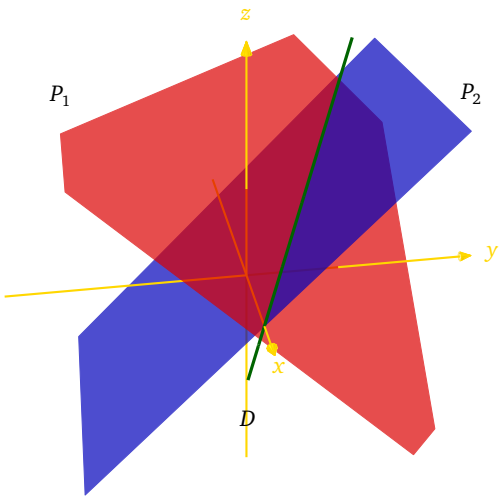
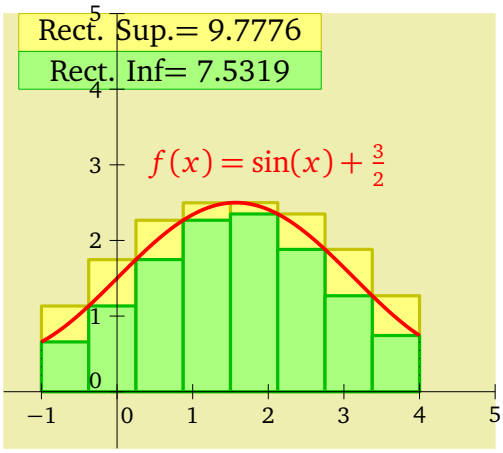
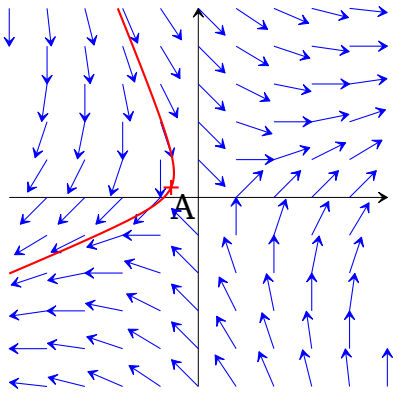
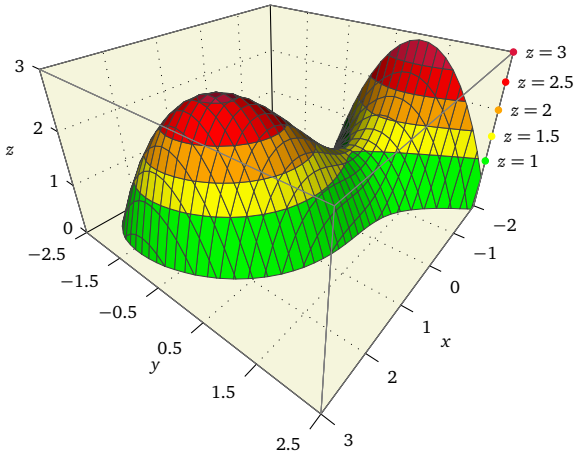
TEXGRAPH 1.974

Help









$z = -0.3x^3 - xy - y^2 + x + 1.5$

Patrick FRADIN

October 19, 2013

Contents

List of Figures

Chapter I

Introduction to TeXgraph

1) First overview

- TeXgraph is a program for creating graphics for maths (ie: drawing curves, surfaces, building geometric figures...) and exporting as text file using formats: LaTeX (eepic macros), PsTricks, Pgf/Tikz (macros pgf), Eps, Psf (eps+Psfrag), pdf (eps -> pdf conversion), svg... There are also specific exports dedicated to 3D scenes.

- It has been written for Windows and Linux.

- TeXgraph version 1.974 is distributed under the GPL (General Public Licence) terms.

  This release was written using Free Pascal with Lazarus (0.9.31 Svn).

  This program is free, you are allowed to redistribute and/or modify it under the terms of the GNU General Public Licence published by the Free Software Foundation (version 2 or later).

  This program is distributed because potentially useful, but WITHOUT ANY GUARANTEE neither explicit nor implicit, including selling guaranties or adaptating guaranties in a specific aim. Please report to the GNU General Public Licence for further details.

  You must have recieved a copy of the GNU General Public Licence with this program; otherwise, ask for one and write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, United States.

- You may help the author and the community by letting us know the bugs you encounter:

  By Email texgraph@tuxfamily.org.

  or mail to :

  Patrick FRADIN

  La Jauvigère

  17, Impasse du Vieux Château

  16590 BRIE

  FRANCE

- The TeXgraph program can be downloaded at `http://texgraph.tuxfamily.org/`

  You may find examples there and on the site :`http://melusine.eu.org/syracuse/`.(in french)

- A forum is available (in french at the moment) at :`http://texgraph.tuxfamily.org/forum/`

2) Launching TeXgraph

The program needs to be installed (see the *LisezMoi.txt* file), the executable is called *TeXgraph*, and is launched with a script : **startTeXgraph**.
The installation directory is containing the *TeXgraph* directory where you'll find the executables and other directories : *Exemples*, *doc* and *macros*.
TeXgraph is handling three kinds of files : the source files (**.teg*), the model files (**.mod*) and the macro files (**.mac*).

- The **.teg* files: these are the ordinary source files. You have them as you save a graphic made with the program.

- The **.mod* files: these are the model files that can be loaded, we can consider those as "ready-to-use" source file that can be completed.

- The **.mac*: these are the macro files and are to be loaded. Those may also include variable declarations. On the contrary to the first two file types, **all the code within a **.mac* file is considered as predefined** and won't be saved with the graphic (but the source file will contain a command to load that macros file).

These three file types follow the same syntax rules, as described in the *src4latex* (p. 159) section.

At program launch, several macro files are loaded: *TeXgraph.mac*, *couleurs.mac*, *scene3d.mac* and *interface.mac* (the later is only loaded by the GUI). Those files are considered as predefined and will stay in memory during the whole session until the program is closed.

It's also possible to load one or several other macro files at program launch by adding those as parameters in the command line . On the same way, the macros loaded are considered as predefined and will be removed from memory at the end of the session, ie: the closing of the program.

A macro file is loaded using the *File/Load Macros*. The variables and macros loaded are also considrered as predefined and won't be part of the graphics. **Though, these will be removed from memory at the next graphic file loading** . The variables and macro loaded with the *Filer/Import a model* are added to the current file, will be saved with it, and **will be removed at the next graphic file loading**

For a satisfying user experience with TeXgraph, your system should come with:

1. A working TEX distribution, including the *tikz/pgf*, *pstricks* packages.

2. The ImageMagic software suite is mandatory to convert images (*Snapshot* button or animated gifs from the *Animation.mod* model).

