=false`, allows you to add a circle around the star,
- `fill=""`, string representing a color. When not empty, the asterisk is replaced by a circled hexagonal facet and filled with the color specified by this option.

The points on the hidden face of the sphere have the color given by the sphere option `insidelabelcolor`, which defaults to `"darkgray"`.

### Inverse Stereography: `g:DSinvstereo_curve` and `g:DSinvstereo_polyline`

The method `g:DSinvstereo_curve(L, options)`, where  $\langle L \rangle$  is a 3D polygonal line representing a curve drawn on a plane with equation  $z = cte$ , draws the image of  $\langle L \rangle$  on the sphere by inverse stereography, the pole being the point  $C+r*vecK$ , where  $C$  is the center of the sphere and  $r$  is the radius.

The method `g:DSinvstereo_polyline(L, options)`, where  $\langle L \rangle$  is a 3D polygonal line drawn on a plane with equation  $z = cte$ , draws the image of  $\langle L \rangle$  on the sphere by inverse stereography, the pole being the point  $C+r*vecK$ , where  $C$  is the center of the sphere and  $r$  is the radius.

In both cases, the  $\langle options \rangle$  are the same as for the `g:DScurve()` method.

### Examples

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

local g = ld.graph3d:new{window={-9,9,-4,5},viewdir={25,70},size={16,8}}
require 'luadraw_spherical'
g:Linewidth(6); ld.Hiddenlinestyle = "dashed"
local a = 4
local O = Origin
local cube = ld.parallelelep(0,a*vecI,a*vecJ,a*vecK)
local G = pt3d.isobar3d(cube.vertices)
cube = ld.shift3d(cube,-G) -- to center the cube at the origin
local R = pt3d.abs(cube.vertices[1])

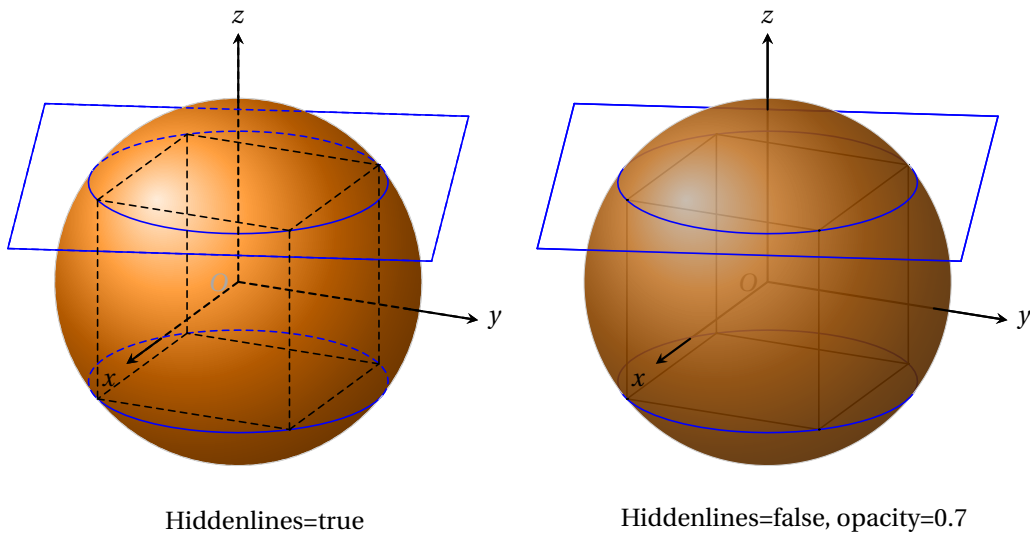
local dessin = function()
    g:DSpolyline({{0,5*vecI},{0,5*vecJ},{0,5*vecK}}, {arrows=1, width=8}) -- axes
    g:DSPlane({a/2*vecK,vecK},{color="blue",scale=0.9,angle=20});
    g:DScircle({-a/2*vecK,vecK},{color="blue"})
    g:DSpolyline( ld.facetedges(cube) ); g:DLabel("$0$",0,{pos="W"})
    g:Dspherical()
end

g:Saveattr(); g:Viewport(-9,0,-4,5); g:Coordsystem(-5,5,-5,5)
ld.Hiddenlines = true; g:Define_sphere({radius=R, arrowBstyle = "-stealth"})
dessin()
g:Dlabel3d("$x$",5*vecI,{pos="SW"}, "$y$",5*vecJ,{pos="E"}, "$z$",5*vecK,{pos="N"})
g:Dlabel("Hiddenlines=true",0.5-4.5*cpx.I,{})
g:Restoreattr()

g:Saveattr(); g:Viewport(0,9,-4,5); g:Coordsystem(-5,5,-5,5)
ld.Hiddenlines = false; g:Define_sphere({radius=R,opacity=0.7, arrowBstyle = "-stealth"} )
dessin()
g:Dlabel3d("$x$",5*vecI,{pos="SW"}, "$y$",5*vecJ,{pos="E"}, "$z$",5*vecK,{pos="N"})
g:Dlabel("Hiddenlines=false, opacity=0.7",0.5-4.5*cpx.I,{})
g:Restoreattr()
g:Show()
\end{luadraw}
```



Figure 2: Cube in a Sphere



### Spherical curve

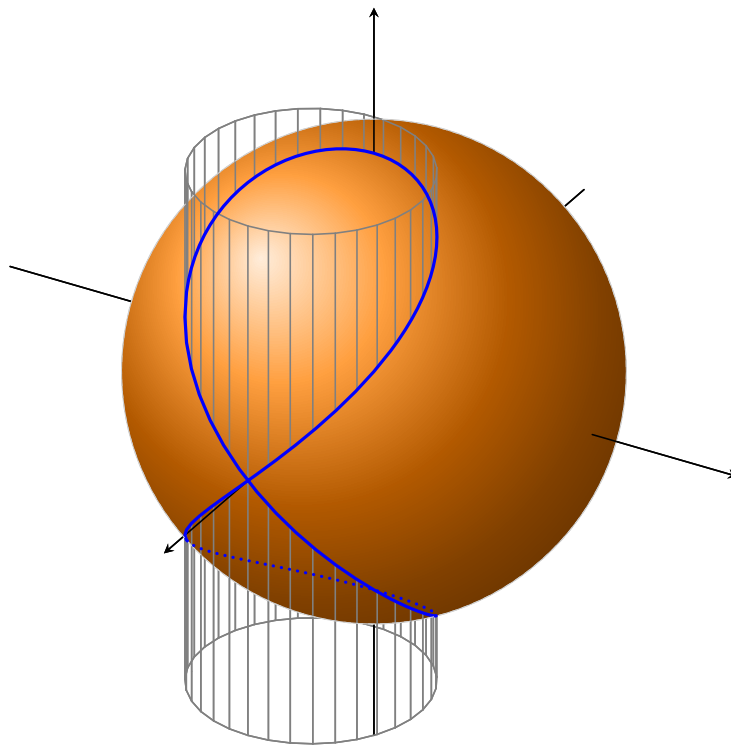
```

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

local g = ld.graph3d:new{window={-4.5,4.5,-4.5,4.5},viewdir={30,60},margin={0,0,0,0},size={10,10}}
require 'luadraw_spherical'
g:Linewidth(6); ld.Hiddenlinestyle = "dotted"
ld.Hiddenlines = false;
local C = ld.cylinder(M(1.5,0,-3.5),1.5,M(1.5,0,3.5),35,true)
local L = ld.parametric3d( function(t) return Ms(3,t-math.pi/2,t) end, -math.pi,math.pi) -- the curve
g:Define_sphere({arrowBstyle = "-stealth"})
g:DSpolyline(ld.facetedges(C),{color="gray"}) -- drawing cylinder
g:DSpolyline({{-5*vecI,5*vecI},{-5*vecJ,5*vecJ},{-5*vecK,5*vecK}},{arrows=1}) --axes
ld.Hiddenlines=true; g:DScurve(L,{width=12,color="blue"}) -- curve with hidden part
g:Dspherical()
g:Show()
\end{luadraw}

```

Figure 3: Viviani window



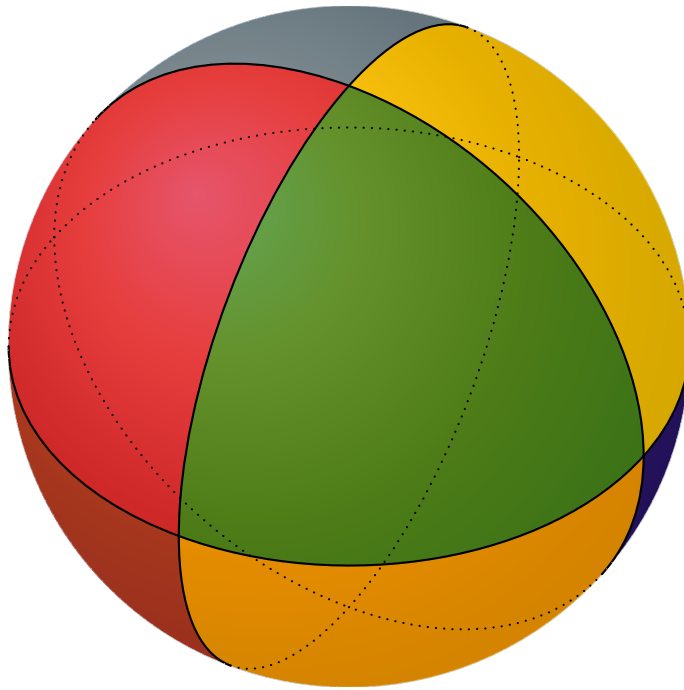
To avoid compromising the readability of the drawing, the hidden parts have not been displayed except for the curve.

### A spherical tiling

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

local g = ld.graph3d:new{window={-3,3,-3,3},viewdir={30,60},size={10,10}}
require 'luadraw_spherical'
local poly = require "luadraw_polyhedrons"
g:Linewidth(6); ld.Hiddenlines = true; ld.Hiddenlinestyle = "dotted"
local P = ld.poly2facet( poly.octahedron(Origin, ld.sM(30,10)) )
local colors = {"Crimson","ForestGreen","Gold","SteelBlue","SlateGray","Brown","Orange","Navy"}
g:Define_sphere()
for k,F in ipairs(P) do
    g:DSfacet(F,{fill=colors[k],style="noline",fillopacity=0.7}) --facets without edges
end
for _, A in ipairs(ld.facetedges(P)) do
    g:DSarc(A,1,{width=8}) -- each edge is an arc of a great circle
end
g:Dspherical()
g:Show()
\end{luadraw}
```

Figure 4: A spherical tiling



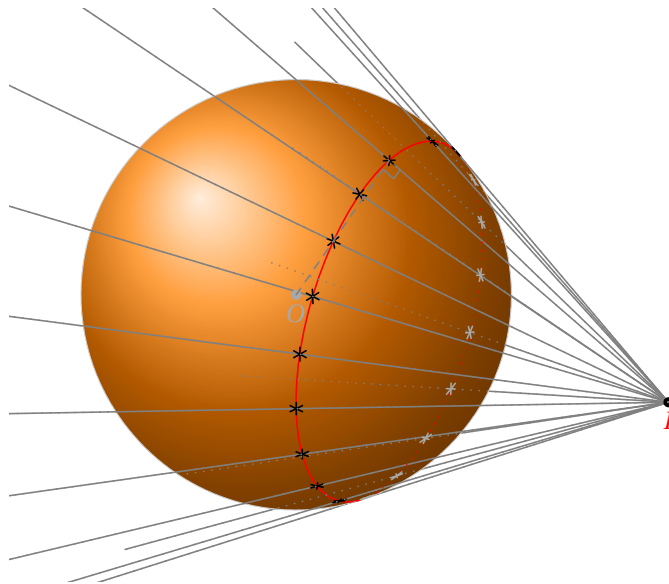
For this spherical tiling, we chose a regular octahedron with a center identical to that of the sphere and with one vertex on the sphere (and therefore all vertices are on the sphere).

### Tangents to the sphere from a point

```
\begin{luadraw}{name=tangent_to_sphere}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, M = pt3d.Origin, pt3d.M

local g = ld.graph3d:new{window={-4,5.5,-4,4},viewdir={30,60},size={10,10}}
require 'luadraw_spherical'
ld.Hiddenlines=true; g:Linewidth(6)
local O, I = Origin, M(0,6,0)
local S,S1 = {0, 3}, {(I+O)/2,pt3d.abs(I-O)/2}
-- the circle of tangency is the intersection between spheres S and S1
local C,r,n = ld.interSS(S,S1)
local L = ld.circle3d(C,r,n)[1] -- list of 3D points on the circle
local dots, lines = {}, {}
-- draw
g:Define_sphere({opacity=1})
g:DScircle({C,n},{color="red"})
for k = 1, math.floor(#L/4) do
    local A = L[4*(k-1)+1]
    table.insert(dots,A)
    table.insert(lines,{I, 2*A-I})
end
g:DSpolyline(lines ,{color="gray"})
g:DStars(dots) -- drawing points on the sphere
g:DSDots({O,I}); -- points in the scene
g:DLabel("$I$",I,{pos="S",node_options="red"},"$O$",O,{})
g:Dspherical()
g:Dseg3d({O,dots[1]},"gray,dashed"); g:Dangle3d(O,dots[1],I,0.2,"gray")
g:Show()
\end{luadraw}
```

Figure 5: Tangents to the sphere from a point



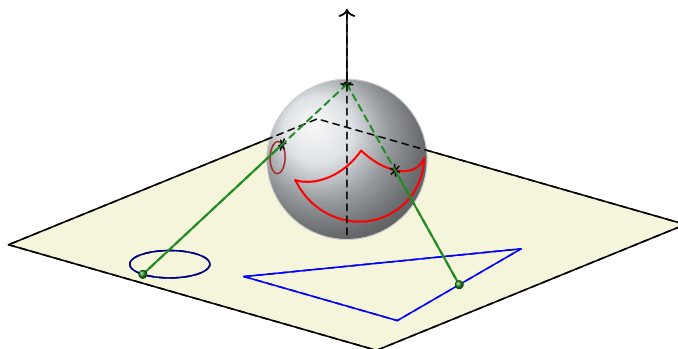
### Inverse Stereography

```

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

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

```

Figure 6: *DSinvstereo\_curve* and *DSinvstereo\_polyline* methods

### 3) The *luadraw\_palettes* Module

The **luadraw\_palettes** Module<sup>1</sup> defines 261 color palettes, each with a name. A palette is a list (table) of colors, which are themselves lists of three numerical values between 0 and 1 (red, green, and blue components). The list of these palettes, as well as their rendering, can be viewed in this [document](#).

**This module does not add any new graphics methods, but it returns a table of color definitions, plus the function `getPal(palname, options)`.** Therefore, it is used as follows (example):

```
local pal = require 'luadraw_palettes'
local color = pal.Blackbody
local BlackbodyTransformed = pal.getPal( -- returns a new palette
  color, -- a palette name
  {
    extract = {2, 5, 8, 9}, -- color numbers to extract
    shift = 1, -- offset among the extracted colors, which gives here: 5,8,9,2
    reverse = true -- reversing the order, which gives here: 2,9,8,5
  }
)
```

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

**Warning:**

- This module adds new graphics methods to the *ld.graph* and *ld.graph3d* classes, as well as several functions in the *luadraw* namespace, but it does not return anything.
- This module has been tested under Linux, but not (yet) under Windows or macOS.
- This module requires that the following programs be installed on your system: *pdflatex*, *pdf2ps*, and *pstoedit*.

The **ld.compile\_tex\_default(options)** function allows you to modify the default settings. These parameters are (with their default values):

- `pdflatexcmd="pdflatex",`
- `pstoeditcmd="pstoedit",`
- `pdf2pscmd="pdf2ps",`
- `preamble="\documentclass[12pt]{article}\n",`
- `usepackage="\usepackage{amsmath,amssymb}\n\usepackage{fourier}\n".`

Depending on your operating system, you may need to modify these variables to add the program path, for example :

```
ld.compile_tex_default( {pstoeditcmd="/usr/bin/pstoedit"} )
```

This module allows you to:

<sup>1</sup>This module is a contribution of [Christphe BAL](#).

1. compile a text fragment in TeX,
2. convert the resulting file into an *eps* file containing *flattened postscript*,
3. read the content of the *eps* file and return its content as a list of paths, with the line thickness at the beginning of each path, and the fill instruction at the end.
4. The list thus obtained can be:
  - (a) drawn on the screen,
  - (b) converted into 3D paths in a given plane and drawn,
  - (c) converted into 3D polygonal lines in a given plane (the thickness and fill command are then lost) and drawn.

### Part One: Compilation and Reading

**Warning** : This step requires compiling the document with the `-shell-escape` or `-enable-write18` option. Without this option, the fragment will not be compiled, which is not a problem if the `<filename>.eps` file already exists and you did not intend to modify it.

The first step is handled by the `ld.compile_tex(text [, filename, conv_stroke])` function. The `<text>` argument is a string; this is the fragment to be compiled. The optional `<filename>` argument is also a string; this is the name of the file that will be created. This name must not contain **a path or an extension**. By default, this name is `"tex2FlatPs"`. It is created in the current directory (but will then be deleted). The optional argument `<conv_stroke>` is a boolean that indicates whether polygonal lines (*stroke* instructions) should be converted into filled strips with a *fill*, or not (`false` by default).

The process unfolds in several steps:

1. Creation of the TeX file. This uses the parameters `preamble` and `usepackage`. Compilation is done with *pdflatex*.
2. The resulting file is converted to PostScript using the *pdf2ps* utility.
3. The resulting PS file is then converted using the *pstoedit* utility into an EPS file in flattened PostScript (all content is in the form of paths).
4. The resulting file `<filename>.eps` is copied to the working directory of *luadraw* (the name of this directory is in the global variable `ld.cachedir`), and all compilation remnants are erased.
5. The contents of the file thus created are automatically read by the function `ld.read_compiled_tex(filename)`, which returns a list of paths. Each path is a list starting with the line thickness, followed by dots and instructions like a regular path, and ending with the fill command (`"fill"`, `"eofill"`, or `"stroke"`).

### Part Two: Using the Result

**In 2D** The result can be drawn using the method `g:Dcompiled_tex(L, anchor, options)` where `<L>` is the result returned by the function `compile_tex()`. The `<anchor>` argument is a complex number; it represents the center of the bounding box of the drawing contained in `<L>`. The `<options>` argument is a table whose fields define the options, which are (with their default value):

- `pos="center"`, indicates the position of the content of `<L>` relative to the anchor point, the possible values are the same as for the `pos` option of labels, that is: `"center"`, `"N"`, `"NE"`, `"E"`, `"SE"`, `"S"`, `"SW"`, `"W"`, `"NW"`.
- `scale=1`, allows you to adjust the size of the drawing, this option can be a number or a table of two numbers: `{scaleX, scaleY}`,
- `color=<current default color>`,
- `dir=nil`, writing direction (`nil` means the usual direction). Generally, it is a table consisting of two vectors `dir={v1, v2}` indicating the writing direction,
- `hollow=false`, enables or disables the filling of shapes. With the value `true`, only the outlines are drawn,
- `drawbox=false`: allows you to draw or not draw the bounding box,

- `draw_options=""`: string containing the options that will be passed directly to the `\draw` command.