3. The swftools if you use the *Animation.mod* model and export using Flash format.

4. The pstoedit program is needed to convert compiled TEX formulas into paths.

5. The povray program if you use the *povray.mod* model.

If you also intend to use the *geom* and *jvx* 3D exports, you'll need to visualize the results using:

1. The geomview program : with the **.geom* files.

2. The javaview program : with the **.jvx* files. This program can be run locally, and as an applet in a web page as we can see on that page.

These programs are all free software available for linux and windows.

3) Graphic compositing

A TeXgraph graphic is made of:

- *Parameters* (p. 10): like graphic window coordinates, axes scales, margins....
- *Global Variables* (p. 36): These are mostly a list of complex numbers, eventually the *Nil* value.
- *Macros* (p. 37): useful to simplify the graphic compositing.
- *Graphic Elements* (p. 13): like axes, curves,...

4) Parameters

These correspond to the *Preferences* menu item, with the options:

- **Window**: to specify the working rectangular area where the drawing is done. Here, the Xmin, Xmax, Ymin, Ymax constants are set, then the two axes scale : Xscale,Yscale (centimeters). The constants can be reused in commands, but not directly modified, unless you use the *Fenetre* (p. 44) command. There is an orthonormal frame when Xscale=Yscale.

- **Margins**: to set margins around the graphic in case of text overflow using labels for example. The constant's value are set: margeG, margeD, margeH, margeB (centimeters). The constants can be reused in commands, but not directly modified, unless you use the *Marges* (p. 49) command.
- **Export the border**: In case of this option is selected, there will be a framework around the drawing at exportation, and on the screen. The frame is a plain black line englobing the margins. That option can be modified using the *Border* (p. 39) command.
- **Export colors**: In case of this option is not selected, the graphic will be exported with gray scale colors.
- **Export names**: In case of this option is selected, the name of each graphic element will be added in the exported file right before the drawing of the element (as a remark). This makes easyier to find LaTeX, pgf or pstricks code in the exportations and modify it if necessary.
- **Print global variables**: In case of this option is selected, the global variables are printed on screen (not exported). This can be useful while preparing a drawing.
- It's also possible to hide the right and/or left column of the graphic interface, and to show/hide the label's anchor.

5) Colors

5.1 Predefined colors

The predefined color list can be found on the page : couleurs.html.

5.2 Commands and macros linked with colors

- **Lcolor(<color> [, gray scale])**: macro that shows the three components red, green, blue of the given color in a list-form [r,g,b]. The second argument is not mandatory, is 0 by default, if set to 1, the color is converted into gray scale before computing the components.
- **Bcolor(<color>)**: macro that shows the blue level of the given color.
- **Gcolor(<color>)**: macro that shows the green level of the given color.
- **Rcolor(<color>)**: macro that shows the red level of the given color.
- **Cplcolor(<color>)**: macro that shows the complementary color level of the given color.
- **Dark(<color>, <factor>)**: macro that is doing a barycenter between the color and the black, the factor is a number in the interval [0;1] and is the proportion of black color (1=100%).
- **Light(<color>, <factor>)**: macro that is doing a barycenter between the color and the white, the factor is a number in the interval [O;1] and is the proportion of white (1=100%).
- **GrayScale(<0/1>)**: that command is described *here* (p. 46). It is used to activate/desactivate the color conversion into gray scale.
- **HexaColor(<Hexa value>)**: The command is described *here* (p. 46). Example: Color:=HexaColor("F5F5DC").
- **MixColor(<color1>, <proportion1>, <color2>, <proportion2>, ..., <colorN>, <proportionN>)**: macro that returns the color (rgb) obtained after combining the given colors and its corresponding proportions.
- **Palette(<[Color1, Color2, ..., ColorN]>, <factor in [0;1]>)**: returns one color from the given palette in function of the factor, 0 for the first color, and 1 for the last.
- **Hsb(<hue (0..360)>, <saturation (0..1)>, <brightness (0..1)>)**: macro that returns a color from its components hue, saturation and brightness. Example: Color:=Hsb(60,1,1).
- **HueColor(<color>)**: returns the hue color's component.
- **SatColor(<couleur>)**: returns the saturation color's component.
- **BrightColor(<couleur>)**: returns the brightness color's component.

- **ColorJump(<color>)**: macro that returns the constant *jump* and the <*color*> as the imaginary part. The *Ligne* (p. 90) command read the value and uses it as the fill color if the *Fillstyle* variable is empty (value *none*).
- **Rgb(<red (0..1)>, <green (0..1)>, <blue (0..1)>)**: the command is described *here* (p. 55). Example: Color:= Rgb(0.5, 1, 0.6).
- **RgbL(<[red, green, blue]>)**: The macro has the same effect as *Rgb*, apart from the fact that the three component are in a list.
- **Ryb(<red (0..1)>, <yellow (0..1)>, <blue (0..1)>)**: macro that returns a color from its components red, yellow, blue. Example: Color:= Ryb(0.5, 0.8, 0.6).
- **Rgb2Hsb(<couleur>)**: macro for converting a color (rgb) into an Hsb color, ie a list [hue, saturation, brightness].
- **Rgb2Hexa(<color Rgb>)**: returns the string representing the Hexadecimal color code, eg: "FF0000" for the red color.
- **Rgb2Gray(<color Rgb>)**: returns the gray scale color (rgb form).

Example(s): Each facet is coloured according to the rating of the center of gravity. The color is added thanks to the *ColorJump* macro in the *jump* ending facet constant. The *Hsb* macro is made for continuously varying the color. Before drawing the surface, the facets are sorted using the *SortFacet* (p. 119) command, then are drawn.

```
\begin{texgraph}[name=ColorJump, file]
Graph image = [
view(-6.5,6,-6.5,5.5),
Marges(0,0,0,0),size(7.5),
view3D(-3,3,-3,3,-3,3),ModelView(central),
S:=GetSurface([u+i*v,2*sin(u)+cos(v)],
          -3+3*i,-3+3*i),
stock:=for facette in S By jump do
  z:=Zde(isobar3d(facette)),
  facette,
  ColorJump(Hsb(270*(Zsup-z)/(Zsup-Zinf),1,1))
    od,
FillStyle:=full, LabelSize:=footnotesize,
BoxAxes3D(grid:=1, FillColor:=lightblue),
Ligne3D(SortFacet(stock),1)
];
\end{texgraph}
```

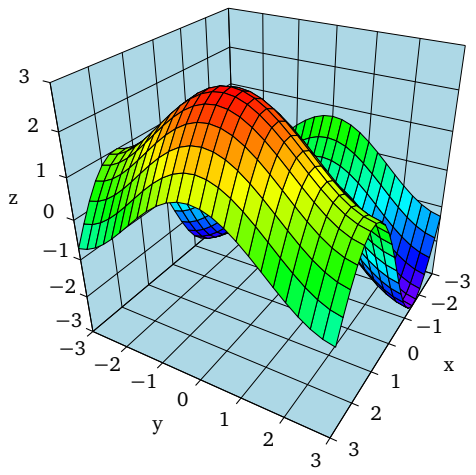


Figure 1: *Heat type colouring*

Chapter II

Graphic elements

A graphic is a stack of graphical elements [1], those can be created, modified and removed individually. The graphical elements are independants, appart eventually those created by the user.

Each graphical element is defined from a name (a name starts with a letter. The maximum length is 35 characters among 0..9, a..z, A..Z, ' and _) and a *command* (p. 27), added that each graphical element has attributes like: colors, line style, line thickness ...

There are some basic graphical elements. Those can be created using the menu, a shortcut or also using a graphical command in a *User*-defined element.

The predefined graphical elements are:

1) The grid

To draw a grid (there can be several ones).

- Shortcut: *Ctrl+G*
- The window is asking for the Origin's coordinate, the graduations units on Ox and Oy axes, that must be positive, if it's zero then the graduations aren't shown.
- There is no labels drawn with the grid. If needed, it is sufficient to create axes.
- Corresponding graphical command: *Grille* (p. 89) (to be used in a User-defined type element).

2) Axes

To draw orthogonal axes.

- Shortcut: *Ctrl+A*
- The window is asking for the Origin's coordinate, the graduations units on Ox and Oy axes, that must be positive, if it's zero then the graduations aren't shown.
- We can set two parameters (two global variables):
  - *xyticks*: it's the graduation's length (in centimeters) on the axes.
  - *xylabelsep*: it's the distance (in cm) between the labels and the ticks.
- Corresponding graphical command: *Axes* (p. 85) (to be used in a User-defined type element).

[1] The drawing is ordered, and can be modified with "drag and drop" using the mouse.

3) Curves

To draw a plane curve: cartesian, polar, parametric.

- Shortcuts: Parametric curve: *Ctrl+P*, Polar curve: *Ctrl+O*, Cartesian curve: *Ctrl+R*.
- We must give a name.
- Then:
  - For a cartesian curve $y = f(x)$, give the expression of f(x).
  - For a polar curve $r = f(t)$, give the expression of f(t).
  - For a parametric curve $(x(t), y(t))$, give the expression of the function $f(t) = x(t) + i * y(t)$.
- We can set two curve's parameters:
  - Division(s): It's a positive integer or zero that shows how many times TeXgraph can cut into two pieces (dichotomy) the interval between two consecutives values of t (5 by default). This raises the number of points in the neighbourhood of brutal variations.
  - Discontinuité: 0 or 1, if it's 1 and the distance between two consecutive points is above a certain level, then a discontinuity is inserted in the points list.
- The interval for the t parameter (global variables *tMin* and *tMax*), is also the interval for the x variable of the cartesian curves. That interval and the number of points (variable *NbPoints*) can be set using the Attributes button.
- Corresponding graphical commands *Cartesienne(f(x))*, *Polaire(r(t))* and for the parametric curves : *Courbe* (p. 87) (to be used in a User-defined type element).

4) Differential equation

Approximated solution (Runge-Kutta 4 method) of an equation of the kind: $x'(t) + iy'(t) = f(t, x, y)$ with initial condition $x(t_0) = x_0$ and $y(t_0) = y_0$:

- Shortcut: *Ctrl+E*
- We give a name.
- we give a command like [f(t,x,y), t0, x0+i*y0],
- we choose the representation mode: coordinates (x,y) or (t,x) or (t,y).
- Corresponding graphical command: *Equadif* (p. 89) (to be used in a User-defined type element).

5) Implicit function

Set of points of coordinates (x, y) so that $f(x, y) = 0$.

- Shortcut: *Ctrl+I*
- We give a name.
- we give a command like f(x,y), or [f(x,y),n,m], n stands for the subdivisions number of the Ox axe and m the subdivisions number of the Oy axe (50 by default). On each pavement obtained, a sign change is tested, if yes then a dichotomy is applied on the edges of the pavement.
- Corresponding graphical command: *Implicit* (p. 89) (to be used in a User-defined type element).

6) Bezier curve

Several BEZIER curves (with eventually some line segments)

- Shortcut: *Ctrl+B*
- We give a name.
- Then a command like [<list of points>]. The list of points can be:
  1. A list of three points [A,C,B]. The Bezier curve's origin is <*A*> and the other extremity is <*B*> with a control point C. It's the curve parametrized with $(1-t)^2A+2t(1-t)C+t^2B$.
  2. A 4 points list (or more): [A1,C1,C2,A2,C3,C4,A3...]: it's successive Bezier curves with 2 control points, the first one goes from A1 to A2, is controled by C1, C2 (parametrized with: $(1-t)^3A1+3(1-t)^2tC1+3(1-t)t^2C2+t^3A2$), the second one goes from A2 to A3 and the control points are C3,C4 ...etc. One exception though, the control points can be replaced with the *jump* constant. In that case, we jump directly from A1 to A2 with a line segment.
- The number of points computed (by curve) can be modified in the Attributes (variable *NbPoints*).
- Corresponding graphical command:*Bezier* (p. 86) (to be used in a User-defined type element).
- Example(s): [-2, -1+i, i, 1, jump, 1-i, jump, -2-i, jump, -2].

7) Cubic spline

Third degree curve passing through given points with or without restraint at the ends.

- Shortcut: *Ctrl+S*
- We give a name.
- Enter a command like [v0,A1,A2,...,An,v1]. The complex numbers v0 and v1 are the affixs of the tangent vectors at the ends (if zero : there is no restraint), and the A1,...,An complexes are affix's points interpolated by the curve.
- The total number of the calculated points can be modified in the Attributs with the variable *NbPoints*.
- Corresponding graphical command: *Spline* (p. 93) (to be used in a User-defined type element).

8) Straight line

Straight line in the plane defined by two points, one point and a direction vector, or a cartesian equation.

- Shortcut: *Ctrl+D*
- We give a name.
- We give a command like:
  - [A,B] for a line passing through the point's affixes A and B.
  - [A,A+v] for a line passing through the point A and directed by the vector v.(A and v are complex numbers).
  - [a,b,c] for a line whose cartesian equation is ax+by=c.
- It is possible to deterrmine the intersection of two lines with Inter operator. For example, if A,B,C,D are four point's affixes, then executing [A,B] Inter [C,D] will give the affix of the intersection point of (AB) and (CD) if they are secants, *Nil* if not.
- Corresponding graphical command:*Droite* (p. 87) (to be used in a User-defined type element).

9) Point(s)

To draw one or several points.