The result can also be transformed into a polygonal line using the function `ld.compiled_tex2polyline(L [, scale])` where  $\langle L \rangle$  is the result returned by the `ld.compile_tex()` function. The optional argument  $\langle scale \rangle$  allows you to adjust the size; it can be a number or a table of two numbers:  $\{scaleX, scaleY\}$ .

```
\begin{luadraw}{name=compile_tex2d}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{bbox=false}
require 'luadraw_compile_tex'
local i = cpx.I
local text = "\\[\\int_0^{+\\infty} e^{-\\frac{x^2}{2}}dx = \\frac{\\sqrt{2\\pi}}{2}\\]" -- text to compile
-- compile with -shell-escape the first time to create the file
local L = ld.compile_tex(text,"gauss_integral",true) -- with true the lines (strokes) are changed into strips
g:Shift(2*i) -- a first drawing
g:Dcompiled_tex(L,0,{scale=2,hollow=true, drawbox=true, draw_options="fill=pink", dir={1-i/4,i}}) -- we draw L

g:Shift(-4*i) -- a second drawing
L = ld.compiled_tex2polyline(L,{3,3}) --L is converted to a polygonal line
local f = function(z) return Z(z.re,z.im+math.sin(z.re*1.5)) end -- this function produces sinusoidal waves
L = ld.ftransform(L,f) -- we apply f to L
g:Dpath( ld.polyline2path(L, 'draw=none,fill=blue') -- we draw L as a path
g:Show()
\\end{luadraw}
```

Figure 7: Example with `compile_tex` in 2D

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

**In 3D** The result can be converted to 3D using the method `g:Compiled_tex2path3d(L, options)` where  $\langle L \rangle$  is the result returned by the function `ld.compile_tex()`. The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `scale=1`, allows you to adjust the size of the drawing, this option can be a number or a table of two numbers:  $\{scaleX, scaleY\}$ ,
- `anchor=pt3d.Origin`, 3D point which represents the center of the bounding box of the drawing, `pos="center"`, indicates the position of the content of  $\langle L \rangle$  relative to the anchor point, the possible values are the same as for the `pos` option of labels, that is: "center", "N", "NE", "E", "SE", "S", "SW", "W", "NW",
- `color=<current default color>`,

- `dir={pt3d.vecJ,pt3d.vecK}`, basis of the plane in which the result will be located (this plane will also contain the `anchor` point), these two vectors indicate the direction of writing,
- `polyline=false`, with the value `true` the returned result will be a list of lists of 3D points and can therefore be drawn with the method `g:Dpolyline3d()`, however, the information: line thickness and fill command, are lost. With the value `false` the result is a list of paths, each path is a list starting with the line thickness, followed by 3D points and instructions like an ordinary 3D path, and ending with the fill command (`"fill"`, or `"eofill"` or `"stroke"`).

With the option `polyline=false` (default value), the output can be drawn using the method `g:Dcompiled_tex3d(L, options)` where  $\langle L \rangle$  is the result of the method `g:Compiled_tex2path3d()`. The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- `color=<default current color>`,
- `hollow=false`, enables or disables the filling of shapes. With the value `true`, only the outlines are drawn.
- `drawbox=false`: allows you to draw or not draw the bounding box,
- `draw_options=""`: string containing the options that will be passed directly to the `\draw` command.

```
\begin{luadraw}{name=compile_tex3d}
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Origin, vecI, vecJ, vecK, M, Mc = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M, pt3d.Mc

local g = ld.graph3d:new{ window={-3,3,-4,4}, margin={0,0,0,0}, size={10,10}, viewdir={-50,60}}
require 'luadraw_compile_tex'

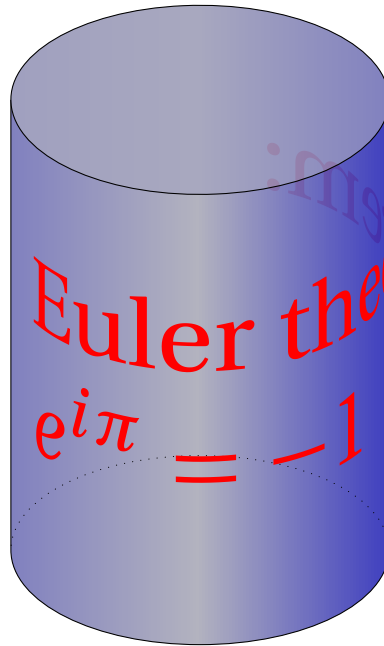
function curve_on_cylinder(curve,cylinder,screenNormal)
-- curve is a 3D polyline on a cylinder
-- cylinder = {A,r,B} estle cylindre
-- this function separate the visible part from the hidden part of the curve
  local A,r,B = table.unpack(cylinder)
  local U = B-A
  local visibility_function = function(N)
    local I = ld.dproj3d(N,{A,U})
    return (pt3d.dot(N-I,screenNormal) >= 0)
  end
  return ld.split_points_by_visibility(curve,visibility_function)
end

local A, r, B = -3*vecK, 2, 2.5*vecK -- the cylinder
local text = "Euler theorem: \par \((e^{i\pi}=-1)\)"
local L = ld.compile_tex(text, "essai") -- compile with shell-escape the first time to create "essai.eps" file
local C = g:Compiled_tex2path3d(L,{scale=3, anchor=M(r,0,0), dir={vecJ,vecK}, polyline=true})
--C is the text converted into 3D polylines, in the plane passing through anchor and basic direction dir, with a scale
  of 3.

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
C = ld.ftransform3d(C,f) -- plane curve -> cylindrical curve transformation
local Cv, Ch = curve_on_cylinder(C, {A,r,B}, g.Normal) -- visible part and hidden part of C, this may take some time
Ch = ld.polyline2path3d(Ch) -- hidden part, conversion to path
g:Dpath3d(Ch, "draw=none,fill=red!30")
g:Dcylinder(A,r,B,{color="blue",opacity=0.5})
Cv = ld.polyline2path3d(Cv) -- visible part, conversion to path
g:Dpath3d(Cv, "draw=none,fill=red")
g:Show()
\end{luadraw}
```



Figure 8: Write on a cylinder



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

This module adds a graphics method to the class *ld.graph3d*, as well as several functions in the namespace *luadraw*, but it does not return anything.

#### Basic Function

The module *luadraw\_cvx\_polyhedra\_nets* allows you to « unfold » a **convex** polyhedron to obtain a net. The function that performs the unfolding is:

#### **ld.unfold\_polyhedron(P, options)**

The argument  $\langle P \rangle$  must be a convex polyhedron. The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- **opening=1**, a value between 0 and 1 representing the "opening rate". With the value 1, the polyhedron is fully unfolded; the facets returned by the function will therefore all be in the same plane. With the value 0, the function returns the polyhedron's faces without modification.
- **root=1**, the number of the polyhedron face that will serve as the root, because the function represents the polyhedron as a tree by determining, for each face, its neighbors (adjacent facets), as well as any shared edges and angles. This option allows you to choose the face that will serve as the starting point.
- **model=nil**, a list of facet number lists to impose a pattern model, for example **model={{1,6},{1,3},{1,4},{1,5,2}}**, the sublist  $\{1,5,2\}$  means that facet 1 is the ancestor of facet 5, and that facet 5 is the ancestor of facet 2, that is to say that facets 5 and 1 are adjacent, and facet 5 will rotate around its common edge with facet 1 (same for 5 and 2). For the model to be consistent, all facets of the polyhedron EXCEPT one (which will be the facet **root**), must have one and only one ancestor; If facets 1 and 5 are not adjacent in the polyhedron, the function stops and displays an error in the terminal. When the **model** option is set to **nil** (the default value), the algorithm calculates a consistent model itself.
- **to2d=false**, is a boolean value that returns a 2D version of the pattern in the screen plane coordinate system. With the value **true**, the **opening** option automatically takes the value 1, and the facets returned by the function will have vertices expressed as complex numbers.
- **tabs=false**, is a boolean value that allows adding or excluding tabs from the pattern in the 2D version. With the value **true**, the **to2d** option automatically takes the value **true** as well. The facets returned by the function will have

vertices expressed as complex numbers in the screen coordinate system, and the function also returns a 2D polygonal line representing tabs for certain edges (these are determined automatically).

- `tabs_wd=0.2`, a numeric value representing the thickness of the tabs when the `tabs` option is set to `true`.
- `tabs_lg=0.5`, a numerical value between 0 and 1, determines the length of the shorter side of the tabs. This length is equal to the length of the edge (which is the longer side) multiplied by `tabs_lg` (when the `tabs` option is set to `true`).
- `rotate=0`, when the `to2d` option is set to `true`, rotates the drawing by an angle equal to `rotate` (in degrees) around its center. In the 3D version, the drawing is rotated by an angle equal to `rotate` (in degrees) around the axis passing through the centroid of the facet `root` and oriented by a normal vector to this facet pointing outwards from the polyhedron.

The function returns a table containing the following fields:

- The field `facets`: which contains the list of facets, with vertices in 2D (complex numbers) if the `to2d` option is `true`, or vertices in 3D (3D points) otherwise.
- The field `tree`: which is a list of the form:

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

Each element of this list represents a facet, with the following information for each facet:

- `ancestor`: the number of the ancestor facet, its position in the list `tree` (the facet that served as the root has the number 0 as its ancestor, which does not correspond to any facet).
- `n1, n2`: the number of the vertices of the ancestor facet representing the common edge.
- `angle`: the angle in degrees with the ancestor facet.
- `vertices`: the list of vertices (3D points) of the facet.
- The `bounds` field: which contains, as a list, the bounding box of the facets (either 2D or 3D)
- When the `tabs` option is set to `true`, there are two additional fields in the result:
  - The `tabs` field: which contains a 2D polygonal line (a list of lists of complex numbers) representing the tabs, only when the `tabs` option is set to `true`.
  - The `twins` field: which contains a list of the form  $\{\{a1, b1\}, \{a2, b2\}\}, \dots$  representing the list of twin edge pairs (twin edges coincide when the polyhedron is closed). `a1, b1, a2, b2` are complex numbers representing the endpoints of the edges in the 2D version of the polyhedron net. This list is calculated only when the `tabs` option is set to `true`.

### The Drawing Method

This is the method `g:Dpolyhedron_net(P, options)` where  $\langle P \rangle$  denotes a convex polyhedron. The options are those of the previous function, plus the following:

- In the case of a 2D pattern (when the `to2d` option, or the `tabs` option, has the value `true`):
  - `facet_name=false`, with the value `true` the facet number (preceded by the letter 'F') will be displayed in the center of each facet.
  - `edge_name=false`, with the value `true`, the edge number (preceded by the letter 'e') will be displayed in the center of each edge, allowing you to identify twin edges and therefore neighboring facets.
  - `tabs_options=""`, string representing TikZ drawing options for the tabs if the `tabs` option has the value `true`.
  - `facet_options=""`, string representing TikZ drawing options for the `g:Dpolyline()` method that will draw the facets.
- In the case of a 3D pattern, there is only the following additional information:
  - `facet_options={}`, a list of drawing options for the `g:Dfacet()` method that will draw the facets.

The drawing is accompanied by a display in the terminal of its 2D bounding box.

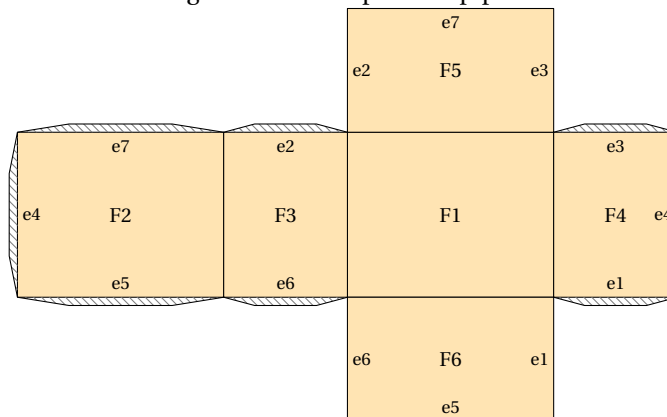
## Examples

In this example, we display the default 2D net of a parallelepiped  $P$  with hatched tabs, the facet numbers (this number is the position in the  $P$ .facets list), and the edge numbers to see which ones need to be glued together:

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

local g = ld.graph3d:new{viewdir={30,60},window={-8.5,8,-5,5},bbox=false, size={10,10}}
require 'luadraw_cvx_polyhedra_nets'
P = ld.parallelepiped(Origin, 4*vecI,5*vecJ,3*vecK)
g:Dpolyhedron_net(P, {tabs=true, tabs_options="pattern=north west lines, pattern color=gray",
  facet_options="fill=Orange!30", facet_name=true, edge_name=true})
g:Show()
\end{luadraw}
```

Figure 9: Net of a parallelepiped

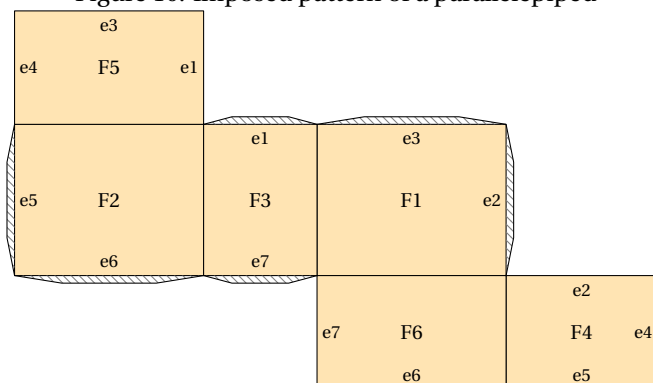


The default pattern here would correspond to the option `model={{1,3},{1,4},{1,5},{1,6},{3,2}}`<sup>2</sup>, but we might want to impose a different model, for example, with the same parallelepiped:

```
\begin{luadraw}{name=parallelep_net2}
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Origin, vecI, vecJ, vecK = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK

local g = ld.graph3d:new{viewdir={30,60},window={-9,9,-5,5},bbox=false,size={10,10}}
require 'luadraw_cvx_polyhedra_nets'
P = ld.parallelepiped(Origin, 4*vecI,5*vecJ,3*vecK)
g:Dpolyhedron_net(P, {model={{4,6,1,3,2,5}},tabs=true,
  tabs_options="pattern=north west lines, pattern color=gray",
  facet_options="fill=Orange!30", facet_name=true,
  edge_name=true, rotate=-90})
g:Show()
\end{luadraw}
```

Figure 10: Imposed pattern of a parallelepiped



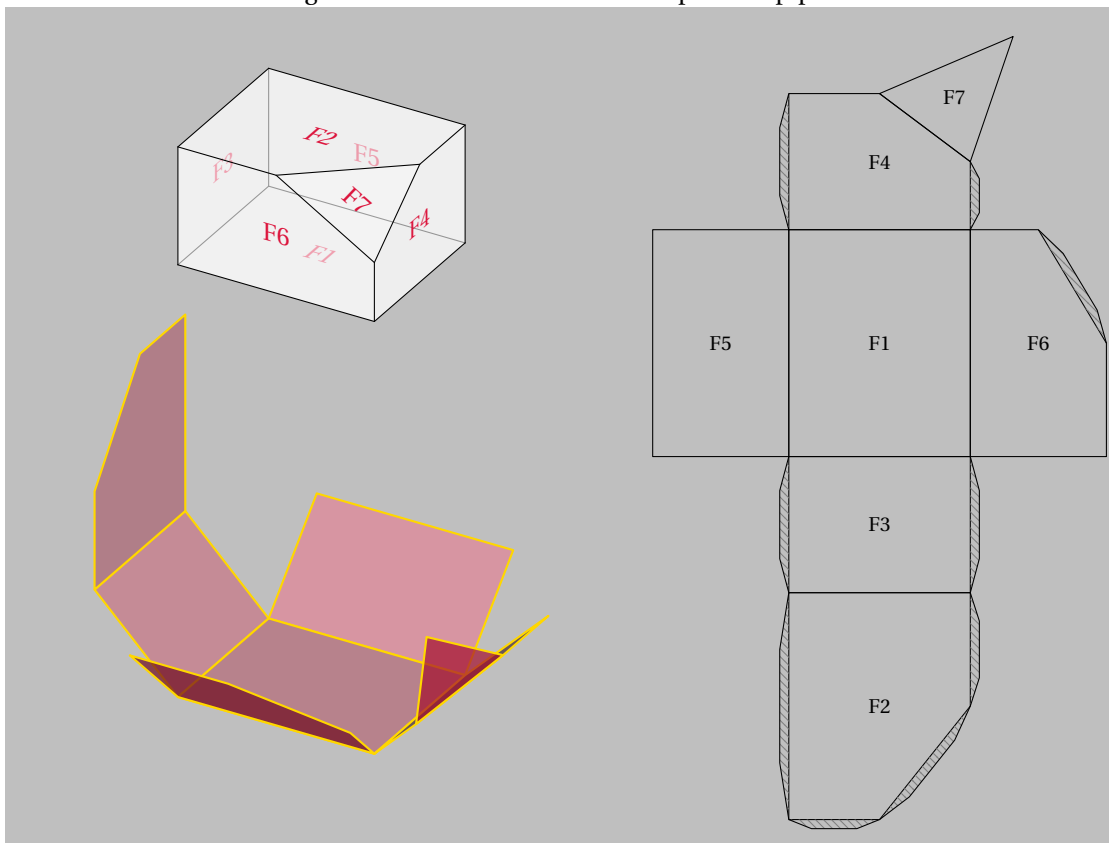
<sup>2</sup>The algorithm takes the first face, then looks for its neighbors, then the neighbors of the first neighbor, etc.

Here is an example with a truncated parallelepiped that is half-unfolded:

```
\begin{luadraw}{name=parallelep_net3}
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

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

Figure 11: Half unfolded truncated parallelepiped



**NB:** The functions `ld.unfold_polyhedron()` and `g:Dpolyhedron_net()` apply to any convex polyhedron, but they will not give the expected result with a non-convex polyhedron.

### The `unfold_tree()` function

It can be useful to retrieve the tree generated by the `ld.unfold_polyhedron()` function to avoid recalculating it multiple times, for example, during an animation. The `ld.unfold_tree(tree, opening [, num])` function also allows you to unfold the polyhedron. The argument `<tree>` is the tree provided by the `ld.unfold_polyhedron()` function, the optional argument `<opening>` is a number between 0 and 1 that represents the opening rate (1 by default), the optional argument `<num>` is the

number of the facet you want to open (and all descendants of that facet will rotate in the same way), when this argument is omitted, all facets rotate.