- Shortcut: *Alt+P*
- We give a name.
- We enter a command like: [A1,...,An]. A1,...,An are the point's affixes.
- The value of *DotStyle* can be set in the Attributes. So are the predefined variables :*DotScale*, *DotAngle*, *DotSize*.
- Correponding graphical command: *Point* (p. 92) (to be used in a User-defined type element).

10) Polyline

To draw an open or closed polyline (list of points) with one or several connex components.

- Shortcut: *Ctrl+L*
- We give a name.
- We enter a command like: [A1,...,An]. A1,...,An are the point's affixes that build the line. In case of several connex components, we separate them with the *jump* constant, for example: [A1,A2,A3,jump,A4,A5,A6].
- We can set two line parameters:
  - A radius (>0) for rounded angles (arcs of the desired radius).
  - A boolean to set if the line is open or closed (0=open).
- The line style can be set in the Attributes and so are the thickness, the fill mode and the colors. It is also possible to add arrows at the ends.
- Some of the graphical elements are build from a list of points (curves, differential equations,...). It is possible to get the list with the function *Get* (p. 44). For example, if you created a spline called *S*1, you can get all the points of that curve and store the list in a *A* variable with : A:=Get(S1).
- It is possible to determine the intersection of two polyline with the InterL operator. For example, the execution of Get(Courbe(t+i*t^2)) InterL Get(Droite(0,1+i)) returns:

  [0,0.999368819693+0.999368819693*i].

- Corresponding graphical command:*Ligne* (p. 90) (to be used in a User-defined type element).

11) Path

To draw an open or closed path.

- Shortcut: *Ctrl+H*
- We give a name.
- We enter a command as a list of points (affixes) and instructions that indicate what the points correspond to. These instructions are:
  - **line**: link the points with a polyline,
  - **linearc**: link the points with a polyline but the angles are rounded with an arc. The value preceeding the linearc command is interpreted as the arc's radius.
  - **arc**: draw an arc of circle. It needs four arguments: 3 points and the radius, plus eventually a fifth argument: (+/-1). 1 (default) for counterclockwise.

- **ellipticArc**: draw an arc of ellipse. That needs five arguments: 3 points, the Xradius, the Yradius, and eventually a sixth argument: the direction angle (degrees) of the great axis with the horizontal axis.
- **curve**: link the points with a natural cubic spline.
- **bezier**: link the first and the fourth point with a Bézier curve (the second and third points are the control points).
- **circle**: draw a circle. Needs two arguments: one point and the center, or three arguments that are three points of the circle.
- **ellipse**: draw an ellipse, the arguments are: one point, the center, rX radius, ry radius, great axis direction in degrees (optional).
- **move**: a move without drawind anything.
- **closepath**: close the current component.

By convention, the first argument of the part number n+1 is the last point of the part number n.

- Example(s): Path([-3+2*i,-3,-2,line,0,2,2,-1,arc,3,3+3*i,0.5,linearc,1,-1+5*i,-3+2*i,bezier,closepath])
- Corresponding graphical command: *Path* (p. 91) (to be used in a User-defined type element).

12) Ellipse

To draw an ellipse defined with its center and two radius rx, ry and its direction with the horizontal axis.

- Shortcut: *Ctrl+C*
- We give a name.
- We enter a command like: [A, rx, ry] or [A, rx, ry, direction in degrees]: A is the center's affix, rx and ry the radius. The direction with the horizontal axis is by default set to zero.
- If the system is not an orthonormal coordinate system, the ellipse is deformed. The system is orthonormal when the variables *Xscale* and *Yscale* are equal: see the option Preferences/window of the menu. To get a straight ellipse with a non orthonormal system, use the macro *Rellipse()*.
- There can't be an arrow an an ellipse. (draw an elliptical arc instead).
- Corresponding graphical command: *Ellipse* (p. 88) (to be used in a User-defined type element).

13) Elliptical arc

To draw an arc of an ellipse defined with three points B, A, C (defining an oriented angle), two radius rx, ry and a direction.

- Shortcut: *Alt+Maj+A*
- We give a name.
- We enter a commad like [B, A, C, rx, ry]: A is the center's affix, the starting point of the arc is on the half-line $[AB)$, the last point on the half-line $[A, C)$. the boolean "counterclockwise" is used to set the turning direction.
- If the coordinate system is not orthonormal, the arc will be deformed. The system is orthonormal as soon as the variable *Xscale* and *Yscale* are equal: see option Preferences/window in the menu. To get a non deformed arc in a non orthonormal system, use the macro : *RellipticArc()*.
- Arrows can be set at the ends of an arc.
- Corresponding graphical command: *EllipticArc* (p. 88) (to be used in a User-defined type element).

14) Label

To print some text in the graphic.

- Shortcut: *Alt+L*
- We give a name.
- We choose an affix (reference point for the label).
- We enter the plain text (without "'s).
- We enter the label style (variable *LabelStyle*) in the Attributes, and the size (variable *LabelSize*) and direction (*LabelAngle*).
- Labels can be written with maths formulas and TEX macros that will be compiled at exports except: eps, pdf and svg (see the section *Exportations* (p. 19)).
- Corresponding graphical command:*Label* (p. 90) (to be used in a User-defined type element).

15) User-defined

That element gives the possibility to the user to create its own graphical element. It will be considered as a whole entity.

- Shortcut: *Ctrl+U*
- We give a name.
- We enter a command. It can use some graphical commands (straights, curves, ...) or graphical macros (macros that have a graphical effect) like those in the file TeXgraph.mac.
- Examples:
  - Here is a command of a User-defined graphical element:

    [Courbe(t+i*sin(t)), Arrows:=2, tangente(sin(t), pi/3,2)]

    It draws the sinus curve with usual parameters. We set the global variable Arrows to 2 (number of arrows), then we draw a piece of the tangent [2] to the sinus curve at $\pi/3$, length 2 (graphical units).
  - Other example:

    for m in [-1,-0.25,0.5,2] do Color:=Rgb(Rand(),Rand(),Rand()), Courbe(t+i*t^m) od

    It draws cartesian curves family: $t \mapsto t^m$ with m in the list $[-1, -0.25, 0.5, 2]$. For each value of m the color of the drawing is changing.
- Corresponding command: *NewGraph* (p. 51).

[2] tangente is a graphical macro from the file TeXgraph.mac

Chapter III

Graphics Exports

The graphics created with TeXgraph can be saved as source files (*.teg) and/or exported as files dedicated to be included in a (La)TeX document. Pay attention to the fact that (La)TeX has to be able to locate those files at compilation : put them in the same directory as your document or give the whole path of the file in the document. There are several export formats:

1) TeX format

- The export's extension is *.tex*. It uses several packages: xcolor, epic and eepic (line drawing) and eventually rotating (rotating the labels, only visible in the postcript output of the document). Those packages are quite poor in graphical capabilities: no solid filling, no transparency....so that kind of export should be only used with basic graphic drawings. For more complex graphics, it is better to choose the pgf/tikz, pstricks, eps or pdf formats.
- Example (minimal):

```
\documentclass{article}
\usepackage{xcolor,rotating,epic,eepic}
\begin{document}
  \input{Mongraph.tex}
\end{document}
```

- Possible compilations:
  - latex
  - LaTex + dvips
  - latex + dvips + ps2pdf
  - latex + dvipdfmx (or dvipdfm)

2) pst format

- Those file are exported as *.pst*. The needed macros are those from the pstricks package (version 1.27 minimum)
- Example (minimal):

```
\documentclass{article}
\usepackage{pstricks}
\begin{document}
  \input{Mongraph.pst}
\end{document}
```

- Possible compilations:
  - LaTex + dvips
  - latex + dvips + ps2pdf

3) pgf format

- The file is exported with the *.pgf* extension. Package needed : pgf (version 2.0 minimum)
- Example (minimal):

```
\documentclass{article}
\usepackage{pgf}
\begin{document}
  \input{Mongraph.pgf}
\end{document}
```

- Possible compilations:
  - pdflatex
  - LaTex + dvips
  - latex + dvips + ps2pdf
  - latex + dvipdfmx (or dvipdfm), in that case, it is mandatory to add:

    ```
    \def\pgfsysdriver{pgfsys-dvipdfm.def}
    ```

    right before declaring the pgf package.

4) tkz format

- The files are exported with the *.tkz* extension. Macros from the pgf (version 2 minimum) are used in an *tikzpicture* environment therefore tikz macros can be also added.
- Example (minimal):

```
\documentclass{article}
\usepackage{tikz}
\begin{document}
  \input{Mongraph.tkz}
\end{document}
```

- Possible compilations:
  - pdflatex
  - LaTex + dvips
  - latex + dvips + ps2pdf
  - latex + dvipdfmx (or dvipdfm), In that case, it is mandatory to add:

    ```
    \def\pgfsysdriver{pgfsys-dvipdfm.def}
    ```

    right before declaring the tikz package.

5) eps format

- The files are exported with the *.eps* extension and are using postscript language. In that format, labels won't be compiled with TeX, therefore TeX macros or maths formulae will be printed but not interpreted.
- Example (minimal)

```
\documentclass{article}
\usepackage{graphicx}
\begin{document}
  \includegraphics{MonGraph.eps}%extension is optional
\end{document}
```

- Possible compilations:
  - LaTex + dvips
  - latex + dvips + ps2pdf
  - latex + dvipdfmx (or dvipdfm), in that case your system must be configured so that dvipdfmx can convert eps image into pdf image on-the-fly (with epstopdf).

6) psf (eps+psfrag) format

- The files are exported with the *.psf* extension. There are two generated file, one eps file and one psf file. The first comes with the postscript version of the graphic without the labels, and the second comes with the labels that the psfrag package will replace in the graphic after a (La)TeX compilation. In that format, the maths formulae and the TeX macros will be compiled. The last line of the psf file is:

```
\includegraphics{<nom>.eps}
```

- Example (minimal):

```
\documentclass{article}
\usepackage{pstricks,psfrag,graphicx}
\begin{document}
  \input{MonGraph.psf}
\end{document}
```

- Possible compilations:
  - LaTex + dvips
  - latex + dvips + ps2pdf

Rem: in that format, some labels use pstricks macros.

7) pdf format

- The files are exported with the *.pdf* extension. There are two generated files: TeXgraph create one eps file then call a tool to convert eps to pdf, *epstopdf* by default. It is possible to change the tool by editing the macro file TeXgraph.mac and modifying the *pdfprog* macro. In that format, the labels won't be compiled. so, labels with maths formulae and TeX macros will be printed but not interpreted.
- Example (minimal):

```
\documentclass{article}
\usepackage{graphicx}
\begin{document}
  \includegraphics{MonGraph.pdf}
\end{document}
```

- Possible compilations:
  - pdflatex

8) Compiled formats

8.1 epsc format

When exporting to that format, the program is asking for a filename for the eps file that is created, let us call it *Toto.eps*. The graphic is then exported using pstricks format in a file called *file.pst* in the TeXgraph's " Temp" directory and the script `./CompileEps.sh` under linux and `CompileEps.bat` under windows is launched with the filename `Toto` as an argument.

The linux script is like the following (similar to the windows version):

```
#!/bin/sh
latex -interaction=nonstopmode CompileEps.tex
dvips -E -o $1.eps CompileEps.dvi
```

The script is launching the compilation of the following file *CompileEps.tex*:

```
\documentclass[11pt]{article}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{lmodern}
\usepackage{pstricks-add,pst-eps,amssymb,amsmath}
\usepackage[dvips,margin=0cm,a4paper]{geometry}
\pagestyle{empty}

\begin{document}
           \TeXtoEPS%
           \input{file.pst}%
           \endTeXtoEPS%
\end{document}
```

Its conversion to an eps image is done via the *dvips* program. Of course, you can modify that file by removing the copy that is in the (`$HOME/.TeXgraph` for linux et `c:\tmp` for windows) TeXgraph-temp directory and modify the original in the TeXgraph directory.

8.2 pdfc format

When exporting to that format, the program is asking for a name of the pdf file that is created, let us call it *Toto.pdf*. The graphic is then exported using pgf format in a file called *file.pgf* in the TeXgraph temp directory. The script `CompilePdf.sh` under linux et `CompilePdf.bat` under windows is then launched with two arguments: The number 1, followed by the filename `Toto`.

The linux script is like the following (similar to the windows version):

```
#!/bin/sh
cat > CompilePdf.tex <<EOF
           \documentclass[11pt,frenchb]{article}
           \usepackage[utf8]{inputenc}
           \usepackage[upright]{fourier}
           \usepackage{pgf,amssymb,amsmath,amsfonts,babel}
           \usepackage[a4paper,margin=0cm,pdftex]{geometry}
           \usepackage[active,tightpage]{preview}
           \pagestyle{empty}
           \begin{document}
                      \newcounter{compt}
                      \setcounter{compt}{1}
                      \loop
                      \begin{preview}
                      \input{frame\thecompt.pgf}%
                      \end{preview}
                      \ifnum \thecompt<$1\addtocounter{compt}{1}
                      \repeat
           \end{document}
EOF
pdflatex -interaction=nonstopmode CompilePdf.tex
cp -f CompilePdf.pdf $2.pdf
```

That script is creating the file *CompilePdf.tex* as we can read in the script. The file is put in the temp directory, then the compilation is launched with pdflatex. The value 1 means that there is only one image to create. (the same script is used to create animated graphics)

9) svg format

It is a vector format to be used in web pages. The exported file is a text xml file that can be included in an html page like the following:

```
<object type="image/svg+xml" data="source.svg" width="450" height="450">
</object>
```

Warning ! All html readers are not necessary able to print svg parts natively. Try Firefox !