### Example of animation:

```
\begin{luacode*}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK

ld.nbimages = 70
-- images creation
local g = ld.graph3d:new{ viewdir={"central",30,60}, bg="gray", size={10,10}, margin={0,0,0,0} }
-- declarations
local poly = require 'luadraw_polyhedrons'
require 'luadraw_cvx_polyhedra_nets'
local p = ld.linspace(0,1,36)
local T = ld.linspace(0,360, ld.nbimages+1)
local P = poly.dodecahedron(Origin, -2*vecI)
local net = ld.unfold_polyhedron(P)
local tree = net.tree
-- create the image number k, this function must be global
function ld.makeframe(k) -- do not modify this line
    local r = k
    if k > 36 then r = 72-k end
    local P1 = ld.rotate3d( ld.unfold_tree(tree,p[r]), T[k],{Origin,vecK})
    g:Dfacet(P1, {color="Crimson", edgecolor="Gold", edgewidth=8})
    -- send image number k
    g:Sendtotex() -- send the TikZpicture to TeX
    g:Cleargraph()
end
\end{luacode*}
```

The TeX code (with the *animate* package):

```
\newcommand*\nb{\directlua{tex.print(luadraw.nbimages)}}
\newcommand*\makeframe[1]{\directlua{luadraw.makeframe(#1)}}%

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

The result :

Figure 12: Unfolding a dodecahedron

## 6) The *luadraw\_fields* module

This module does not return anything; it adds new graphics methods to the *ld.graph* class and a new function to the *luadraw* namespace. These are the same methods given at the end of Chapter 1 for drawing vector and gradient fields. It includes:

- The function **ld.field(f, x1, x2, y1, y2 [, grid, length])**, which returns the vector field (a list of line segments). The argument  $\langle f \rangle$  is a function  $\langle f \rangle: (x, y) \mapsto f(x, y) \in \mathbf{R}^2$  ( $f(x, y)$  is a list of two real numbers). The field is computed over the rectangle  $[x_1; x_2] \times [y_1; y_2]$ . The argument  $\langle grid \rangle$ , which defaults to  $\{25, 25\}$ , specifies the number of subdivisions of the intervals  $[x_1; x_2]$  and  $[y_1; y_2]$ . The argument  $\langle length \rangle$  allows one to fix the vector length (all vectors have the same norm); by default, a suitable length is computed from the step sizes along both axes.
- The method **g:Dvectorfield(f, options)** draws the vector field associated with the function  $\langle f \rangle$ . The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):
  - **view=<default window>**: a list of the form **view={x1,x2,y1,y2}** defining the rectangle  $[x_1; x_2] \times [y_1; y_2]$ . By default, this is the window selected when the graph was created.
  - **grid={25,25}**: a list of two integers defining the number of subdivisions along each axis.
  - **length=nil**: allows one to impose the vector length; by default, the length is computed from the step sizes along both axes.
  - **draw\_options=""**: a string containing the drawing options passed to TikZ.
- The method **g:Dgradientfield(f, options)** draws the gradient field associated with the **scalar** function  $\langle f \rangle: (x, y) \mapsto f(x, y) \in \mathbf{R}$ . The argument  $\langle options \rangle$  is the same as in the **Dvectorfield()** method.

An example of both methods has already been given on page ??.

## 7) The *luadraw\_shadedforms* module

This module does not return anything; it adds new graphics methods to the *ld.graph* class. It allows you to draw polygonal lines or fill a shape using a color gradient.

**Dshadedpolyline**

The function **g:Dshadedpolyline(L, palette, options)** enables the rendering of the two-dimensional polygonal line  $\langle L \rangle$  with a continuous color gradient determined by the computational method and the selected  $\langle palette \rangle$ . The argument  $\langle L \rangle$  may be either a list of complex numbers or a list of lists of complex numbers. The  $\langle palette \rangle$  is a list of colors, each color being represented as a list of three real numbers between 0 and 1, corresponding respectively to the red, green, and blue components. The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

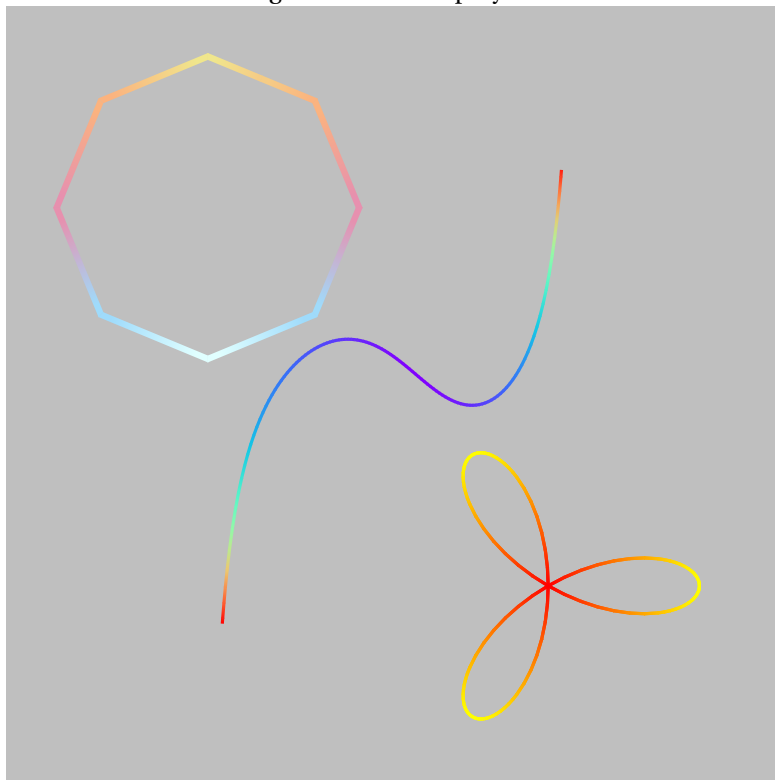
- **values="x"**: for each point in  $\langle L \rangle$ , a numerical value is computed, which determines the associated color according to the chosen palette. The **values** option specifies the evaluation mode and may take one of the following forms:
  - **"x"** (default): the value corresponds to the point's abscissa;
  - **"y"**: the value corresponds to the point's ordinate;
  - a function **f**:  $(x, y) \mapsto f(x, y) \in \mathbf{R}$ : the value for each point  $(x, y)$  in  $\langle L \rangle$  is given by  $f(x, y)$ .
- **width=<current line width>**: specifies the line thickness in tenths of a point.
- **close=false**: a Boolean value indicating whether the polygonal line should be closed.
- **clip=nil** — this option can be either **nil** (default) or a table  $\{x1, x2, y1, y2\}$ . In the former case, the line is clipped by the current two-dimensional window **after** applying the graph's 2D transformation matrix; in the latter, the line is clipped by the window  $[x_1; x_2] \times [y_1; y_2]$  **before** the transformation.

This procedure transforms  $\langle L \rangle$  into a sequence of trapezoids, which are subsequently filled using a smooth color gradient.

```
\begin{luadraw}{name=shading_polyline}
local ld = luadraw
local cpx = ld.cpx
local i, Z = cpx.I, cpx.Z

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

Figure 13: Shaded polyline



### Dcolorbar

The method **g:Dcolorbar(A, pal, options)** allows you to draw a rectangle with a color gradient from a palette, optionally with a gradation. The argument  $\langle A \rangle$  is a complex number; it is the reference point for constructing the rectangle. The argument  $\langle pal \rangle$  is a color palette (a list of lists of the form  $\{r, g, b\}$  where  $r$ ,  $g$ , and  $b$  are between 0 and 1). The  $\langle options \rangle$  argument is a table whose fields define the options, which are (with their default value):

- **minmax**={0,1}, a list containing the minimum value (which will be assigned to the first color of the palette) and the maximum value (which will be assigned to the last color of the palette). Thus, any numeric value between *min* and *max* corresponds to a color in the palette.
- **dir**=cpx.I, direction of the longer side of the rectangle (this vector is automatically normalized), so by default the rectangle is vertical.
- **length**=8, length of the rectangle.
- **width**=0.5, width of the rectangle. The vertices of the rectangle are:

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

- **values**=0, this option allows you to define either the number of numeric values displayed equally within the interval **minmax** (none by default), or the list of numeric values displayed (in this case **values** must be a list of numeric values within the interval **minmax**).
- **addvalues**=nil, this option allows you to define a list of numeric values to display in addition to the *min* and *max* values. This option takes precedence over the previous one.
- **digits**=2, the number of decimal places for the numeric displays.
- **labelpos**="E" option positions the labels relative to anchor points ("N", "NE", "E", "SE", "S", "SW", "W", "NW"). The labels are positioned along the  $(A, \text{dir})$  axis.

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

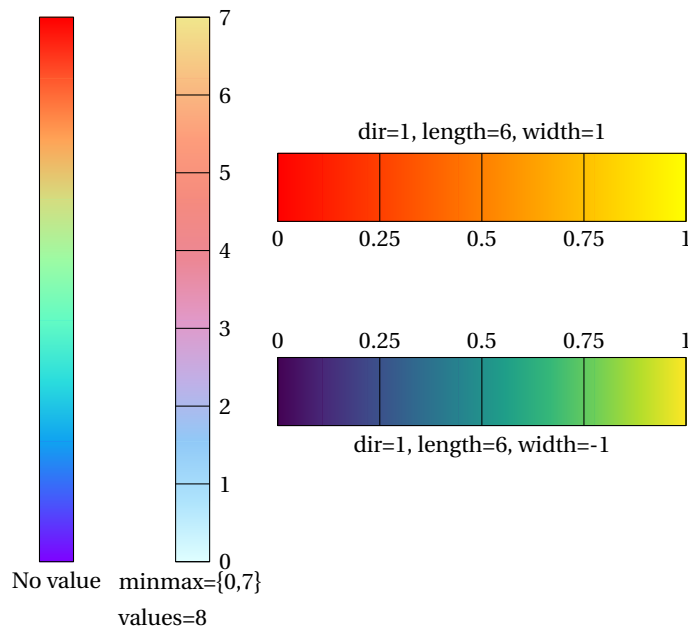


```

local g = ld.graph:new{size={10,10}, bbox=false}
g:Labelsize("small")
require 'luadraw_shadedforms'
local pal = require 'luadraw_palettes'
g:Dcolorbar(Z(-4,-4), pal.Rainbow)
g:Dcolorbar(Z(-2,-4), pal.GasFlame, {minmax={0,7},values=8})
g:Dcolorbar(Z(-1,1), pal.Autumn, {dir=1,length=6,width=1,addvalues={0.25,0.5,0.75},labelpos="S"})
g:Dcolorbar(Z(-1,-1), pal.Viridis, {dir=1,length=6,width=-1,addvalues={0.25,0.5,0.75},labelpos="N"})
g:Dlabel("No value", Z(-4.25,-4),{pos="S"},
  "\\parbox{1.95cm}{minmax=\\{0,7\\}\\}\\values=8}", Z(-2.25,-4), {},
  "dir=1, length=6, width=1", Z(2,2), {pos="N"},
  "dir=1, length=6, width=-1", Z(2,-2),{pos="S"})
g:Show()
\\end{luadraw}

```

Figure 14: Color bars



### Dshadedrectangle

**NB:** Using this method requires the *shadings* library.

The **g:Dshadedrectangle**(*x1*, *x2*, *y1*, *y2*, *pal*, *options*) method fills the rectangle  $[x_1; x_2] \times [y_1; y_2]$  with a gradient of colors extracted from the *pal* palette. Each point  $(x, y)$  in the rectangle has a color from the palette, calculated from a value  $f(x, y)$  where  $f$  is a numeric function defined on the rectangle. The *options* argument is a table whose fields define the options, which are (with their default value):

- **values=function(x,y) return cpx.abs(Z(x,y)) end**, this option defines the function  $f$  used to calculate the color of each point. The maximum value of  $f$  on the rectangle will correspond to the last color in the palette, and the minimum value of  $f$  will correspond to the first color in the palette. By default, this function is the modulus.
- **grid={15,15}**, this option defines the number of subdivisions for the interval  $[x_1; x_2]$  and for the interval  $[y_1; y_2]$ . The finer the subdivision, the longer the display will take.
- **bar="none"** allows you to add or not add a legend using the *Dcolorbar* method. This option can be: "none", "right", "bottom", "left", or "top".
- **bardist=1**, the distance between the rectangle and the legend, if there is one.
- **baroptions={}**, a list of options for the **Dcolorbar()** method if there is a legend.
- **out=nil**, if a list variable is assigned to this parameter *out*, then the method adds the two values *min* and *max* of the  $f$  function to this list (which allows you to retrieve these two values if necessary).

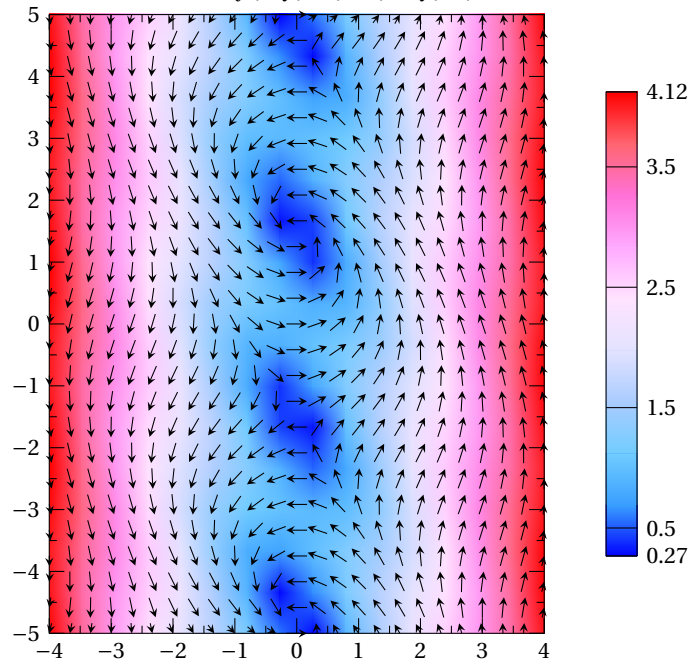
```

\begin{luadraw}{name=Dshadedrectangle}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

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

```

Figure 15: Shaded rectangle  
vector field  $f(x,y) = (\cos(x+y), x)$



### Dshadedregion

The method **g:Dshadedregion(*apath*, *pal*, *options*)** fills the region defined by the path (*apath*) with a gradient of colors extracted from the palette (*pal*). This method uses the previous one (**Dshadedrectangle()**) on the rectangle defined by the bounding box of the path. Each point  $(x,y)$  in this rectangle has a color from the palette, calculated from a value  $f(x,y)$  where  $f$  is a numeric function defined on this rectangle. The drawing is clipped to the path. The argument (*options*) is the same as that of the previous method **Dshadedrectangle()**.

```

\begin{luadraw}{name=Dshadedregion}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{window={-4.5,6,-4.5,5}, size={10,10},bg="lightgray", bbox=false}
g:Labelsize("small")
require 'luadraw_shadedforms'
local pal = require 'luadraw_palettes'
local L1, L2 = -2, 2
local f = function(x,y)
  local z = Z(x,y)

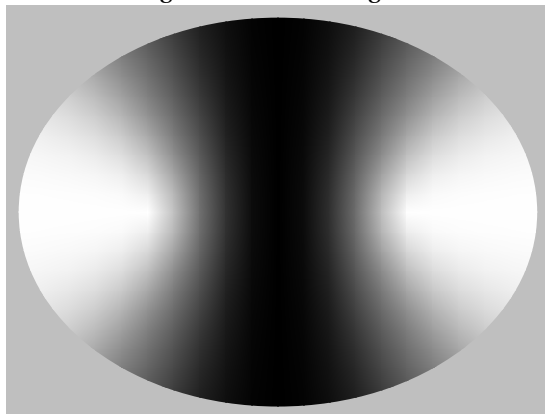
```

```

    return (cpx.abs(z-L1)-cpx.abs(z-L2))^2
end
g:Dshadedregion({4,0,4,3,"e"}, pal.getPal(pal.Grays,{reverse=true}), {values=f, grid={20,20}})
g:Show()
\end{luadraw}

```

Figure 16: Shaded region



## 8) The *luadraw\_povray* Module

This module does not return anything; it adds new graphics methods to the *ld.graph3d* class, as well as a function in the namespace *luadraw*.

### Prerequisites and introduction

First and foremost, the POV-Ray software must be installed on your computer in its command-line version (non-graphical version).

- On Linux: POV-Ray is always a command-line application and is installed using the package manager.
- On macOS: the command-line version of POV-Ray can be installed via [HomeBrew](#).
- On Windows: the installer can be downloaded from the [POV-Ray](#) website.

This module enables the creation of (relatively simple) source files for the POV-Ray software (command-line version) and allows them to be compiled on the fly. Once created, the resulting image (in *png* format) can be automatically included in the current graphic, allowing further drawing both underneath and on top of it. The image is inserted with perfect alignment, meaning that the 3D reference frames of the graphic and of the image will match exactly, provided that the system operates in **orthographic projection** mode. Source files and generated images are created in the working directory of *luadraw*. The source files can of course be compiled separately by the user; the parameters to be passed to POV-Ray are written in comments at the beginning of the source file. The source file is divided into three parts: a preamble, the object declarations, and the rendering of those objects.

### Default Settings

Using this module requires certain default settings that may vary depending on the operating system. These default values can be modified using the function:

#### **ld.povray\_default(options)**

The *<options>* argument is a table whose fields define the options, which are (with their default value):

- **bg=""** This option defines the background. It can be an empty string (default value), in which case the background will be transparent (essential if drawing under the image), or a non-empty string, in which case it must be a color name known to POV-Ray, or an array of the form  $\{r, g, b\}$  representing a color with the values  $r$ ,  $g$ , and  $b$  between 0 and 1.
- **shadow=true**, this boolean indicates whether objects should cast a shadow or not (this boolean can be modified locally for each object created).

- `imagescale=1`, this option allows you to change the size of the generated *png* image; values greater than 1 obviously increase its file size.
- `param="-V +A +FN"`, these are the basic parameters passed to POV-Ray. The image width and height will be automatically added to these, as well as the "+UA" option if the background should be transparent.
- `pov_cmd=`, the name of the command to run POV-Ray. On Unix, the default value is "`povray`", and on Windows, the default value is "`pvengine64.exe`".
- `pov_cmd_ext=""`, a string that is appended to the end of each command sent to POV-Ray.
- `win_param_ext="/RENDER /EXIT"`, additional parameters that only apply to Windows.
- `include={}`, this option allows POV-Ray to include *.inc* files, for example:

```
include={"textures.inc", "colors.inc"}
```

These two files are normally distributed with POV-Ray; the first defines textures, the second colors. By default, no inclusions are made.

If you modify one of these options, the value you assign becomes the new default value for the rest of the document. However, it's not necessary to repeat this function in every graph in your document using POV-Ray; it only needs to be done once in the preamble. For example, on macOS, if you installed POV-Ray via *HomeBrew*, then this is suitable:

```
\documentclass{standalone}%
\usepackage[svgnames]{xcolor}
\usepackage[3d]{luadraw}
\directlua{%
require "luadraw_povray"
luadraw.povray_default( {pov_cmd = "/opt/homebrew/bin/povray"} )
}%
...
```

Of course, if the default values are suitable for your system (this should be the case under Linux), you do not need to use the function `ld.povray_default()`.

### Before Creating Objects

The POV-Ray must be initialized using the method:

**`g:Pov_new(options)`.**

The argument *(options)* is exactly the same as for the previous function `ld.povray_default()`, except that the effect will be local to the current graph only.

### Creation of Objects

Objects are created using methods following the format **`g:Pov_<command>(<data>, options)`**. Each object is first declared in the POV-Ray source file with a name (similar to a variable), and then this object is "rendered" in the third section of the source file with a texture defined from the options.