10) Summary

Export	package(s)	Compilation(s)	code	Labels TEX interpreted
tex	epic, eepic, xcolor, rotating	LATEX LATEX+dvips LATEX+dvips+ps2pdf LATEX+dvipdfm(x)	TEX	X
pst	pstricks ou pstricks-add	LATEX+dvips LATEX+dvips+ps2pdf	pstricks	X
pgf	pgf	pdflatex LATEX LATEX+dvips LATEX+dvips+ps2pdf LATEX+dvipdfm(x)	pgf	X
tkz	tikz	pdflatex LATEX LATEX+dvips LATEX+dvips+ps2pdf LATEX+dvipdfm(x)	tikz/pgf	X
eps	graphicx	LATEX+dvips LATEX+dvips+ps2pdf LATEX+dvipdfm(x)	postscript	
psf	pstricks, psfrag, graphicx	LATEX+dvips LATEX+dvips+ps2pdf	postscript	X
epsc	graphicx	LATEX+dvips LATEX+dvips+ps2pdf LATEX+dvipdfm(x)	pstricks	X
pdf	graphicx	pdflatex	postscript	
pdfc	graphicx	pdflatex	pgf	X
svg	none	non recognized format	xml	

11) Export to the clipboard

There is a button in the toolbar to copy the current graphic to the clipboard. The graphic is copied in a text format like a text file. It is possible to copy the graphic using the formats:

- **tex**, **pgf**, **tkz**, **pst**: Then we can copy the graphic directly in a (La)TeX document without loading any file with the *input* macro.
- **teg**: it is the TeXgraph's source file format .
- **src4latex**: it is the TeXgraph's source file format but in an environment to be directly included in a LATEXdocument. That format is described in *that section* (p. 159).
- **texsrc**: this is the source file written in colors in the TEX language. Useful to display coloured examples in LATEX documents like the one you are now reading.

12) Preview

Click on that button (the eye). The *Apercu* macro from the file *interface.mac* will be executed. The command defining that macro is:

```
[Export(pgf,[TmpPath,"file.pgf"]),
 Exec("pdflatex", ["-interaction=nonstopmode apercu.tex"],TmpPath,1),
 Exec(PdfReader,"apercu.pdf",TmpPath,0)
]
```

The current graphic is exported using pdf format in the file *file.pgf*, in the TeXgraph's temp directory. Then we launch the comilation of the *apercu.tex* file using pdflatex, and finally the created file is opened: *apercu.pdf* with the pdf reader (defined in the configuration file, option Preferences/config file). The *apercu.tex* file looks like:

```
\documentclass[a4paper,12pt]{article}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{lmodern}
\usepackage{pgf,amssymb,amsmath}
\usepackage{mathrsfs}
\usepackage[margin=1cm,pdftex]{geometry}
\pagestyle{empty}

\begin{document}
            \begin{figure}
            \centering
            \input{file.pgf}%
            \end{figure}
\end{document}
```

That file can, of course, be modified. First you have to remove the copy that can be found in TeXgraph's temp directory (`$HOME/.TeXgraph` under linux et `c:\tmp` under windows), then you can modify the original in the TeXgraph directory.

13) User-defined export

Via the *MyExport* command (or *draw*, an alias) it is possible to create new graphical elements with a personalized export, different from default's TeXgraph export.

- **MyExport(<"name">, <parameter 1>, ..., <parameter n>)**
- Description: The command is used like a graphical command. The user choose a <*"name"*> and must create two macros:
  - The first whose name must concatenate the word *Draw* and the <*"name"*>. That macro is creating the drawing.
  - The second one whose name must concatenate the word *Export* and <*"nom"*>. That macro is creating the export writing in the export file using the command:*WriteFile* (p. 60).

  When the graphic is evaluated, the command *MyExport* calls the drawing macro "*Draw*"+"*name*" giving to it the parameters: <*parameter 1*>, . . . , <*parameter n*>.

  When exporting, the command *MyExport* calls the export macro "*Export*"+"*name*" giving to it the parameters:<*parameter 1*>, . . . , <*parameter n*>. If the macros returns 0, then TeXgraph is using the " classical" export.
- Example(s): exporting cartesian curves as pstricks format using the macro `\psplot`. Let us choose the name `pstcartesian`, we write then the drawing macro *Drawpstcartesian(f(x), [options])* with options:
  - clip := ⟨ 0/1 ⟩: to clip or not with the window defined with the option : *clipwin* (0 by default),
  - clipwin := ⟨ [xmin+i*ymin, xmax+i*ymax] ⟩: defining the default clipping window, that is the default graphical window,
  - x := ⟨ [xmin, xmax] ⟩: drawing interval for the function, [tMin, tMax] by default.

Those options must be *global variables*.

```
{Drawpstcartesian(f(x),[options])}
[SaveAttr(), clip:=0, clipwin:=[Xmin+i*Ymin, Xmax+i*Ymax], x:=[tMin,tMax],
$aux:=%2, {options evaluation}
tMin:=x[1], tMax:=x[2],
if clip then
        SaveWin(), $a:=clipwin[1], $b:=clipwin[2],
        Fenetre( Re(a)+i*Im(b), Re(b)+i*Im(a))
fi,
Cartesienne(%1,0),
if clip then RestoreWin() fi,
RestoreAttr() ]
```

Then we write the exporting macro:*Exportpstcartesian(f(x), [options])*

```
{Exportpstcartesian(expression,[options])}
if ExportMode=pst then {We test the exporting mode}
 SaveAttr(), clip:=0, clipwin:=[Xmin+i*Ymin, Xmax+i*Ymax], x:=[tMin,tMax],
 $aux:=%2, {Evaluation des options}
 tMin:=x[1], tMax:=x[2],
 WriteFile([if clip then
            $a:=clipwin[1], $b:=clipwin[2],
            "\psclip{",
            "\psframe[linestyle=none,fillstyle=none]",
            @coord(a),@coord(b),"}%",LF
            fi,
           "\psplot[algebraic",
           if NbPoints<>50 then ",plotpoints=",NbPoints fi,
           "]",
           "{",Round(tMin,6),"}{",Round(tMax,6),"}{", @cvfunction(String(%1)),"}",
          if clip then LF,"\endpsclip" fi
          ]),
 RestoreAttr()
else 0 { <- 0 means normal export}
fi
```

The macro *cvfunction* returns the function using pstricks format as a string:

```
{cvfunction( string ): converting to pstricks's syntax}
[$aux:=StrReplace(%1,"cos","COS"),
aux:=StrReplace(aux,"sin","SIN"),
aux:=StrReplace(aux,"tan","TAN"),
aux:=StrReplace(aux,"arccos","ACOS"),
aux:=StrReplace(aux,"arcsin","ASIN"),
aux:=StrReplace(aux,"arctan","ATAN"),
aux:=StrReplace(aux,"ch","COSH"),
aux:=StrReplace(aux,"sh","SINH"),
aux:=StrReplace(aux,"th","TANH"),
aux:=StrReplace(aux,"argch","ACOSH"),
aux:=StrReplace(aux,"argsh","ASINH"),
aux:=StrReplace(aux,"argth","ATANH"),
aux:=StrReplace(aux,"exp","EXP"),
aux]
```

If we then create a graphical element with the command: MyExport("pstcartesian", x^2, [x:=[-2,2], clip=1]), then the pstricks export will give the file:

```
\psset{xunit=1cm, yunit=1cm}
\begin{pspicture}(-5.5,-5.5)(5.5,5.5)%
%objet1  (Utilisateur)
\psclip{\psframe[linestyle=none,fillstyle=none](-5,-5)(5,5)}%
\psplot[algebraic]{-4}{4}{x^2*SIN(x)}
\endpsclip
\end{pspicture}%
```

NB: this example is not complete. Exporting the attributes is not handled: color, thickness, line style,...

Chapter IV

The TeXgraph language

1) TeXgraph commands

Commands are in fact true mathematical functions. Those return a result that can be a **list of complex numbers and/or strings** or *Nil* .

Some commands are allowing minimal programming : value assignment to variables, and control structures (alternative, loops).

1.1 General syntax

- The general syntax of a TeXgraph command is : [argument1, ..., argumentN], in case there is only one argument, hooks are optional (the hooks represent the function *Liste* (p. 48)). Each argument is a mathematical expression.
- Running the command consist in *evaluating each argument* and returning the *results's list* that differ from *Nil*.
- Example(s):
  - [2,1+i,sqrt(-2),"toto",1/2] returns the list: [2,1+i,"toto",0.5].
  - Seq(k^2,k,1,5) returns the list: [1,4,9,16,25].
  - Droite(0,1+i) returns the value *Nil*, but the function *Droite* (p. 87) has a graphical effect if used in a graphical *user* element.
  - Consider three global variables: A, B and C. Then the command [C,C+i*(B-A)] returns C's value followed by the value $C + i(B - A)$. That expression can be used to define the perpendicular to (AB) passing through C.
  - Supose we have to build a triangle (ABC) with its three medians as a single graphical element, then :
    - we choose *Graphical elements/Create/User*,
    - we give a name to the objet,
    - we enter the command:
      [Ligne([A,B,C],1), Droite(A,(B+C)/2), Droite(B,(A+C)/2), Droite(C,(A+B)/2)]
      The functions *Ligne* (p. 90) and *Droite* (p. 87) return the *Nil* value and has a graphical effect in the *user* context.
    - There are still three variables to create: A, B, C (if not already done). Of course, the polyline and the three straight lines can be separately created.
- Calculations on the positive numbers are theorytically in the interval $[10^{-324}, 10^{308}]$.
- TeXgraph is case sensitive.
- Each TeXgraph object is identified by an *identifier* (or name), that has to follow the rules:
  - Beginning with a letter.
  - Less than 35 caracters.
  - A caracter has to be among: a letter, a figure, a quote, or an underscore.

1.2 Control structures

Following structures had been introduced so that entering commands is made easier:

- the alternative: *if then else fi*,
- the Conditional loop: *while do od*,
- the repeat loop: *repeat until od*,
- and the iterating loop: *for do od*.

We also add that:

- The command *Set* (p. 55) (assignment) can be replaced with :=, eg: x:=2 instead of Set(x,2). The *Set* (p. 55) command is evaluating the first argument alphanumerically, that is to say that if k is a variable with the value 2, then the command Set(["x",k], 5) will be interpreted as : Set(x2,5). This can also be done with the assignment symbol: ["x",k]:= 5.
- If x is a variable containing a list, the command Copy(x, n, 1), that is returning the n-th element of the list, can be replaced by : x[n]. In general: x[start, number] with the convention that if number=0 then we go until the end of the list, and if start=−1 then we start from the end of the list backwards.

  NB: The command x[n]:=1 won't change the n-th element of the list x, because x[n] is a value ! It is the *replace* macro that can modify the list's elements : replace(x, n, 12) will replace the n-th element of the x list (that must be a variable) with the value 12. The replacement value can also be a list.

The alternative

This is equivalent to the *if* (p. 56) command. That function returns the *Nil* value.

- **if <condition1> then <instructions> elif <condition2> then ... else <instructions> fi**
- Description: <condition> is a boolean expression, whose value is 0 (false) or 1 (true), elif stands for the contraction of else and if, so that tests cascads are allowed. Instructions are separated with a comma.
- Example(s): A piecewise defined function of the t variable:

  if t<=0 then 1-t elif t<pi/2 then cos(t) else t^2 fi

  to draw such a function, it is a better way to create a macro that is representing the function. For example, create a macro called *f* defined with the command if %1<=0 then 1-%1 elif %1<pi/2 then cos(%1) else %1^2 fi, the sequence %1 represents the first parameter of the macro. We can then create a graphical element (curve) by giving it a name and the parameters :t+i*f(t) or t+i*\f(t), in that second version, $f(t)$ is directly replaced by its expression.

The conditional loop

This is a version of the command:*Loop* (p. 48).

- **while <condition> do <instructions> od**
- Description: <condition> is a boolean expression, whose value is 0 (false) or 1 (true). Instructions are separated by a comma.
- Example(s): List of cubes under 1000:

  [x:=0, k:=0, while x<=1000 do x, Inc(k,1), x:=k^3 od]

  Running that command (in the command line at the bottom of the window) gives: [0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]. The first loop's instruction (x) returns x's value, the second (*Inc* (p. 47)) adds 1 to the k variable and returns *Nil*, the third (:=) assigns the cube of k to the variable x and returns the value *Nil*.

The iterative loop

This is a version of the commands *Seq* (p. 55) and *Map* (p. 49). There are two syntaxes:

- **for <variable> in <list of values> [step <step> or by/By <package of>] do <instructions> od**
- Description: For each value of the variable taken in the list, the instructions are executed. The *step* is 1 by default that is to say that the values are read one by one. The option **by** or **By** permits to read the values by packets and handling the case of the *jump* constant : with the *by* option the structure returns *jump* if there is one in the list. With the option **By**, *jump* is not returned. If the last packet is incomplete, it is not treated.

  To follow a list by components (two components are separated by a *jump*), we use *by jump* or *By jump*. For example : for z in [1,2,jump,3,4,5] by jump do sum(z) od returns [3,jump,12], but for z in [1,2,jump,3,4,5] By jump do sum(z) od returns [3,12].
- **for <variable> from <initial value> to <final value> [step <step> ou by/By <packets of>] do <instructions> od**
- Description: For each variable's value from the start to the final value, the instructions are executed. The value is incremented with the step (1 by default). The step can be negative and a non integer real. The **by/By** option is the same as above.