The argument *(options)* is a table whose fields specify the options of the object. The following are the **options common to all objects**, along with their default values:

- `name="object<num>"`, a string representing the name of the created object. By default, it is the word "object" followed by a number (its order of appearance). Assigning a name is useful when the object needs to be reused later.
- `shadow=true`, a Boolean indicating whether the object casts a shadow.
- `render=true`, a Boolean indicating whether the object should be displayed. It may be desirable not to render an object if it is only used in the construction of another one.
- Options defining the base texture, composed of a *pigment* (color + opacity) and a *finish* (ambient + diffuse + phong):

- `color=ld.White`, this option may be either a table in the form  $\{r, g, b\}$  (with  $r$ ,  $g$ , and  $b$  between 0 and 1), or a string that must represent a color known to POV-Ray. It can also refer to a texture defined only by a *pigment* (for example, `color="Blue_Sky3"`, where this name is defined in the file *textures.inc*).
- `usepalette=nil`, this option allows the use of a color palette. The syntax is `usepalette={palette, func, minmax}` where  $\langle palette \rangle$  is the list (table) of colors, each of which is a list in the format  $\{r, g, b\}$  with  $r$ ,  $g$ , and  $b$  between 0 and 1. The argument  $\langle func \rangle$  is a string that defines the gradient type; it can be either "x", "y", or "z" (in this case, the argument  $\langle minmax \rangle$  must be a list containing the minimum and maximum values), or a string of the form "function <expression depending on x, y, z> ". The expression that depends on  $x$ ,  $y$ , and  $z$  is a function (for POV-Ray) that must return a value between 0 and 1 (in this case, the  $\langle minmax \rangle$  argument is not needed).
- `opacity=1`, a real number between 0 and 1 defining the object's opacity.
- `ambient=0.35, diffuse=0.8, and phong=0.5`: three parameters defining the *finish*.
- `mytexture=nil`, this parameter allows specifying a texture directly as a string or referencing an existing POV-Ray texture. In that case, the previous parameters are ignored. For example: `mytexture="texture{Silver_Metal}"` (this texture is declared in the file *textures.inc*).
- `matrix=nil`, a 3D transformation matrix applied locally to the object. The global 3D transformation matrix of the graphic is also taken into account.
- `clipbox=nil`, this option defines a list (table) of objects used to clip the object being constructed. These objects may be: an existing object, referenced by its name as a string; a box, specified as a table of the form  $\{M(xmin, ymin, zmin), M(xmax, ymax, zmax)\}$  (representing the box diagonal); or a sphere, given as a table  $\{center, radius\}$ , where *center* is a 3D point and *radius* a positive number.
- `clipplane=nil`, this option defines a list (table) of planes to clip the object being constructed. Each plane must be of the form  $\{A, n\}$ , where  $A$  is a point on the plane (a 3D point) and  $n$  the normal vector to the plane. Only the part of the object lying in the half-space containing  $n$  is preserved.

### List of Predefined Objects

The following is the list of objects that can be drawn using the corresponding methods:

- **Implicit surfaces** defined by the equation  $f(x, y, z) = 0$ . The corresponding method is:

**g:Pov\_implicit( povfunction, luafunction, options).**

The argument  $\langle povfunction \rangle$  is a string containing the expression of  $f(x, y, z)$ . POV-Ray supports standard mathematical functions; however, note that the power function is written as *pow*, i.e.  $x \mapsto \text{pow}(x, n)$ , and that POV-Ray does not handle numerical errors such as division by zero.

The options are those already described, plus the specific option `containedby=<current 3D window>`, which specifies the box (or sphere) within which the computations are carried out. By default, this box is the current 3D window of the graphic. A box is given as a table of the form  $\{M(xinf, yinf, zinf), M(xsup, ysup, zsup)\}$  (representing a diagonal of the box), and a sphere is a table of the form  $\{C, r\}$ , where  $C$  is the center (a 3D point) and  $r$  the radius. Example:

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

- **Parameterized Surfaces.** Two possible syntaxes:

1. The method: **g:Pov\_surface(f, u1, u2, v1, v2, options)**

draws the (smoothed) surface parameterized by the function  $\langle f \rangle: (u, v) \mapsto f(u, v) \in \mathbf{R}^3$ . The interval for the parameter  $u$  is given by  $\langle u1 \rangle$  and  $\langle u2 \rangle$ . The interval for the parameter  $v$  is given by  $\langle v1 \rangle$  and  $\langle v2 \rangle$ . The options are those already provided, plus two specific options:

- `grid={25,25}`, this option defines the number of points to calculate for the parameter  $u$  followed by the number of points to calculate for the parameter  $v$  (25 by default).
- `clip=false`, with the value `true` the surface is clipped to the current 3D window.

Example:

```
local f = function(x,y) return M(x,y,x^2+y^2) end
g:Pov_surface(f,-2,2,-2,2, {color=ld.SteelBlue, clip=true})
```

## 2. The method: `g:Pov_surface(xfunc, yfunc, zfunc, u1, u2, v1, v2, options)`

draws the surface parameterized by  $(u, v) \rightarrow (x(u, v), y(u, v), z(u, v))$ . The arguments  $\langle xfunc \rangle$ ,  $\langle yfunc \rangle$ , and  $\langle zfunc \rangle$  are three strings containing the expressions  $x(u, v)$ ,  $y(u, v)$ , and  $z(u, v)$ , respectively. The arguments  $\langle u1 \rangle$  and  $\langle u2 \rangle$ , respectively  $\langle v1 \rangle$  and  $\langle v2 \rangle$ , define the interval bounds for the parameter  $u$  and for the parameter  $v$ , respectively. The options are those already given, plus two specific options:

- `containedby=<current 3D window>`, this indicates in which box (or sphere) the calculations will be performed; by default, this box is the 3D window of the current graph. A box is a table of the form  $\{M(xinf,yinf,zinf), M(xsup,ysup,zsup)\}$  (this represents a diagonal of the box), a sphere is a table of the form  $\{C, r\}$ , where  $C$  is the center (3D point) and  $r$  is the radius.
- `max_grad=nil`, numerical value (optional) allowing calculations to be optimized, here is what the POV-Ray help says about this number:

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.

Example:

```
g:Pov_surface("x", "y", "x*x+y*y",-2,2,-2,2, {color=ld.SteelBlue, max_grad=4})
```

Note: This is the same surface in both examples, but the second method is longer, and if the `max_grad=4` option is not specified, then the entire surface is not drawn.

- **Polyhedron or list of facets.** This is the method:

### `g:Pov_facet(F, options).`

The argument  $\langle F \rangle$  is either a polyhedron or a list of facets. The options include the common ones, plus the following specific options (with their default values):

- `edge=false`, boolean indicating whether edges should be drawn.
- `edgestyle=<current line style>`, style of the edges.
- `edgecolor=ld.Black`, color of the edges.
- `edgewidth=<current line width>`, width of the edges.
- `hidden=false`, boolean indicating whether hidden edges should be drawn.
- `hiddenstyle=ld.Hiddenlinestyle`, style of hidden edges.

Example:

```
local T1 = tetra(M(-1,-1,-1), 3*vecI, 3*vecJ, M(1,1,3))
g:Pov_facet(T1, {color=ld.SteelBlue, edge=true, hidden=true})
```

Note: drawing hidden edges is not always optimal; sometimes it is better to draw them with TikZ over the rendered image.

- **Polygonal line.** This is the method:

### `g:Pov_polyline(L, options).`

The argument  $\langle L \rangle$  is either a list of 3D points or a list of lists of 3D points. The options include the common ones, plus the following specific options (with their default values):

- `style=<current line style>`, line style.
- `width=<current line width>`, line width.
- `close=false`, boolean indicating whether the line should be closed.
- `arrows=0`, three possible values: 0 (no arrow), 1 (arrow at the end), or 2 (arrows at both start and end).
- `arrowscale=1`, scale factor for arrow size.
- `hidden=false`, boolean indicating whether hidden parts should be drawn.
- `hiddenstyle=ld.Hiddenlinestyle`, style of hidden parts.

Example: drawing the axes

```
g:Pov_polyline({{-5*vecI,5*vecI},{-5*vecJ,5*vecJ},{-5*vecK,5*vecK}}, {arrows=1,width=8})
```

- **3D points.** This is the method:

**g:Pov\_dots(L, options).**

The argument  $\langle L \rangle$  is a list of 3D points. The options include the common ones, plus the following specific options (with their default values):

- `style="ball"`, two possible styles: "ball" (sphere) or "box" (a box with faces parallel to the 3D window).
- `dotscale=1`, scale factor for dot size.

- **A plane.** This is the method:

**g:Pov\_plane(P, options).**

The argument  $\langle P \rangle$  is a table of the form  $\{A, n\}$ , where  $A$  is a point on the plane  $\langle P \rangle$  (a 3D point) and  $n$  is a vector normal to the plane. The plane is automatically clipped by the 3D window. The options are the options of **g:Pov\_facet()**, plus the specific option `scale=1`.

- **A circle.** This is the method:

**g:Pov\_circle(A, R, N, options).**

It draws the circle with center  $\langle A \rangle$ , radius  $\langle R \rangle$ , and normal vector  $\langle N \rangle$  specifying the plane of the circle. These are the common options.

- **Axes.** This is the method:

**g:Pov\_axes(O, options).**

It draws the axes using point  $\langle O \rangle$  (3D point) as the intersection point. The options are the same as for **g:Pov\_polyline()**. There are no graduations or legend.

- **Basic solids.** These are the methods:

- **g:Pov\_sphere(center, radius, options)**, which draws a sphere with center  $\langle center \rangle$  (3D point) and radius  $\langle radius \rangle$ . These are the common options.
- **g:Pov\_torus(center, R, r, N, options)**, which draws a torus centered at  $\langle center \rangle$  (3D point), with major radius  $\langle R \rangle$ , minor radius  $\langle r \rangle$ , lying in the plane normal to vector  $\langle N \rangle$ . These are the common options.
- **g:Pov\_cylinder(A, R, B, options)**, which draws a cylinder along the axis  $\langle AB \rangle$  from  $\langle A \rangle$  to  $\langle B \rangle$  (3D points), with radius  $\langle R \rangle$ . These are the common options plus the specific option `hollow=false`, indicating whether the cylinder is hollow or not.
- **g:Pov\_cone(A, R, B, [r, options])**, which draws a cone along the axis  $\langle AB \rangle$  from  $\langle A \rangle$  to  $\langle B \rangle$  (3D points), with radius  $\langle R \rangle$  at end  $\langle A \rangle$  and radius  $\langle r \rangle$  at end  $\langle B \rangle$ . The radius  $\langle r \rangle$  is optional and defaults to 0; in that case,  $\langle B \rangle$  is the apex of the cone. These are the common options plus the specific option `hollow=false`, indicating whether the cone is hollow or not.
- **g:Pov\_box(A, B, options)**, which draws a box whose faces are parallel to those of the 3D window of the graphic. The 3D points  $\langle A \rangle$  and  $\langle B \rangle$  define a diagonal of the box, more precisely  $\langle A \rangle = M(xinf, yinf, zinf)$  and  $\langle B \rangle = M(xsup, ysup, zsup)$ . These are the common options.



- **CSG geometry.** These are the following methods (for each of them, the options are the common options):
  - **g:Pov\_union(list, options)**, where  $\langle list \rangle$  is a list of POV-Ray objects; these objects may be either the name of an already created object or a POV-Ray command given as a string. The result is treated as a single object.
  - **g:Pov\_intersection(list, options)**, where  $\langle list \rangle$  is a list of POV-Ray objects; these objects may be either the name of an already created object or a POV-Ray command given as a string. The result is the common part of these objects.
  - **g:Pov\_merge(list, options)**, where  $\langle list \rangle$  is a list of POV-Ray objects; these objects may be either the name of an already created object or a POV-Ray command given as a string. This command works like union, but removes internal surfaces (unlike union), which is useful in the case of transparency.
  - **g:Pov\_difference(list, options)**, where  $\langle list \rangle$  is a list of **two** POV-Ray objects; these objects may be either the name of an already created object or a POV-Ray command given as a string. The result is the difference: object1 minus object2.
- **Writing directly to the source**, with the following methods:
  - **g:Pov\_comment(comment)**, which writes the string  $\langle comment \rangle$  to the source as a comment.
  - **g:Pov\_special(code)**, which writes the string  $\langle code \rangle$  verbatim to the source; it must therefore be valid POV-Ray code. Note that a 3D point written as  $M(x, y, z)$  in *luadraw* is written in POV-Ray code as  $<-x, y, z>$ ; the POV-Ray coordinate system is thus not our usual one and is left-handed.

### Save, execution, inclusion

- **Save and execution.** Once all objects have been created, saving and executing the file with POV-Ray is done using the method:

**g:Pov\_exec([filename]).**

The argument  $\langle filename \rangle$  must be a filename without path or extension, but it is optional; by default, the name of the current graphic is used. The created file will have the *pov* extension and will be saved in the *luadraw* working directory (this directory is stored in the `cachedir` variable). The command used for execution is displayed in the terminal, along with the output from the POV-Ray software, allowing you to see if it encountered an error. The image construction appears on screen, but the window closes as soon as rendering is complete<sup>3</sup>. The image has the same name as the source, except the extension which is *png* instead of *pov*.

**NB:** POV-Ray execution requires compiling the document with the *-shell-escape* or *-enable-write18* option. Once the image is obtained, this option is no longer necessary unless the image has been modified.

- **Save only.** This is done with the method:

**g:Pov\_save([filename]).**

With the same remarks as previously for the  $\langle filename \rangle$  argument.

- **Image inclusion.** Once obtained, the image can be included in the graphic with the method:

**g:Pov\_show([filename]).**

If the  $\langle filename \rangle$  argument is not specified, it refers to the name of the current graphic; otherwise,  $\langle filename \rangle$  must be a complete image filename (with extension). The inclusion is done with `\includegraphics[] {filename}`.

### Examples

#### Intersection of two implicit surfaces

```
\begin{luadraw}{name=intersection_surf}
local ld = luadraw
local cpx, pt3d = ld.cpx, ld.pt3d
local Z, M = cpx.Z, pt3d.M
```

<sup>3</sup>It depends on your operating system.

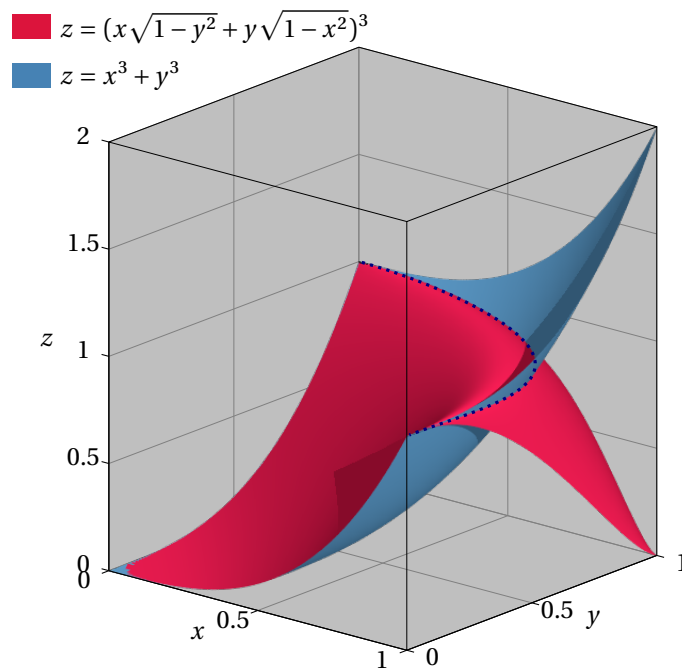


```

local g = ld.graph3d:new{window3d={0,1,0,1,0,2}, window={-0.25,1.5,-0.5,2.25}, size={10,10,0}, viewdir={-50,60}}
local sqrt = math.sqrt
require "luadraw_povray"
local f = function(x,y,z) return x^3+y^3-z end
local h = function(x,y,z) return (x*sqrt(1-y^2)+y*sqrt(1-x^2))^3-z end
local L = ld.implicit(function(x,y) return f(x,y,0)-h(x,y,0) end,0.01,1,0.01,1,{25,25})
L = ld.map(function(z) return M(z.re,z.im,z.re^3+z.im^3) end, L[1]) -- intersection curve
-- using povray
g:Pov_new()
g:Pov_implicit("pow(x,3)+pow(y,3)-z", f, {color=ld.SteelBlue})
g:Pov_implicit("z-pow(x*sqrt(1-y*y)+y*sqrt(1-x*x),3)", h, {color=ld.Crimson, containedby={M(0,0,0),M(0.999,0.999,2)}})
g:Pov_exec()
-- drawing
g:Dboxaxes3d({grid=true, gridcolor="gray",fillcolor="lightgray",xyzstep=0.5, drawbox=true})
g:Pov_show() --Pov-Ray image
g:Dpolyline3d(L, "line width=1.2pt,dotted,Navy") --we draw the curve over the image
local d = 0.1
g:Dsquare(Z(-0.25,2.25), Z(-0.25,2.25-d),1,"draw=none,fill=Crimson")
g:Dsquare(Z(-0.25,2.25-2*d), Z(-0.25,2.25-3*d),1,"draw=none,fill=SteelBlue")
g:Dlabel("$z=(x\\sqrt{1-y^2}+y\\sqrt{1-x^2})^3$", Z(-0.25+d,2.25-d/2) , {pos="E"},
        "$z=x^3+y^3$", Z(-0.25+d,2.25-5*d/2),{f})
g:Show()
\\end{luadraw}

```

Figure 17: Intersection of two implicit surfaces



### Villarceau circles

```

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

local g = ld.graph3d:new{window={-6.5,6.5,-5,5},margin={0,0,0,0}, size={10,10}, viewdir={20,65}}
require "luadraw_povray"
local R, r = 3, 1
local N = ld.rotate3d(vecK, math.asin(r/R)*ld.rad, {Origin, vecJ})
local P = {Origin, -N}
-- using povray
g:Pov_new({bg=ld.LightGray})

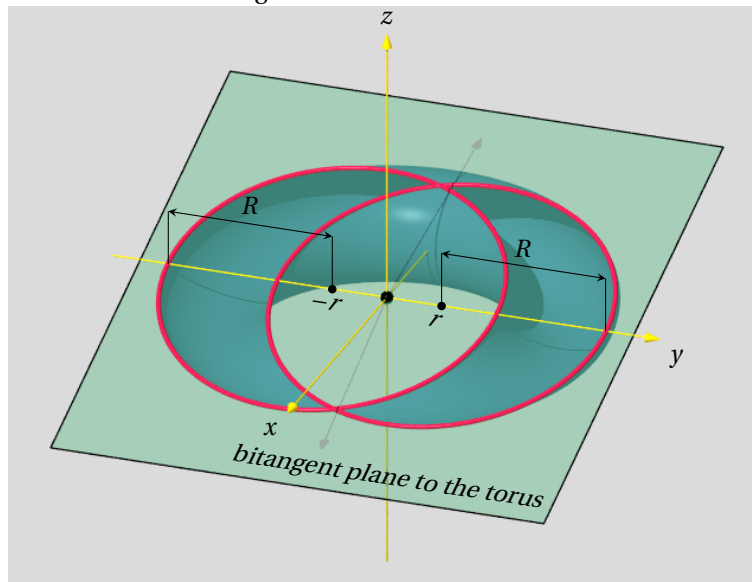
```

```

g:Pov_torus(Origin, R, r, vecK, {color=ld.SteelBlue, clipplane=P})
g:Pov_plane(P, {color=ld.SeaGreen, opacity=0.4, edge=true, scale=0.9})
g:Pov_axes(Origin, {color=ld.Gold, arrows=1})
g:Pov_dots(Origin, {dotscale=1.5})
g:Pov_circle( M(0,r,0), R, N, {color=ld.Crimson, width=12})
g:Pov_circle( M(0,-r,0), R, N, {color=ld.Crimson, width=12})
g:Pov_exec()
-- drawing
g:Pov_show()
local F = g:Plane2facet(P, 0.9)
g:Dpolyline3d({{-r*vecJ, -r*vecJ+vecK}, {(R+r)*vecJ, -(R+r)*vecJ+vecK}})
g:Dpolyline3d({{r*vecJ, r*vecJ+vecK}, {(R+r)*vecJ, (R+r)*vecJ+vecK}})
g:Dpolyline3d({{-r*vecJ+vecK, -(R+r)*vecJ+vecK}, {r*vecJ+vecK, (R+r)*vecJ+vecK}}, "stealth-stealth")
g:Ddots3d({-r*vecJ, r*vecJ})
g:Dlabel3d("$x$", 5*vecI, {pos="SW"}, "$y$", 5*vecJ, {pos="SE"}, "$z$", 5*vecK, {pos="N"},
"$R$", -(R/2+r)*vecJ+vecK, {}, "$R$", (R/2+r)*vecJ+vecK, {},
"bitangent plane to the torus", F[1], {pos="NW", dir={vecJ, pt3d.prod(N, vecJ)}},
"$-r$", -r*vecJ, {pos="S"}, "$r$", r*vecJ, {} )
g:Show()
\end{luadraw}

```

Figure 18: Villarceau circles



### Holes in an half sphere

```

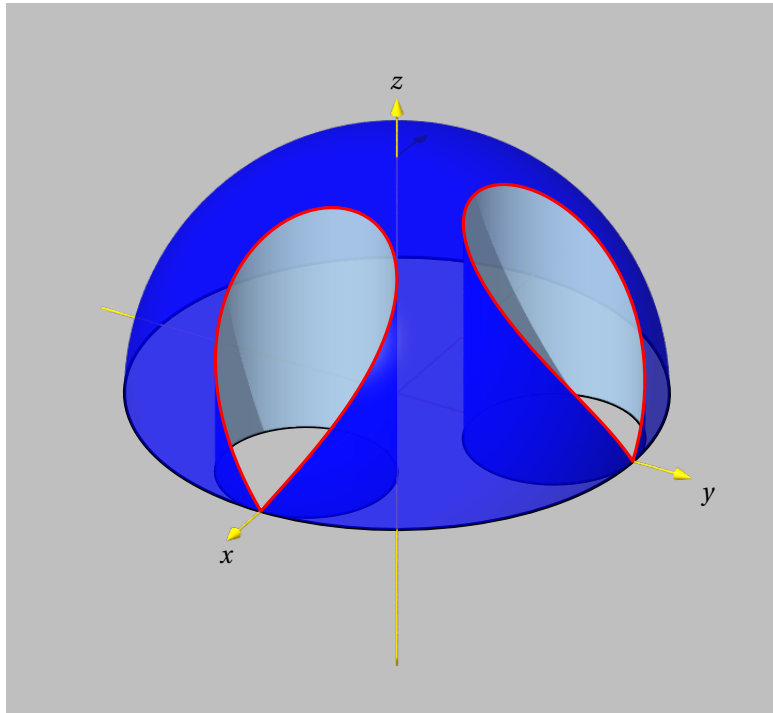
\begin{luadraw}{name=holes_in_hemisphere}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{window={-5,5,-4,5}, size={10,10}, bg="lightgray"}
require "luadraw_povray"
local O, R = Origin, 4
local A, r = 2*R/3*vecJ, R/3
local P = {Origin, vecK}
local base = ld.circle3d(A,r,vecK)[1] --circular base of the cylinder (list of 3D points)
local Cylborder = {}
for _, C in ipairs(base) do -- we project each point C of the base onto the half-sphere
    table.insert(Cylborder, C + math.sqrt(R^2-pt3d.abs2(C))*vecK)
end -- Cylborder is now the intersection between the half-sphere and the cylinder
-- using povray
g:Pov_new()
g:Pov_sphere(Origin, R, {name="sph", clipplane=P, render=false}) -- only declaration
g:Pov_cylinder(A-vecK, r, A+R*vecK, {name="cyl1", render=false})
g:Pov_union({"cyl1", "cyl1 rotate 90*z"}, {name="cyl1", render=false})
g:Pov_cylinder(A-vecK, r-0.001, A+R*vecK, {name="cyl2", render=false})

```

```
-- renderings
g:Pov_difference({"sph","cyl2"}, {color=ld.Blue, opacity=0.7})
g:Pov_union( {"cyl2", "cyl2 rotate 90*z"}, {color=ld.LightBlue,opacity=0.8,clipbox="sph"})
g:Pov_axes(Origin,{color=ld.Gold,arrows=1})
g:Pov_exec()
-- drawing
g:Dcircle3d(0,R,vecK,"line width=1.2pt"); g:Dcircle3d(A,r,vecK,"line width=1.2pt")
g:Dcircle3d(ld.rotate3d(A,-90,{Origin,vecK}), r,vecK,"line width=1.2pt")
g:Pov_show()
g:Dpolyline3d( {Cylborder, ld.rotate3d(Cylborder,-90,{Origin,vecK})}, "red, line width=1.2pt")
g:Dlabel3d("$x$",5*vecI,{pos="S"}, "$y$", 5*vecJ, {pos="SE"}, "$z$", 5*vecK, {pos="N"})
g:Show()
\end{luadraw}
```

Figure 19: Holes in a hemisphere



## 9) The *luadraw\_log\_axes* module

This module does not return anything; it adds new graphing methods to the *ld.graph* class, as well as a function in the namespace *luadraw*.

The *luadraw\_log\_axes* module allows you to display a semi-logarithmic or logarithmic grid and to place points on this grid or draw polygonal lines. Creating this grid modifies the 2D matrix of the graph; therefore, the drawing process takes place in three steps:

1. Initializing the grid with the **`g:Beginlogview()`** method.
2. Drawing with the **`g:Dlogplyline()`** and **`g:Dlogdots()`** methods.
3. Ending the grid with the **`g:Endlogview()`** method, which restores the original matrix and allows you to optionally add to the drawing in "normal" coordinates in the original window.

### Initialization: **`g:Beginlogview()`**

The grid initialization and display are handled by the method:

**`g:Beginlogview(type, xmin, xmax, ymin, ymax, options).`**

- The argument  $\langle type \rangle$  specifies the desired grid type. Possible values are the strings: `"logx"`, `"logy"`, or `"logxy"`.
- The four arguments  $\langle xmin \rangle$ ,  $\langle xmax \rangle$ ,  $\langle ymin \rangle$ , and  $\langle ymax \rangle$  define the value ranges on the two axes. These ranges are completely independent of the graph's 2D window.

- The argument `<options>` is an array whose fields define the options. These are (with their default values):
  - `viewport=<current 2D window>`. This option is an array of the form  $\{x_1, x_2, y_1, y_2\}$ , indicating which area of the graph's 2D window should be used to draw the grid. By default, it's the entire current window (the one defined by the `window` option when the graph was created).
  - `clip=true`. Boolean indicating whether the grid should clip the drawings. Clipping is disabled after the execution of the `g:Endlogview()` method.
  - `grid={true,true}`. This option allows you to show or hide the vertical grid lines (corresponding to the  $x$  axis) and the horizontal grid lines (corresponding to the  $y$  axis).
  - `unit={"", ""}`. This option specifies the increments for the graduations on the non-logarithmic axes. The default value is the step size (`xstep` on  $Ox$ , or `ystep` on  $Oy$ ), EXCEPT when the `labeltext` option is not an empty string, in which case `unit` takes the value 1.
  - `nbsubdiv={0 or 3, 0 or 3}`. This option specifies the number of Subdivisions between two main graduations on the axis. When the axis is non-logarithmic, the default value is zero; otherwise, the default value is 3.
  - `xstep=nil or 1` and `ystep=nil or 1`. This option specifies the step size on the axes. If the axis is non-logarithmic, the default value is 1. If the axis is logarithmic, this option represents the difference between the first graduation (the one corresponding to the minimum value) and the second main graduation. If there are multiple decades, this option defaults to `nil`, and each decade is divided into 9 parts. If there is only one decade, this option defaults to one-tenth of the range of values.
  - `defaultloglabels={2,3,5,10}`. This list of values is used to determine the values per decade on the logarithmic axes.
  - `xdecadeloglabels=nil` and `ydecadeloglabels=nil`. This option applies to logarithmic axes. It specifies the list of values for the first decade; these values will provide the main graduations and labels for all decades. If the minimum value on the axis is  $vmin$ , then the first decade is the interval  $[vmin; 10 \times vmin]$ . By default, this list of values is calculated as:  $\{2 \times vmin, 3 \times vmin, 5 \times vmin, 10 \times vmin\}$  (it is unnecessary to specify the first value  $vmin$ , as it is automatically added). The list  $\{2, 3, 5, 10\}$  is the default value for the `defaultloglabels` option.
  - `xexponent=0` and `yexponent=0`. This option applies to logarithmic axes. For example, when `xexponent=2`, all the labels on the  $x$  axis are divided by  $10^2$ , and the string  $(\times 10^2)$  will be added to the legend. The same principle applies to the  $y$  axis if it is logarithmic.
  - `xaddloglabels={}` and `yaddloglabels={}`. This option applies to logarithmic axes. It allows you to add a list of values that will provide additional labels; these values must be between the minimum and maximum values of the axis.
  - `tickpos={0.5,0.5}`. This option specifies the position of the tick marks relative to each axis. These are two numbers between 0 and 1. The default value of 0.5 means they are centered on the axis (0 and 1 represent the endpoints).
  - `xyticks={0.2,0.2}`. This option specifies the length of the tick marks on the axes.
  - `xylabelsep={0,0}`. This option specifies the distance between the labels and the tick marks on the axes.
  - `originloc=<lower left corner>`. This option specifies the origin point of the tick marks for the non-logarithmic axis.
  - `originnum={minimum,minimum}`. This option specifies the value at the origin point of the tick marks (tick number 0) for the non-logarithmic axis. The formula that defines the label at tick number  $n$  is:
 
$$(\text{originnum} + \text{unit} \times n) \text{ "labeltext" / labelden.}$$
  - `legend={"", ""}`. This option specifies a legend for the axes. By default, the  $x$ -axis legend is located below the axis and in the middle, and the  $y$ -axis legend is located to the left of the axis, in the middle, and is written vertically.
  - `legendpos={0.5,0.5}`. This option specifies the position (between 0 and 1) of the legend relative to each axis.
  - `legendsep={-0.5,-1}`. This option specifies the distance between the legend and the axis. The legend is on the opposite side of the axis from the graduations.

- `legendangle={0,90}`. This option specifies the angle (in degrees) that the legend should make with the axis.
- `legendstyle={"S","N"}`. Specifies the label style for the legends, with the value `"auto"` it is determined automatically, otherwise you can use the values: `"N"`, `"NW"`, `"W"`, `"SW"`, `"S"`, `"SE"`, `"E"`, `"NE"`.
- `labelpos={"bottom","left"}`. This option specifies the position of the labels relative to the axis. For the  $Ox$  axis, the possible values are: `"none"`, `"bottom"`, or `"top"`; for the  $Oy$  axis, it is: `"none"`, `"right"`, or `"left"`.
- `labelstyle={"S","W"}`. This option defines the style of the labels for each axis. The possible values are: `"auto"`, `"N"`, `"NW"`, `"W"`, `"SW"`, `"S"`, `"SE"`, `"E"`, `"NE"`.
- `labelangle={0,0}`. This option defines the angle of the labels in degrees relative to the horizontal for each axis.
- `labelcolor={"",""}`. This option allows you to choose a color for the labels on each axis. The empty string represents the default color.
- `labelden={1,1}`. This option specifies the denominator of the labels (integer) for the non-logarithmic axis.
- `labeltext={"",""}`. This option defines the text that will be added to the numerator of the labels for the non-logarithmic axis.
- `xynode_options=""`. String that will be passed as is to the `\node{}` instruction for all labels on both axes (but not for legends).
- `xnode_options=xynode_options`. String that will be passed as is to the `\node{}` instruction for all labels on the x-axis (but not for the legend).
- `ynode_options=xynode_options`. String that will be passed as is to the `\node{}` instruction for all labels on the y-axis (but not for the legend).
- `use_siunitx={siunitx,siunitx}`. This option specifies whether numeric values should be formatted using the `siunitx` package; the default value is that of the global variable `siunitx`, which is `false` by default.
- `mylabels=""`. This option allows you to apply custom labels to the non-logarithmic axis. When labels are required, the value passed to this option must be a list of the type: `{pos1,"text1",pos2,"text2",...}`. The number `<pos1>` represents an x-coordinate in the coordinate system ( $A, \text{step}$ ) ( $A$  being the origin of the axis), which corresponds to the point with affix  $A+pos1*step$ , the step being either `xstep` or `ystep` along the axis.
- `gridstyle="solid"`. This option defines the line style for the primary grid.
- `subgridstyle="solid"`. This option defines the line style for the secondary grid. A secondary grid appears when there are subdivisions on one of the axes.
- `gridcolor="gray"`. This sets the color of the primary grid.
- `subgridcolor="lightgray"`. This sets the color of the secondary grid.
- `gridwidth=4`. Line thickness of the primary grid (which is 0.4pt).
- `subgridwidth=2`. Line thickness of the secondary grid (which is 0.2pt).

### Drawing and Conversion Methods

- The `g:Dlogpolyline(L [, close, draw_options])` method draws the 2D polygonal line  $\langle L \rangle$  on the grid; coordinate conversion is automatic. The `<close>` argument is a boolean indicating whether the line should be closed (`false` by default), and the `<draw_options>` argument is a string (empty by default) that will be passed to the `\draw` instruction.
- The method `g:Dlogdots(L [, mark_options])` draws the scatter plot contained in  $\langle L \rangle$  (a complex number, a list of complex numbers, or a list of lists of complex numbers) on the grid. Coordinate conversion is automatic. The argument `<mark_options>` is a string (empty by default) that will be passed to the `\draw` instruction.
- The method `g:Dlogline(A, B [, draw_options])` draws the line passing through points  $\langle A \rangle$  and  $\langle B \rangle$  on the grid (coordinate conversion is automatic). The argument `<draw_options>` is a string (empty by default) that will be passed to the `\draw` instruction.
- Conversion function: `ld.Zlog(z)` where  $\langle z \rangle$  is a complex number, returns the affix on the grid of the corresponding point.

**End of grid usage: g:Endlogview()**

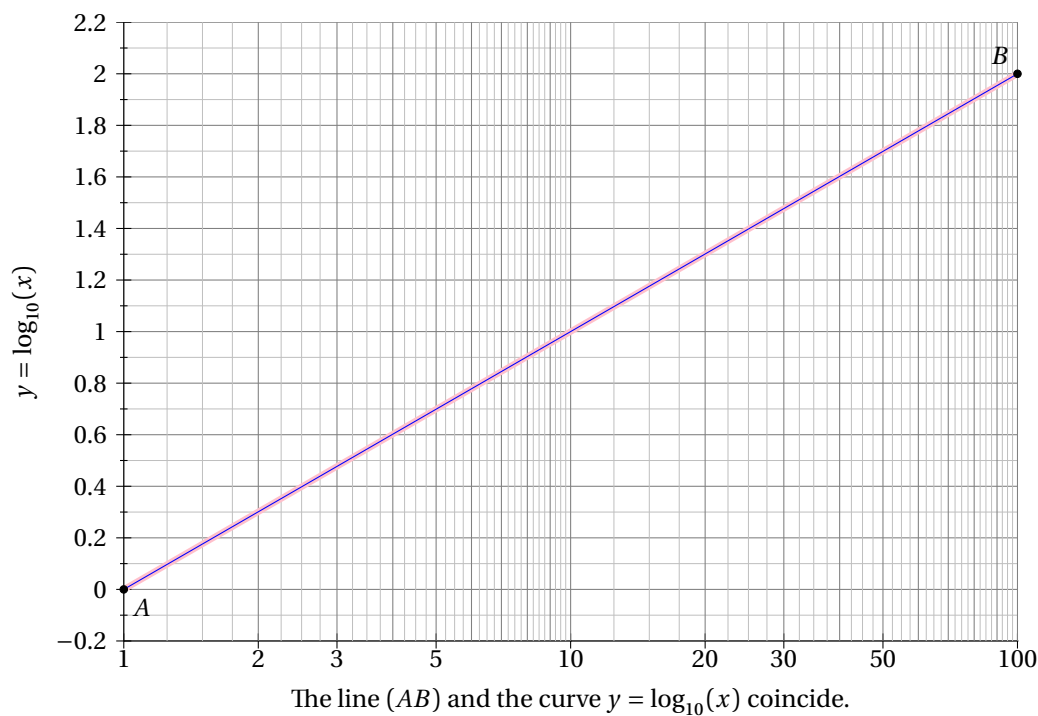
The **g:Endlogview()** method restores the initial graph matrix and the 2D window.

**Examples**

```
\begin{luadraw}{name=logx_example}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z
local log10 = math.log10

local g = ld.graph:new{ size={14,10,0} }
require 'luadraw_log_axes'
g:Beginlogview("logx", 1, 100, -0.2, 2.2, {
  nsubdiv={3,1},
  ystep = 0.2,
  legend={"The line $(AB)$ and the curve $y=\log_{10}(x)$ coincide.", "$y=\log_{10}(x)$"}
})
local L = ld.cartesian(function(x) return log10(x) end, 1,100)
local A, B = 1, Z(100,2)
g:Dlogline(A,B, "Pink,line width=2.4pt")
g:Dlogpolyline(L,"blue")
g:Dlogdots({A,B})
g:Dloglabel("$A$", A, {pos="SE"}, "$B$", B, {pos="NW"})
g:Endlogview()
g:Show()
\end{luadraw}
```

Figure 20: Logarithmic x-axis



```
\begin{luadraw}{name=logy_and_logxy_example}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z
local exp = math.exp

local g = ld.graph:new{ window={-10,10,-5,5}, size={18,10,0} }
require 'luadraw_log_axes'
g:Beginlogview("logy", -1, 5, 0.2, 200, { viewport={0.25,10,-5,4.5},
```

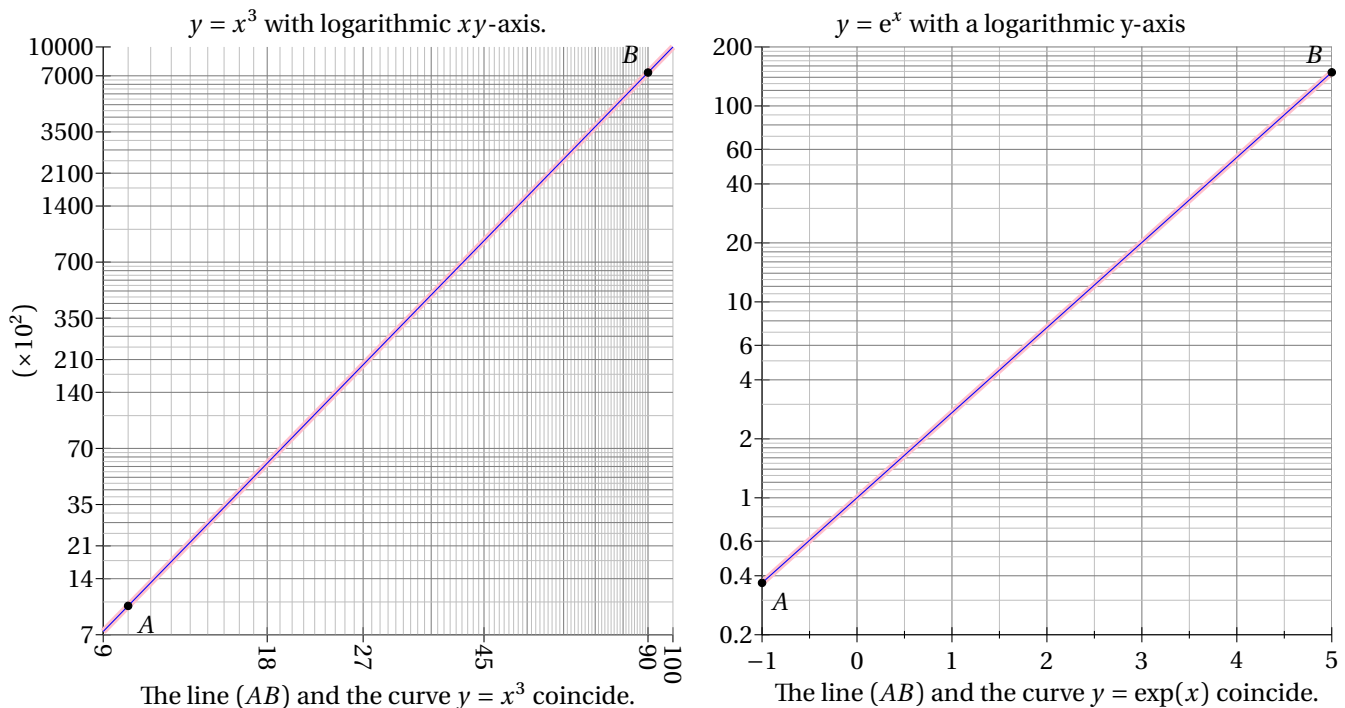
```

nbsubdiv = {1,1},
legendsep = {-0.5,-0.5},
legend = {"The line $(AB)$ and the curve $y=\exp(x)$ coincide.", ""}
})
local L = ld.cartesian(exp, -1, 5)
local A, B = Z(-1,exp(-1)), Z(5,exp(5))
g:Dlogline(A,B, "Pink,line width=2.4pt")
g:Dlogpolyline(L,"blue")
g:Dlogdots({A,B})
g:Dloglabel("$A$", A, {pos="SE"}, "$B$", B, {pos="NW"})
g:Endlogview()

g:Beginlogview("logxy", 9, 100, 7e2, 1e6, { viewport={-10,-0.25,-5,4.5},
nbsubdiv = {8,1},
xaddloglabels={100},
legendsep = {-0.5,-0.75},
yexponent=2,
yaddloglabels={1e6},
labelangle={-90,0}, labelstyle={"E","W"},
legend = {"The line $(AB)$ and the curve $y=x^3$ coincide.", ""}
})
local L = ld.cartesian(function(x) return x^3 end, 9, 100)
local A, B = Z(10,1e3), Z(90,729e3)
g:Dlogline(A,B, "Pink,line width=2.4pt")
g:Dlogpolyline(L,"blue")
g:Dlogdots({A,B})
g:Dloglabel("$A$", A, {pos="SE"}, "$B$", B, {pos="NW"})
g:Endlogview()
-- drawing on the "normal" 2D window
g:Dlabel("$y=\mathrm{e}^x$ with a logarithmic y-axis",Z(5,4.5),{pos="N"},
"$y=x^3$ with logarithmic $xy$-axis.",Z(-5,4.5),{})
g:Show()
\end{luadraw}

```

Figure 21: Logarithmic x-axis and logarithmic xy-axis



## 10) The *luadraw\_decorations* module

The `luadraw_decorations` module redefines certain graphics methods to add options such as adding a label, but for each of them, the old syntax is still valid. This module does not return anything.



## 2D Polygonal Lines: `g:Dpolyline()`

The method `g:Dpolyline(L, options)` draws the polygonal line  $\langle L \rangle$  (a list of complex numbers or a list of lists of complex numbers). The argument  $\langle options \rangle$  is an array whose fields define the possible options, which are (with their default values):

- `close=false`: a boolean indicating whether the line  $\langle L \rangle$  should be closed.
- `clip=nil`: This option is either `nil` (default value) or a table of the form  $\{x1, x2, y1, y2\}$ . In the first case, the row is clipped by the current 2D window **after** its transformation by the 2D matrix of the graph. In the second case, the row is clipped by the window  $[x_1; x_2] \times [y_1; y_2]$  **before** being transformed by the matrix of the graph.
- `draw_options=""`: String (empty by default) that will be passed as is to the `\draw` instruction.
- Options for adding a label:
  - `label=""`: label to add.
  - `anchor1d=nil`: number between 0 and 1 indicating the label's position along line  $\langle L \rangle$  (0 for the beginning of the line, 1 for the end of the line).
  - `anchor=nil`: complex number representing the label's anchor point in the plane.
  - `anchor2d=cpx.Z(0.5,0.5)`: complex number representing the label's position within the bounding box  $[0; 1] \times [0; 1]$  of line  $\langle L \rangle$ ; therefore, by default, the label is at the center of this box.  
The order of priority is: `anchor`, `anchor1d`, `anchor2d` (if the first option is `nil`, the second is chosen; if it is also `nil`, then the third is chosen).
  - `pos="center"` indicates the label's position relative to the anchor point. It can be `"center"` (centered), `"N"` (north), `"NE"` (northeast), `"E"` (east), `"SE"` (southeast), `"S"` (south), `"SW"` (southwest), `"W"` (west), `"NW"` (northwest). By default, it is `center`, and in this case, the label is centered on the anchor point.
  - `dist=0`, the distance in cm between the label and its anchor point when `pos` is not equal to `"center"`.
  - `dir={}`, this option is a table of the form  $\{dirX, dirY, dep\}$  which indicates the writing direction. The three values `dirX`, `dirY`, and `dep` are three complex numbers representing three vectors. The first two indicate the writing direction, and the third indicates a displacement (translation) of the label relative to the anchor point. The vector `dep` is zero by default, and the vector `dirY`, if absent, is equal to the vector `dirX` rotated 90 degrees in the clockwise direction. By default, the `dir` option is equal to the current writing direction.
  - `node_options=""` is a string (empty by default) intended to receive options that will be directly passed to TikZ in the `node[]` instruction.
  - `showanchor=false`: with the value `true`, the anchor point is displayed along with the label's bounding box.

## 2D Paths: `g:Dpath()`

The method `g:Dpath(P, options)` draws path  $\langle P \rangle$ . The  $\langle options \rangle$  are the same as for the method `g:Dpolyline()` except for the options `close` and `clip`, which are ignored.

## 2D Lines: `g:Dline()`

The method `g:Dline(d, options)` draws the line  $\langle d \rangle$ , which is a list of the form  $\{A, u\}$  where  $A$  represents a point on the line (a complex number) and  $u$  a direction vector (a non-zero complex number).

Variant: the method `g:Dline(A, B, options)` draws the line passing through the points  $\langle A \rangle$  and  $\langle B \rangle$  (two complex numbers).

In both cases, the  $\langle options \rangle$  are the same as for the `g:Dpolyline()` method except for the `close` and `clip` options which are not taken into account, and the following option:

- `scale=1`. The option `scale` (which defaults to 1) is either a number (percentage) to vary the length of the displayed segment (from its midpoint), or an array of two numbers (percentages)  $\{scaleA, scaleB\}$  to vary the length of the displayed segment at each of its endpoints.



**2D Segments: g:Dseg()**

The method **g:Dseg(seg, options)** where  $\langle seg \rangle = \{A, B\}$  with  $A$  and  $B$  being two complex numbers, draws the segment  $[A; B]$ . The  $\langle options \rangle$  are the same as for the method **g:Dpolyline()** except for the **close** and **clip** options, which are ignored, and the following two options:

- **scale=1**: The option **scale** (which defaults to 1) is either a number (percentage) to vary the length of the displayed segment (from its midpoint), or an array of two numbers (percentages)  $\{scaleA, scaleB\}$  to vary the length of the displayed segment at each of its endpoints.
- **ticks="none"**: allows adding or removing tick marks (graduations) on the segment. The syntax for adding them is: **ticks={nb [, length, space, draw\_options]}** where  $\langle nb \rangle$  is the number of strokes to add.

**2D Arc: g:Darc()**

The **g:Darc(B, A, C, r, direction, options)** method draws a circular arc centered at  $\langle A \rangle$  (a complex number), with radius  $\langle r \rangle$ , extending from  $\langle B \rangle$  (a complex number) to  $\langle C \rangle$  (a complex number) in the counterclockwise direction if the argument  $\langle direction \rangle$  is 1, and in the opposite direction otherwise. The argument ' $\langle options \rangle$ ' is a table whose fields define the possible options. These are (with their default values):

- **arc\_options=""**: string passed to the `\draw` instruction for drawing the arc.
- **sector\_options=""**: string defining the fill mode for the angular sector.
- **label=""**: text to be displayed.
- **node\_options=""**: string defining the options for the label.
- **pos="auto"**: specifies the label's position relative to the anchor point. Other possible values are the usual values for positioning a label: **"center"**, **"N"**, etc. By default, the anchor point is located at the intersection of the circular arc and the angle bisector.
- **dist=r**: distance between the anchor point and the center of the circle; by default, this is the radius  $\langle r \rangle$  of the circle.
- **angle=0**: angle (in degrees) of rotation that the anchor point should make by default around the center of the circle.
- **rotate="none"**: indicates whether the label should be rotated around its anchor point. Other possible values are: **"auto"**, in which case the label is written along the angle bisector, or **"ortho"**, in which case the label is written perpendicular to the angle bisector.
- **ticks="none"**: allows you to add or not add markings (graduations) on the arc of a circle for an acute angle. The syntax for adding them is: **ticks={nb [, length, space, draw\_options]}** where  $\langle nb \rangle$  is the number of lines to add.
- **showanchor=false**: with the value **true**, the anchor point is drawn, as well as the bisector and the box around the label.

```
\begin{luadraw}{name=decoratedarc2D}
local ld = luadraw
local cpx = ld.cpx
local Z, i = cpx.Z, cpx.I

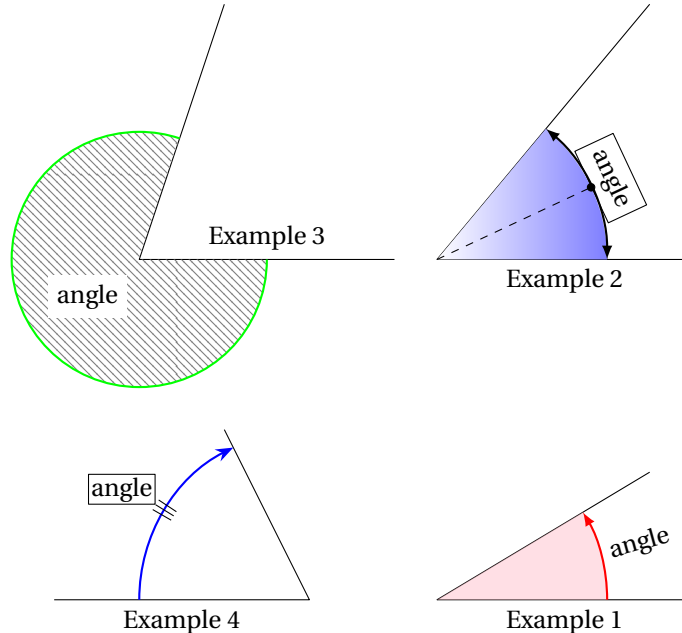
local g = ld.graph:new{window={-5,3,-3,5}, size={10,10}}
require 'luadraw_decorations'
g:Shift(Z(0,-2)) -- example 1
local b,a,c,r = 3, 0, 2.5+1.5*i, 2
g:Dpolyline({b,a,c})
g:Darc(b,a,c,r,1, {
  label="angle", rotate="auto",
  sector_options="fill=Pink,opacity=0.5",
  arc_options="red,line width=0.8pt,-latex" })
g:Dlabel("Example 1",1.5,{pos="S"})
```

```

g:Shift(Z(0,4)) -- example 2
b,a,c,r = 3, 0, 2.5+3*i, 2
g:Dpolyline({b,a,c})
g:Darc(b,a,c,r,1, {
    label="angle", rotate="ortho",
    sector_options="left color=white, right color=blue!50",
    arc_options="line width=0.8pt,latex-latex",
    showanchor=true })
g:Dlabel("Example 2",1.5,{pos="S"})
g:Shift(Z(-3.5,0)) -- example 3
b,a,c,r = 3, 0, 1+3*i, 1.5
g:Dpolyline({b,a,c})
g:Darc(b,a,c,r,-1, {
    label="angle", dist=r/2, pos="center",
    node_options="fill=white",
    sector_options="pattern=north west lines,pattern color=gray",
    arc_options="line width=0.8pt,green" })
g:Dlabel("Example 3",1.5,{pos="N"})
g:Shift(Z(2,-4)) -- example 4
b,a,c,r = -3, 0, -1+2*i, 2
g:Dpolyline({b,a,c})
g:Darc(b,a,c,r,-1, {
    label="angle", dist=r+0.125,
    node_options="draw,inner sep=1pt",
    arc_options="line width=0.8pt,blue,-Stealth",
    ticks=3 })
g:Dlabel("Example 4",-1.5,{pos="S"})
g:Show()
\end{luadraw}

```

Figure 22: g:Darc() Method



### 3D Arc: g:Darc3d()

The method **g:Darc3d(B, A, C, r, direction, options)** draws a circular arc centered at  $\langle A \rangle$  (3D point), with radius  $\langle r \rangle$ , extending from  $\langle B \rangle$  (3D point) to  $\langle C \rangle$  (3D point) in the clockwise direction if the argument  $\langle direction \rangle$  is 1, and in the counterclockwise direction otherwise. This arc is drawn in the plane containing the three points  $\langle A \rangle$ ,  $\langle B \rangle$ , and  $\langle C \rangle$ . When these three points are collinear, the option **normal** (non-zero 3D point) must be specified, which represents a normal vector to the plane. This plane is oriented by the cross product  $\vec{AB} \wedge \vec{AC}$  or by the vector **normal** if specified. The argument  $\langle options \rangle$  is an array

whose fields define the possible options, which are (with their default values):

- `normalfil`: a non-zero 3D point representing a normal vector to plane  $(ABC)$ . It is optional if the three points are not collinear.
- `arc_options=""`: a string passed to the `\draw` instruction for drawing the arc.
- `sector_options=""`: a string defining the fill mode for the angular sector.
- `label=""`: the text to be displayed.
- `node_options=""`: String defining the options for the label.
- `pos="auto"`: Specifies the label's position relative to the anchor point. Other possible values are the usual values for positioning a label: `"center"`, `"N"`, `"NW"`, etc. By default, the anchor point is located at the intersection of the arc of the circle and the angle bisector.
- `dist=r`: Distance between the anchor point and the center of the circle  $(A)$ . By default, this is the radius  $\langle r \rangle$  of the circle.
- `angle=0`: Angle (in degrees) of rotation that the anchor point should make by default around the center of the circle  $(A)$  and in the plane of the circle.
- `rotate="none"`: indicates whether the label should be rotated around its anchor point **in the screen plane**. Other possible values are: `"auto"`, in which case the label is written along the bisector, or `"ortho"`, in which case the label is written perpendicular to the bisector.
- `rotate3d="none"`: Indicates whether the label should be rotated around its anchor point **in the plane**  $((ABC))$ . Other possible values are: `"auto"`, in which case the label is written along the angle bisector, or `"ortho"`, in which case the label is written perpendicular to the angle bisector (still in the plane  $(ABC)$ ). With the value `"none"`, the label is written in the screen plane.
- `ticks="none"`: Allows you to add or not add markings (graduations) on the arc of a circle for an acute angle. The syntax for adding them is: `ticks={nb [, length, space, draw_options]}` where  $\langle nb \rangle$  is the number of lines to add.
- `showanchor=false`: With the value `true`, the anchor point is drawn, along with the bisector and the box around the label.
- `bezier=true`: To draw the arc with Bézier curves, with the value `false`, the arc is a polygonal line. When TikZ adds an arrow to the end of a Bézier curve, it undergoes a slight deformation that can create an artifact when the angular sector needs to be painted; in this case, it is better to use the `bezier=false` option.

```
\begin{luadraw}{name=decorated_arcs3D}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M

local g = ld.graph3d:new{ window={-3,8,-5,8}, viewdir={"xy",0.8,60}, size={10,10} }
require 'luadraw_decorations'
local O, P = Origin, M(6.5, 6, 4)
local Px,Pz,Py,Pxz = ld.px(P), ld.pz(P), ld.py(P), ld.pxz(P)

g:Dpolyline3d({{0, 7*vecI}, {0, 7*vecJ}, {0, 6*vecK}}, {"-Stealth,solid,black, line width=1pt"})
g:Dlabel3d("$x$", 7*vecI, {pos="E"}, "$y$", 7*vecJ, {pos="N"}, "$z$", 6*vecK, {pos="S"})
g:Dpolyline3d({{P,Py}, {P,Pxz}, {Pxz,Px}, {0,Pxz}, {Pxz,Pz}}, {"dashed,blue,thick"})
g:Dseg3d({0, P}, {"-latex,RosyBrown,line width=4pt"})
g:Dpolyline3d({{0, 2.5*vecI}, {0, 2.5*vecJ}, {0, 2.5*vecK}}, {"-latex,cyan,line width=3pt"})

g:Darc3d(Px, O, P, 4, 1, {
  arc_options = "-Stealth,blue,ultra thick",
  sector_options = "fill=blue,opacity=0.2",
})
```

```

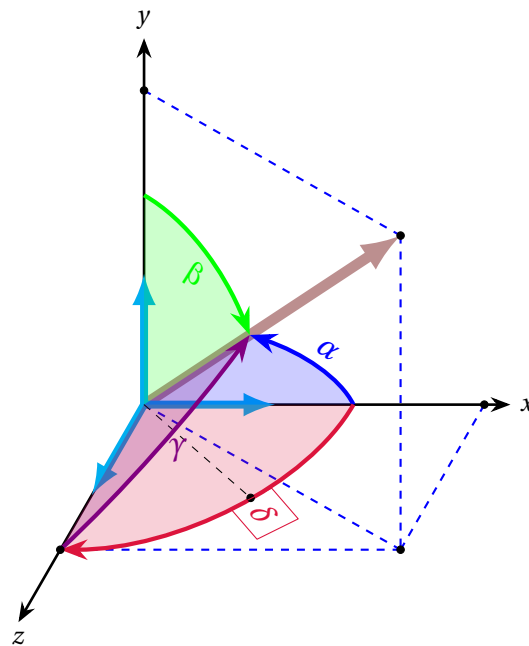
label = "$\\alpha$", node_options="blue,scale=1.25",
rotate = "ortho", pos="N" })

g:Darc3d(Py, 0, P, 4, 1, {
  arc_options = "-Stealth,green,ultra thick",
  sector_options = "fill=green,opacity=0.2",
  label = "$\\beta$", node_options="green,scale=1.25",
  pos = "S", rotate3d = "ortho" })

g:Darc3d(Px, 0, Pz, 4, 1, {
  arc_options = "-Stealth,Crimson,ultra thick",
  sector_options = "fill=Crimson,opacity=0.2",
  label = "$\\delta$", node_options="Crimson,scale=1.25",
  rotate3d = "auto", angle=-5,
  showanchor=true })

g:Darc3d(Pz, 0, P, 4, 1, {
  arc_options = "-Stealth,violet,ultra thick",
  sector_options = "fill=violet,opacity=0.2",
  label = "$\\gamma$", node_options="violet,scale=1.25",
  pos="E" })
g:Ddots3d({P,Pxz,Px,Py,Pz})
g:Show()
\\end{luadraw}

```

Figure 23: Method `g:Ddecoratedarc3d()`**Conclusion :**

1. In all 2D or 3D line drawing methods that call one of the previous methods, for those that have an argument `<draw_options>`, which is normally a string, this can be replaced by an array `<options>` as in the new method `g:Dpolyline(L, options)`.
2. In all 2D or 3D line drawing methods that already have an argument `<options>` (or `<args>`), and that have one of these options called `draw_options`, which is normally a string, this can be replaced by an array `<options>` as in the new method `g:Dpolyline(L, options)`.

```

\\begin{luadraw}{name=decorations2d}
local ld = luadraw

```

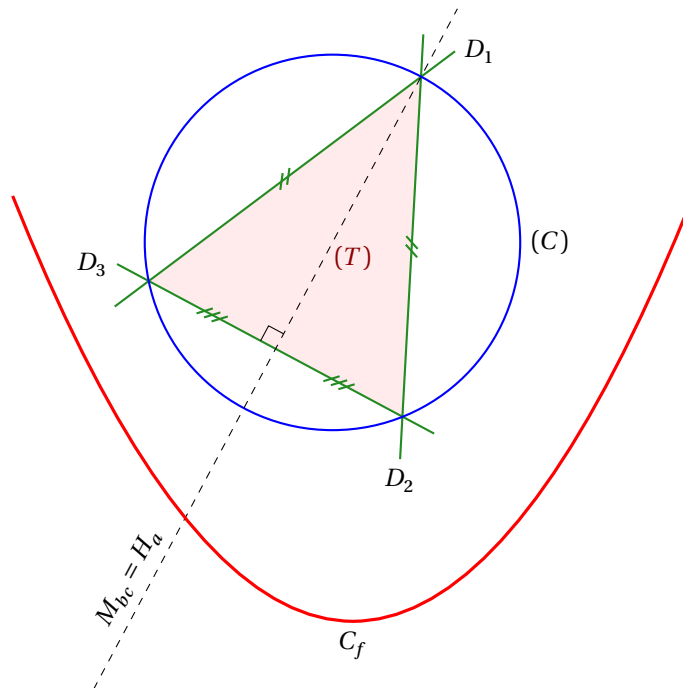
```

local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{size={10,10}}
require 'luadraw_decorations'
local A, B = Z(1,4), Z(-3,1)
local C = ld.rotate(B,50,A)
local A1 = (B+C)/2
g:Dpolyline({A,B,C}, {close=true, draw_options="draw=none,fill=pink,fill opacity=0.3",
    label="$ (T) $" , anchor=(A+B+C)/3, pos="E", node_options="DarkRed"})
g:Lineoptions(nil, "ForestGreen", 8)
g:Dseg({A,B}, {scale=1.25,ticks=2,label="$D_1$", anchor1d=0, pos="E"})
g:Dseg({A,C}, {scale=1.25,ticks=2,label="$D_2$", anchor1d=1, pos="S"})
g:Dseg({C,B}, {scale=1.25,label="$D_3$", anchor1d=1, pos="W"})
g:Dmarkseg(B,A1,3); g:Dmarkseg(A1,C,3); g:Dangle(B,A1,A,0.25, "black,thin")
g:Dcircle({A,B,C}, {label='$ (C) $' , anchor1d=0, pos="E", draw_options="blue"})
g:Dmed(B,C,{label="$M_{bc}=H_a$", anchor1d=0.15, pos="N", dir=cpx.I*(C-B), draw_options="dashed,black,thin"})
g:Dcartesian( function(x) return (x/2)^2-4 end,{x={-5,5},
    draw_options={label="$C_f$", anchor2d=Z(0.5,0), pos="S", draw_options="red,line width=1.2pt"} })
g:Show()
\end{luadraw}

```

Figure 24: 2D Decorations



```

\begin{luadraw}{name=decorations3d}
local ld = luadraw
local pt3d = ld.pt3d
local Origin, vecI, vecJ, vecK, M, Z = pt3d.Origin, pt3d.vecI, pt3d.vecJ, pt3d.vecK, pt3d.M, ld.cpx.Z

local g = ld.graph3d:new{size={10,10}}
require 'luadraw_decorations'
g:Dline3d(-5*vecI,5*vecI, {label="$x$", anchor=5*vecI, pos="S", draw_options="-stealth"}, true)
g:Dline3d(-5*vecJ,5*vecJ, {label="$y$", anchor1d=1, pos="SE", draw_options="-stealth"}, true)
g:Dcircle3d(Origin, 3, vecK, {label="$ (C) $" , anchor1d=0, pos="SE",
    dir={vecJ,-vecI}, draw_options="red,thick", node_options="red"})
local nb = 15
local H = ld.linspace(0,3,nb)
for k = 1, nb-1 do
    local z = H[k]
    local r = math.sqrt(9-z^2)

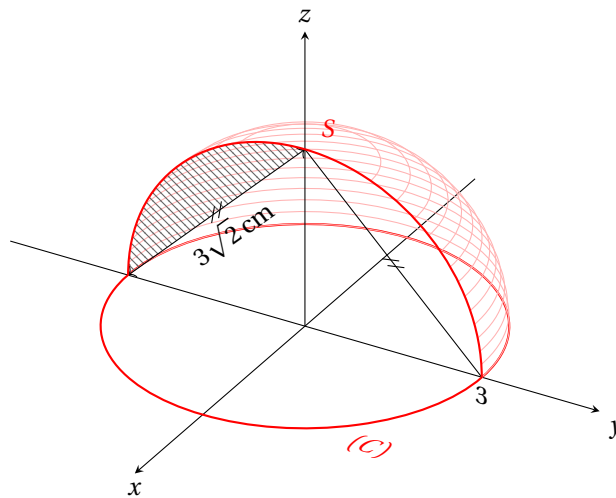
```

```

g:Darc3d(M(0,-r,z),z*vecK,M(0,r,z),r,-1,vecK,"thin, red!30")
end
g:Dpath3d({-3*vecJ,Origin,3*vecK,3,-1,vecI,"ca",3*vecK,"l","cl"},
"draw=none,pattern color=black!60,pattern=north west lines")
g:Dseg3d({Origin,5*vecK},
{label="$z$",anchor2d=Z(0,1),pos="N",draw_options="-stealth"})
g:Dseg3d({-3*vecJ,3*vecK}, {label="$3\\sqrt{2}$\\,cm", pos="S", dir={M(0,1,1),M(0,-1,1)},
ticks=2, draw_options="<->"})
g:Dseg3d({3*vecJ,3*vecK},
{ticks=2, label="$3$", anchor=3*vecJ,pos="S",dir={vecJ,vecK}})
g:Dpath3d({-3*vecJ,Origin,3*vecJ,3,-1,vecI,"ca"},
{label="$S$", anchorId=0.5, pos="N",dist=0.15,draw_options="red,thick", node_options="red"})
g:Show()
\\end{luadraw}

```

Figure 25: 3D Decorations



## 11) The *luadraw\_coils\_chains* module

The module *luadraw\_coils\_chains* allows you to draw springs (two types) and chains (two types). It adds new graphing methods to the *ld.graph* class but returns nothing.

### Springs

- The method **g:Dcoil(list, radius, options)** allows you to draw a spring. The argument  $\langle radius \rangle$  is the radius of the spring. The argument  $\langle list \rangle$  has two possible forms:
  - Form 1:  $\langle list \rangle = \{a_1, n_1, a_2, n_2, \dots, a_N\}$ , where  $a_1, \dots, a_N$  are points (complex numbers),  $n_1$  is an integer, representing the number of turns between  $a_1$  and  $a_2$ ,  $n_2$  is the number of turns between  $a_2$  and  $a_3$ , and so on.
  - Form 2:  $\langle list \rangle = \{C, n\}$ , where  $C$  is a polygonal line (for example, a curve) and  $n$  is an integer representing the total number of turns. The spring will be drawn along the curve  $C$ .

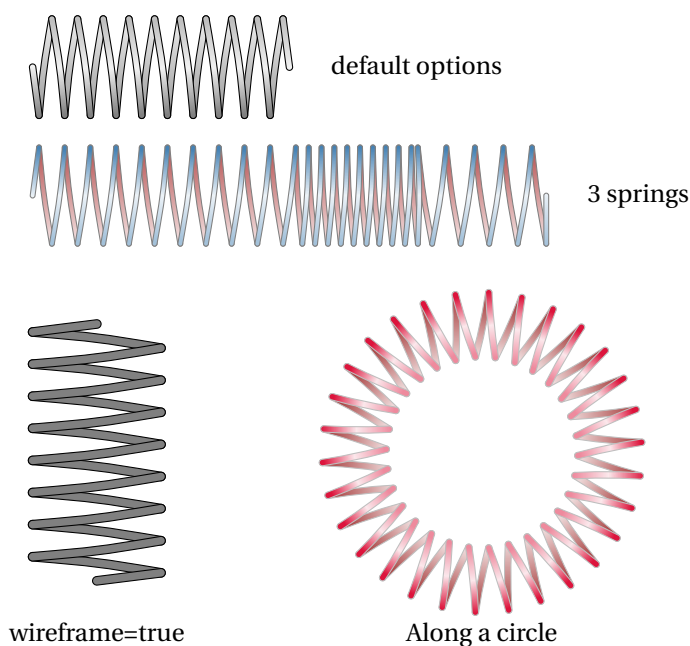
The  $\langle options \rangle$  argument is a table whose fields are the options that will modify the aesthetic appearance of the spring. These options are (with their default values):

- **direction=1**: Direction of rotation of the coils (1 for clockwise, -1 for counterclockwise).
- **wire\_dia=<2\*radius/15>**: Wire diameter.

- `color="gray"`: Fill color of the coils.
- `colorB=color`: Fill color of the "back" coils; by default, this is the same value as the `color` option.
- `border=<current color>`: Color of the coil outline.
- `border_width=<current thickness>`: Thickness of the coil outline.
- `tension=1`: The coils are slightly curved to give a 3D effect; this option allows you to increase or decrease the curvature. With a value of 0, there is no curvature.
- `start_angle=nil`, `end_angle=nil`: angles (in degrees) for the first and last half-turns. These angles are determined automatically, but they can be modified with these options.
- `wireframe=false`: with the value `true`, the turns are filled with a solid color, but with the value `false` (default), they are filled with a gradient, specifically with the following formula:  
`left color=<color>!<leftC>`, `right color=<color>!<rightC>`, `middle color=<color>!<midC>`  
with the three coefficients being options: `leftC=100`, `rightC=50`, `midC=10`.
- `holes=false`: With the value `true`, a dot is drawn at the ends of the spring.
- `reverse=false`: Reverses the display order of the spring when it is made up of multiple pieces.

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

local g = ld.graph:new{window={-5.6,5,-5,5}, size={10,10}}
require 'luadraw_coils_chains'
local a, b, c, d = Z(-5,4), Z(-1,4), Z(1,4), Z(3,4)
g:Dcoil({a,10,b}, 0.75, {}); g:Dlabel("default options", b+0.5, {pos="E"})
g:Shift(Z(0,-2))
g:Dcoil({a,10,b,10,c,5,d}, 0.75, {color="SteelBlue", direction=-1, wire_dia=0.075,
border="gray", colorB="Brown"})
g:Dlabel("3 springs", d+0.5, {pos="E"})
g:Shift(Z(0,-2))
g:Dcoil({Z(-4,4),8, Z(-4,0)}, 1, {wireframe=true}); g:Dlabel("wireframe=true",Z(-4,-0.5),{pos="S"})
local C = ld.circle(Z(2,2),2)
g:Dcoil( {C,30}, 0.5, {color="Crimson", wire_dia=0.1, border="lightgray", colorB="DarkRed"})
g:Dlabel("Along a circle",Z(2,-0.5),{pos="S"})
g:Show()
\end{luadraw}
```

Figure 26: The `g:Dcoil()` Method

- The method **g:Dcoil2(list, radius, options)** allows you to draw a wire spring. The argument  $\langle radius \rangle$  is the radius of the spring. The argument  $\langle list \rangle$  has two possible forms:
  - Form 1:  $\langle list \rangle = \{a_1, n_1, a_2, n_2, \dots, a_N\}$ , where  $a_1, \dots, a_N$  are points (complex numbers),  $n_1$  is an integer, representing the number of turns between  $a_1$  and  $a_2$ ,  $n_2$  is the number of turns between  $a_2$  and  $a_3$ , and so on.
  - Form 2:  $\langle list \rangle = \{C, n\}$ , where  $C$  is a polygonal line (for example, a curve) and  $n$  is an integer representing the total number of turns. The spring will be drawn along the curve  $C$ .

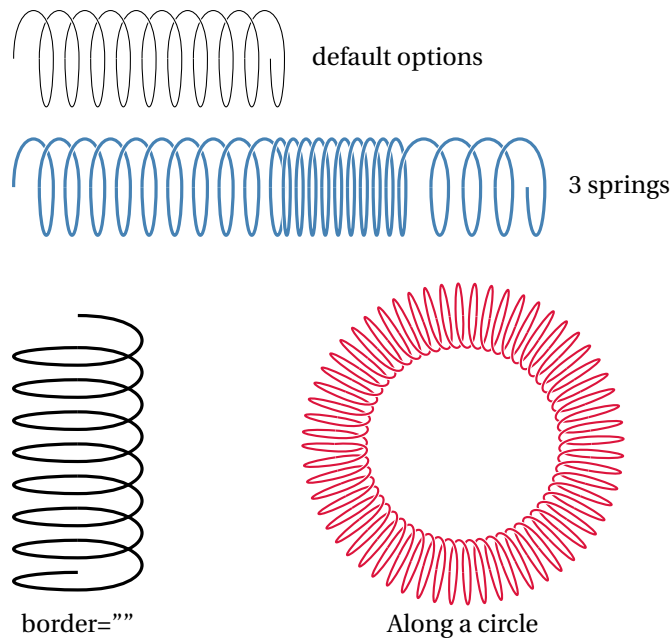
The argument  $\langle options \rangle$  is a table whose fields are the options that will modify the aesthetic appearance of the spring. These options are (with their default values):

- **direction=1**: direction of rotation of the coils (1 for clockwise, -1 for counterclockwise).
- **color=<current color>**: color of the coils.
- **width=<current thickness>**: thickness of the coils (in tenths of a point).
- **border="white"**: this option is either an empty string ("" ) or a string representing a color. In the latter case, the drawing uses the *double* option of TikZ with this color as the border, allowing the "front" coils to be seen passing in front of the "back" coils, like a 3D spring.
- **border\_width=<current thickness>**: border thickness (used when the **border** option is not equal to **nil**).

```
\begin{luadraw}{name=Dcoil2_examples}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{window={-5.6,5,-5,5}, size={10,10}}
require 'luadraw_coils_chains'
local a, b, c, d = Z(-5,4), Z(-1,4), Z(1,4), Z(3,4)
g:Dcoil2({a,10,b}, 0.75, {}); g:Dlabel("default options", b+0.5, {pos="E"})
g:Shift(Z(0,-2))
g:Dcoil2({a,10,b,10,c,4,d}, 0.75, {color="SteelBlue", direction=-1, width=12})
g:Dlabel("3 springs", d+0.5, {pos="E"})
g:Shift(Z(0,-2))
g:Dcoil2({Z(-4,4),8, Z(-4,0)}, 1, {border="", width=12}); g:Dlabel('border="",Z(-4,-0.5),{pos="S"')
local C = ld.circle(Z(2,2),2)
g:Dcoil2( {C,60}, 0.5, {color="Crimson", width=8, border="white"})
g:Dlabel("Along a circle",Z(2,-0.5),{pos="S"})
g:Show()
\end{luadraw}
```



Figure 27: The *g:Dcoil2()* Method

## Chains

- The **g:Dchain(list, link\_length, options)** method allows you to draw a chain. The *list* argument is a list of at least two complex numbers; these are the points through which the chain will pass. The *link\_length* argument is the length of a link in the chain, but this will be adjusted to have a whole number of links. The *options* argument is an array whose fields are the options that will modify the aesthetic appearance of the chain. These options are (with their default values):

- `width=<(calculated length of a link)/1.618>`: width of a link, with the value `width="circle"`, the link will be circular.
- `wire_dia=<width/4>`: wire diameter.
- `color="gray"`: color to fill the links.
- `colorB=color`: Second fill color for the links; by default, it's the same value as the `color` option.
- `border=<current color>`: Color of the link outline.
- `border_width=<current thickness>`: Thickness of the link outline.
- `wireframe=false`: With the value `true`, the links are filled with a solid color, but with the value `false` (the default value) they are filled with a gradient, specifically with the following formula:  

$$\text{left color}=\text{color}!\<\text{leftC}\>, \text{right color}=\text{colorB}!\<\text{rightC}\>, \text{middle color}=\text{color}!\<\text{midC}\>$$
with the three coefficients being options: `leftC=100`, `rightC=50`, `midC=10`.

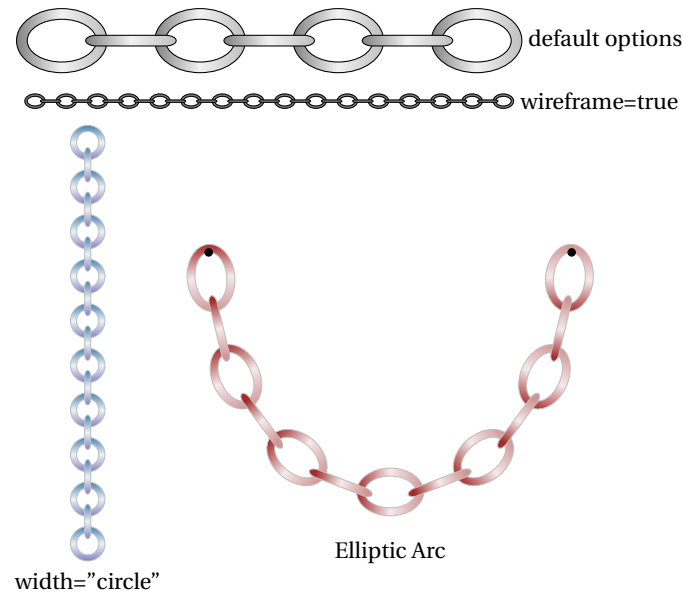
```
\begin{luadraw}{name=Dchain_examples}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

local g = ld.graph:new{window={-5.25,6,-5,5.25}, size={10,10}}
require 'luadraw_coils_chains'
g:Labelsize("small")
local a, b = Z(-5,4.5), Z(3,4.5)
g:Dchain({a,b}, 1, {}); g:Dlabel(" default options", b, {pos="E", dist=0.1})
g:Shift(Z(0,-1))
g:Dchain({a,b}, 0.25, {wireframe=true}); g:Dlabel("wireframe=true", b, {pos="E"})
g:Shift(Z(0,1))
a, b = Z(-4,3), Z(-4,-4)
```

```

g:Dchain({a,b}, 0.35, {color="SteelBlue", colorB="DarkBlue", width="circle", wire_dia=0.1, border="lightgray"})
g:Dlabel('width="circle"', b, {pos="S",dist=0.1})
a, b = Z(-2,1), Z(4,1)
local C = ld.ellipticarc(a,Z(1,1),b,3,4,1)
g:Dchain(C, 0.75, {color="Brown", border="lightgray"}); g:Ddots({a,b})
g:Dlabel("Elliptic Arc",Z(1,-3),{pos="S",dist=0.5})
g:Show()
\end{luadraw}

```

Figure 28: The *g:Dchain()* Method

- The **g:Dchain2(list, R, options)** method draws a chain as two intersecting wavy wires. The *list* argument is a list of at least two complex numbers, representing the points through which the chain will pass. The *R* argument is the radius of the chain (the height of a wave). The *options* argument is an array whose fields are the options that will modify the aesthetic appearance of the chain.

- **direction=1**: direction of rotation of the wires (1 for clockwise, -1 for counterclockwise).
- **color=<current color>**: color of the chain.
- **width=<current thickness>**: thickness of the wires (in tenths of a point).
- **border="white"**: This option is either an empty string ("") or a string representing a color. In the latter case, the drawing uses TikZ's *double* option with that color as the border, thus distinguishing the wire in front from the one behind.
- **border\_width=<current thickness>**: Border thickness (used when the **border** option is not equal to **nil**).

```

\begin{luadraw}{name=Dchain2_examples}
local ld = luadraw
local cpx = ld.cpx
local Z = cpx.Z

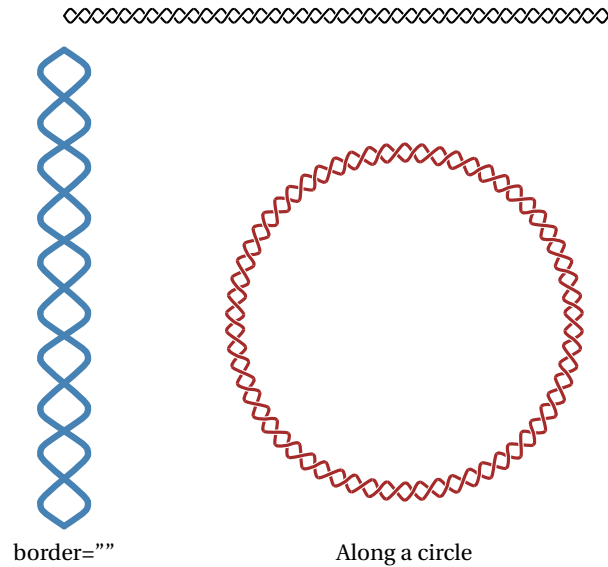
local g = ld.graph:new{window={-5,5,-4.5,5}, size={10,10}}
require 'luadraw_coils_chains'
g:Labelsize("small")
local a, b = Z(-4,4.5), Z(4,4.5)
g:Dchain2({a,b}, 0.1, {width=8})
g:Shift(Z(0,1))
a, b = Z(-4,3), Z(-4,-4)
g:Dchain2({a,b}, 0.35, {color="SteelBlue", width=24, border=""})
g:Dlabel('border=""', b, {pos="S",dist=0.1})
a, b = Z(-2,1), Z(4,1)

```

```

local C = ld.circle(Z(1,-1), 2.5)
g:Dchain2(C, 0.125, {color="Brown", width=12})
g:Dlabel("Along a circle", Z(1,-4), {pos="S", dist=0.1})
g:Show()
\end{luadraw}

```

Figure 29: The *g:Dchain2()* Method

## II History

### 1) Version 3.1

- Added the function **ld.implicit\_inequality()** which returns a **path** representing the outline of the part of the plane located within a certain block and satisfying a condition of the type  $f(x, y) \geq 0$  or  $f(x, y) \leq 0$ .
- Added the method **g:Dimplicit\_inequalities()** which can fill the set of points satisfying a constraint system of the type  $f_i(x, y) > 0$  or  $f_i(x, y) < 0$ .
- Adding the function **plane2rectangle(P, V, L1, L2)** which returns a rectangle (a list of 3D points) to represent this plane. This is the same rectangle drawn by the method **g:Dplane(P, V, L1, L2)**. It can be drawn using the method **g:Dpolyline3d()** or as a facet.
- For the methods **g:addPlane()** and **g:addPlaneEq()** (which are used in **g:Dscene3d()**), the **rectangle** option has been added. When **rectangle=nil** (the default value), the plane is cut by the 3D window and the resulting facet is drawn. When **rectangle={V, L1, L2}**, the plane is drawn as a rectangular facet; this is the same rectangle drawn by the method **g:Dplane(P, V, L1, L2)** (where *P* denotes the plane).
- When creating a 3D graph, the **window3d** option now takes into account the scales on the three axes:

```

window3d={x1,x2,y1,y2,z1,z2 [,xscale,yscale,zscale]}.

```

These three scales are optional and are set to 1 by default; they determine the initial 3D matrix of the graph (which is therefore no longer the identity if one of the scales is different from 1).

- When creating a graph, the **margin** option can now be reduced to a single number when all four margins are equal, for example **margin=0**.
- Added the option **useclip=<boolean>** for the method **g:Dinequalities()** which allows you to choose the technique to use, either contour calculation (with the value **false**, default value), or with a series of clips (with the value **true**).

- The **rectangle()** function now has a second possible syntax: **rectangle(a, b)**, in which case the rectangle has its sides parallel to the axes and has opposite vertices  $\langle a \rangle$  and  $\langle b \rangle$ . The same applies to the corresponding graphical method **g:Drectangle()**.
- For the **line2strip()** function, the argument  $\langle ends \rangle$ , which was previously a boolean, can now take the values **"none"** (equivalent to **false**), **"butt"** (equivalent to **true**), or **"round"**. Boolean values are still accepted.
- The function **line2strip()** now has an additional argument called **mode**, which can be **"center"** (default value), or **"left"**, or **"right"**, in the first case the strip is centered on the polygonal line, in the second case it is on the left of the line, and in the third case it is on the right of the line.
- In the function **read\_csv\_file()**, add the option **comment=<char>**, this indicates the characters that begin a line of comments.
- All methods for drawing lines or half-lines (**g:Dline()**, **g:Dperp()**,...) now have an **scale** option, like the **g:Dseg()** method. This option (which defaults to 1) can be a number (percentage) or a table of two numbers (percentages). The second case allows you to vary the two endpoints separately.
- Bug fixes...

## 2) Version 3.0

This version introduces a major change: all data related to the *luadraw* package is now stored in the namespace (table) named *luadraw*. This necessitates the use of dot notation, for example, **luadraw.graph** instead of **graph**. However, it is possible to create shortcuts; for instance, the instruction **local ld = luadraw** will allow you to use *ld* instead of *luadraw* in dot notation. Refer to the very beginning of this document for more details.

This change results in incompatibility with previous versions; however, the changes required to adapt older code are minimal, especially since there are no changes to the graphics methods (they were already encapsulated in two classes).

This change also brings some (minor) modifications to extensions; refer to the documentation for more details.

- Added a second possible syntax for the functions **ld.surface()** and **ld.obj\_surface()**: **ld.surface(f, mesh)** and **ld.obj\_surface(f, mesh)** where  $\langle mesh \rangle = \{\{u_1, \dots, u_n\}, \{v_1, \dots, v_m\}\}$  (increasing list of values of parameter  $u$  and increasing list of values of parameter  $v$ ).
- In the *luadraw\_decorations* module: the 2D and 3D line drawing methods have been overridden so that the  $\langle draw\_options \rangle$  argument, which is normally a string passed to the  $\backslash draw$  instruction, can be replaced by a table whose fields represent possible options (such as adding a label). The method names remain unchanged, and the old syntax is still valid.
- In the functions **ld.curve2cone()**, **ld.curve2cylinder()**, **ld.line2tube()**, **ld.section2tube()**, **ld.rotcurve()** and **ld.rotline()**, add the option **obj=true/false**; with the value by **false** (default) the functions return a list of facets, with the value **true** they return a table **{vertices={3D points}, facets={{index1,...},...}, normals={3D vectors}}**. If the **g:Dpoly()** method does not take into account the *normals* field, the **g:Pov\_facet()** method of the *luadraw\_povray* module, on the other hand, uses this field when it is present.
- Added the function **ld.cutpolyline2(P,f,sg,close)** where  $\langle P \rangle$  is a polygon (list of complex numbers),  $\langle f \rangle$  is a function ( $x \mapsto f(x) \in \mathbf{R}$ ), and  $\langle sg \rangle$  is a string equal to **">"** or **"<"**. This function returns the outline of the part of the polygon satisfying the constraint  $y > f(x)$  or  $y < f(x)$  depending on the value of  $\langle sg \rangle$ .
- Added the method **g:Dinequalities(constraints, options)** which draws the set of points satisfying a constraint system of the type  $y > f_i(x)$  or  $y < f_i(x)$ .
- In the *luadraw\_povray* module: added the option **usepalette={palette, mode, minmax}**.
- Bug fixes...

### 3) Version 2.8

Non-exhaustive list:

- Added functions: **nth\_root(n,x)** (nth root of a real  $x$ , defined on  $\mathbb{R}$  when  $n$  is odd); **cpx.cosh()**, **cpx.sinh()** (complex hyperbolic cosine and sine) and **cpx.pow(z,a)** (for calculating  $z^a$  with  $z$  a complex number and  $a$  a real number).
- Added the *luadraw\_coils\_chains* extension which allows drawing springs and chains.
- Added the *luadraw\_log\_axes* extension which allows you to create and draw on a logarithmic grid in  $x$ , or in  $y$ , or in  $x$  and  $y$ .
- Added the **use\_siunitx** option for the **g:Daxes()**, **g:DaxeX()**, **g:DaxeY()**, **g:Dgradbox()**, **g:Dgradline()** methods. This allows you to locally use or not use the formatting of numeric values by the *siunitx* package.
- Added the **showlines** option for the **g:Dgrid()** method which allows you to show or hide horizontal and/or vertical lines.
- Added the *luadraw\_decorations* extension, which enhances certain drawing methods by adding options. Currently, there is **g:Ddecoratedarc()** for 2D arcs and **g:Ddecoratedarc3d()** for 3D arcs.
- For the methods **g:Dpoly()**, **g:Dfacet()**, **g:Dmixfacet()**, and **g:addFacet()**, in the option **usepalette={palette, mode}**, the second argument can now be a function,  $\langle mode \rangle: f \mapsto mode(f) \in \mathbb{R}$ , where  $f$  denotes a facet (a list of 3D points). Facets with the smallest value have the first color of the palette, those with the smallest value have the last color of the palette, and for the others, the color is calculated by linear interpolation. Previously, the argument  $\langle mode \rangle$  could only be "x", "y", or "z".
- Added the function **obj\_surface(f,u1,u2,v1,v2 [, grid])** which returns the surface parameterized by  $\langle f \rangle: (u, v) \mapsto f(u, v) \in \mathbb{R}^3$  in *obj* format, that is, a table with three fields {vertices={...}, facets={{...},...}, normals={...}}. The first two fields are identical to the case of polyhedra, and the third field contains the unit vectors normal to the surface at each vertex. The facets are triangular.
- In the *luadraw\_povray* module: a second syntax has been added for drawing smooth parametric surfaces:
 

**g:Pov\_surface(f,u1,u2,v1,v2,options)**

 where  $\langle f \rangle: (u, v) \mapsto f(u, v) \in \mathbb{R}^3$  is the parameterization. This method is faster than the previous one.
- In the *luadraw\_spherical* module: add the global variable **Hiddendelayed = false**. With the value **false**, hidden parts are drawn at the end of the **g:Dspherical()** instruction; with the value **true**, they are drawn at the very end of the current graph, which can be useful if you have added elements after the sphere that hide part of it.
- For the **g:Daxes()** method: the **originloc** option is still the point used as the origin for the graduations, but it is no longer automatically the point of intersection of the two axes.
- For the method **g:Daxes()**: addition of the options **xynode\_options = ""**, **xnode\_options = xynode\_options** and **ynode\_options = xynode\_options** which allow passing options to the instruction `\node{}` for all labels (except legends).
- For the **g:addAaxes()** method, add the option **labels={"\$x\$", "\$y\$", "\$z\$"}** to manage the labels displayed at the end of each axis.
- Bug fixes...

### 4) Version 2.7

Non-exhaustive list:

- The basic solid drawing methods: **g:Dcylinder()**, **g:Dcone()** and **g:Dfrustum()** now have two additional options: **edgestyle** and **edgewidth** (as for the **g:Dsphere()** method).

- In the module *luadraw\_compile\_tex*, for the methods:

**g:Dcompiled\_tex(L, anchor, options)** and **g:Compiled\_tex2path3d(L, options)**,

add the option **pos** identical to the labels.

- In the *luadraw\_compile\_tex* module, three global variables are added to manage access to the *pdflatex*, *pdf2ps* and *pstoedit* programs. These variables are **pdflatexcmd**, **pdf2pscmd** and **pstoeditcmd**, they allow you to optionally add a path to the program, for example: **pstoeditcmd** = `"/usr/bin/pstoedit"`.
- New syntax for the function **circle(data, nbdots)**, where *data* is a list (center and radius, or three points on the circle) and *nbdots* is the desired number of points. The old syntaxes remain valid.
- New syntax for the function **ellipse(data, nbdots)**, where *data* is a list: {center, rx, ry, incline}, and *nbdots* is the desired number of points. The old syntaxes remain valid.
- Added the function **mixpalette(pal, percent, color)** which returns a new palette after mixing each color of the palette *pal* with *color*.
- Correction of numerous typos in the documentation.
- Bug fixes...

## 5) Version 2.6

Non-exhaustive list:

- Added the *luadraw\_povray* extension which allows you to create an image with Pov-Ray and include it in the current graphic to draw over it.
- Added the *luadraw\_fields* extension which allows drawing vector fields or gradient fields.
- Added the *luadraw\_shadedforms* extension, which allows drawing 2D polygonal lines or filling a shape with a color gradient based on the chosen calculation method and palette.
- The **g:Dshadedpolyline()** method is now part of the *luadraw\_shadedforms* extension.
- Added the methods **g:Dimage()** and **g:Dmapimage()**. The first allows you to include an image in the graph, and the second allows you to map an image onto a parallelogram.
- Added the function **parallelogram()**, which returns the vertices of a parallelogram constructed from a vertex and two vectors. Also added is the corresponding drawing method **g:Dparallelogram()**.
- Added the options **gradside** and **gradsection** which allow modification of gradient parameters in the drawing of cylinders, cones and truncated cones (methods **g:Dcylinder()**, **g:Dcone()** and **g:Dfrustum()**).
- Extension of the method **g:Newcolor(name, color)**, the second argument can now be either a table of three RGB components, or a string representing a color.
- Bug fixes...

## 6) Version 2.5

Non-exhaustive list:

- Added the function **read\_csv\_file()**<sup>4</sup> which allows reading a *csv* file with different options.
- The *luadraw\_palettes* extension has been updated to version 1.3.0 of the @prism project of **Christophe BAL**.
- Added the method **g:Dshadedpolyline()** which allows drawing a 2D polygonal line with a color gradient depending on the calculation method and palette chosen.

---

<sup>4</sup>Based on an idea by Christophe BAL.

- Added the **g:Dpolynames()** method which allows displaying a polyhedron with the number of faces and/or those of vertices.
- Added the *luadraw\_cvx\_polyhedra\_nets* extension, which allows you to determine a net for convex polyhedra.
- Bug fixes...

## 7) Version 2.4

Non-exhaustive list:

- Added the central projection.
- Added the **legendstyle** option for axes, to impose a label style ("auto", "N", "E", ...) for legends when there are any (until now, the style was necessarily "auto").
- Added the **g:Labeldir()** method which allows global management of the writing direction.
- Added the functions **interCS()** (intersection between a circle in space and a sphere), and the function **interSSS()** (intersection between 3 spheres).
- Added the function **voronoi()** as a complement to Delaunay triangulation, it allows you to make Voronoi diagrams.
- Added the function **parallel\_polyline()** which returns a parallel polygonal line.
- Added the function **tangent\_from()** and the method **g:Dtangent\_from()** which allows drawing the tangents to a given curve from a given point.
- Bug fixes...

## 8) Version 2.3

Non-exhaustive list:

- Added cavalier perspective projections: on *yz*, on *xz* or on *xy*, as well as the isometric projection.
- Added the function **section2tube()**.
- Added the *luadraw\_compile\_tex* module.
- Added the **g:Proj3dV()** method for calculating the projection of space vectors onto the screen plane.
- Added the functions **circumcircle()** and **incircle()** in 2D, they return a sequence: center and radius.
- Added the function **line2strip()** which returns a path representing a "strip" centered on a given polygonal line.
- Added the function **delaunay()** which performs a Delaunay triangulation on a list of points and returns the list of triangles obtained.
- Added the function **cpx.normalize(z)** which returns the complex number *z* divided by its modulus (or **nil** if it is zero).
- Added the instruction **whatis(variable, msg)** which displays the status of a *variable* (along with the message *msg*) and its contents in the terminal.
- Bug fixes...



## 9) Version 2.2

Non-exhaustive list:

- Added the `clip` option for the methods: `g:Dfacet()`, `Dmixfacet()`, `g:addFacet()`, `g:addPoly()` and `g:addPolyline()`, as well as for point cloud drawing methods, and line drawing methods such as `g:Dpolyline3d()`, `g:Dparametric3d()`, `g:Dpath3d()`, etc.
- Added the `xyzstep` option for the `g:Dboxaxes3d()` method. This option defines a common step for all three axes (1 by default).
- Added the `g:DSdots()`, `g:DSstars()`, `g:DSinvstereo_curve()`, and `g:DSinvstereo_polyline()` methods to the *luadraw\_spherical* module.
- Added the *luadraw\_palettes* module.
- Added the `interDC()` function (intersection between a line and a circle in 2D) and the `interCC()` function (intersection between two circles in 2D).
- Added the `curvilinear_param()` and `curvilinear_param3d()` functions, which allow you to parameterize a list of points (one in 2D and the other in 3D) with a function of a variable  $t$  between 0 and 1.
- Added the function `cvx_hull2d()`, which returns the convex hull (polygonal line) of a list of points in 2D, and the function `cvx_hull3d()`, which returns the convex hull (list of facets) of a list of points in 3D.
- Added the methods `g:Beginclip(path)` and `g:Endclip()`, which make it easier to set up clipping using TikZ.
- Added the functions `normal()`, `normalC()`, and `normalI()`, which return the normal to a 2D curve at a given point. The corresponding graphics methods have also been added.
- Added the function `isobar()`, which returns the isobarycenter of a list of complexes.
- Added the `usepalette={palette,mode}` option for the `g:Dpoly()`, `g:Dfacet()`, `g:Dmixfacet()`, and `g:addFacet()` methods.
- Added the `clipplane()` function, which allows you to clip a plane with a convex polyhedron. The function returns the section, if it exists, as a facet.
- Added the `cartesian3d()` and `cylindrical_surface()` functions, which calculate and return surfaces, with the option to add dividing walls for the `g:Dscene3d()` method.
- Added the function `evalf(f, ...)` which allows a protected evaluation of  $f(\dots)$ . It returns the result of the evaluation if there is no runtime error from Lua, otherwise it returns `nil` but without causing the script execution to terminate.
- Added the function `split_points_by_visibility()` (3d) to separate a curve into two parts: visible part, hidden part.
- In the methods `g:Dfacet()`, `g:Dmixfacet()`, `g:Dpoly()`, `g:Dedges()`, `g:addFacet()`, `g:addPolyline()`, `g:addPoly()`, the default values for the line drawing options (thickness, color, and style) are the current values.
- Bug fixes...

## 10) Version 2.1

Non-exhaustive list:

- By default, TikZ files are saved in a subfolder called *\_luadraw*. The new package option `cachedir` allows you to change this.
- The `line join=round` option is automatically added to the *tikzpicture* environment.
- Two additional options for the *luadraw* environment: `bbox` and `pictureoptions`.
- A number of additional 2D and 3D geometric construction functions.



- Graduated axes (2D, 3D) use the *siunitx* package to format labels when the global variable `siunitx` is set to `true`.
- Added upright and slanted truncated cones (`frustum()` and `g:Dfrustum()`).
- Added regular pyramids (`regular_pyramid()`) and truncated pyramids `truncated_pyramid()`.
- Cylinders and cones are no longer necessarily upright; they can now be slanted.
- Added the `cutpolyline(L, D, close)` function.
- (Elementary) drawing of sets (`set()` function) and operations on sets (`cap()`, `cup()`, `setminus()`).
- Modification of the `mode` option of the `g:Dplane()` method.
- Addition of the `close` option for the `g:addPolyline()` method.
- Bug fixes...

## 11) Version 2.0

- Introduction of the *luadraw\_graph3d.lua* module for 3D drawings.
- Introduction of the `dir` option for the `g:Dlabel()` method.
- Minor changes in color management.

## 12) Version 1.0

First version.