  NB: step and by can't be used at the same time.

  NB: While following a list by component (*by/By jump*), the constant **sep** is containing the value of the *jump* that is ending the current component. That value is in fact a particular complex, this is only the real part that gives the *jump* status (1E308), the imaginary part can be used to store a numerical information.
- Example(s):
  - The command for m in [-1,-0.25,0.5,2] do Color:=4*m, Courbe(t+i*t^m) od used in a User graphical element is useful to draw a cartesian curve family: $t \mapsto t^m$ for m varying in the list $[-1, -0.25, 0.5, 2]$, For each value of m the color of the drawing is also changed.
  - The command for k from -2*pi to 2*pi step pi/2 do Droite(1,0,k) od put in a User graphical element will allow to draw the lines whose equations are: $x = -2\pi$, $x = -2\pi + \pi/2$, ..., $x = 2\pi$.
  - Browsing by packet with the option **by** or **By**: The command for z from 1 to 7 by 2 to z, jump od returns: [1, 2, jump, 3,4, jump, 5, 6, jump]. In that example, the z variable takes successively the values [1, 2], [3, 4], and [5, 6], the last packet is incomplete, therefore not treated
  - Browsing with condition: the command for k in [1, 8, 4, 3, 2, 6, 5, 7] by 3 andif min(k)<=4 do k[1]+k[3] odfi returns [5,9].

2) Strings

2.1 Alphanumerical evaluation

For some functions's needs, some arguments are interpreted in alphanumerical form (strings) and not numerical form.

If the argument is a list, then **each list element is interpreted as a string** and all strings obtained are **concatenated**. During that operation, TeXgraph can encounter several cases:

- A string: It must be delimited with the characters " and ", if the string must have the character ", then it is doubled: "".
- A variable or a constant: its value is returned as a string.
- A function returning a string.
- An expression returning a string:
  - The macro **chaine()**: the command defining that predefined macro in *interface.mac*, is a message. That macro is used to store the strings while using the command *Input* (p. 46). TeXgraph replaces *chaine()* by the corresponding string.

  - Commands: *Map* (p. 49), *Si* (p. 56), *Loop* (p. 48), *Seq* (p. 55): those commands fonction like in the usual environment appart from the fact that the results are not evaluated as numbers but as a string. For example: Map(["(", Re(z), "/", Im(z), ")"], z, [1+i,2,3-i]) will return the string: (1/1)(2/0)(3/-1), but outside such a context it would give : [("(",1,"/",1,")","(",2,"/",0,")","(",3,"/",-1,")"]].

    Note: as structures like *for*, *if*, *repeat* et *while* are calling those commands, it is possible to use those structures in string-type arguments. Example: Message(for z in [1+i,2,3-i] do if Im(z)>0 then "(", Re(z),",", Im(z),")" fi od) will display (1,1).
  - A macro-string, that is to say a *macro returning a string* (p. 30).

- else : TeXgraph evaluate numerically the expression and the result is converted into a string.

Example(s): suppose that the macro *chaine()* was defined by the command "toto", and that we had created a global variable *A* set to 6, then the following list:

["Our friend", UpperCase(chaine()), "has ", A*A, " teeth"]

will give the string: Our friend TOTO has 36 teeth. On the contrary, if the variable *A* has not been defined, then the string will be Our friend TOTO has teeth, because *A*'s value is *Nil*.

2.2 To store a string

From version 1.97, TeXgraph's variables can store strings. But we can also use a macro that can play this role.

To create a macro-string (that part is here for backward compatibility):

- **SetStr(<name>, <expression> [, evaluate])**
- Description: create the macro called <*name*> whose command is defined by the <*expression*>, if <*evaluate*> is 1 (default value) Then the expression is evaluated as a string, else the <*expression*> is copied in the macro corpus as is. The argument <*name*> is alphanumerically evaluated.
- Example(s):
  - the command SetStr(test, sqrt(4)) will create a macro whose name is *test* with the value (a string): "2",
  - The command SetStr(test, sqrt(4), 0) will create a macro called *test* whose value is the string : sqrt(4) (without quotes).
  - The command SetStr(name, ["My name is ", %1], 0) will create a macro called *name* whose value is the string : ["My name is ", %1], At the execution of Message(@name("toto")) we will see the display My name is toto. Then a macro can be used as a function with one or more parameters returning a string.

To get the value of a macro-string (that part is here for backward compatibility):

- **GetStr(<name>)** or **GetStr(<name(arguments)>)**
- Description: evaluate alphanumerically the macro called <*name*> and returns the resulting string. There is a shortcut to that command, by adding the character @ before the <*name*>.
- Example(s): Message(@name) will have the same effect as Message(GetStr(name)).

2.3 Commands linked to strings

- The command **Concat(<argument 1>, <argument 2>, ..., <argument n>)**: each argument is interpreted as a string, the results are concatenated, and the command returns the resulting string (see the command *Concat* (p. 40)).
- The command **IsString(<arg>)**: returns 1 if <*arg*> is a string, 0 if not. If <*arg*> is a list, only the first argument is tested.
- **UpperCase(<expression>)** and **LowerCase(<expression>)**: return the <*expression*> respectively in uppercase and lowercase.
- The command **ScientificF(<real> [, <nb of decimal places>])**: convert the <*number*> into the scientific format and returns the result as a string.

- The command **Str(<macro name>)**: represents the text of the macro called <*macro name*> if it's not predefined (else it's the empty string). The argument <*macro name*> is itself interpreted as a string.
- The command **String(<expression>)**: if <*expression*> represent a variable, then the command returns the variable's name, else it returns the expression as a string.
- The command **String2Teg(<expression>)**: that function is evaluating alphanumerically the <*expression*> and returns the result as a string, doubling the " characters encountered. The result is then readable by the TeXgraph program.
- **StrComp(<string1>, <string2>)**: returns 1 if the two strings are identical, 0 if not.
- **StrCopy(<string>, <start>, <quantity>)**: returns the resulting string of the extraction (like the command *Copy* (p. 40)).
- **StrDel(<variable>, <start>, <quantity>)**: modify the <*variable*> by removing <*quantity*> characters from the <*start*> (like the command *Del* (p. 41)). if the arguvariable is a list of strings, only the first is handled. If the <*variable*> is not a string, the command is without any effect.
- **StrEval(<"expression">)**: The command is evaluating the <*expression*> (that must be a string), and returns the result as a string.
- **StrLength(<string>)**: returns the string length.
- **StrPos(<pattern>, <string>)**: returns the position (integer) of the first pattern in the string.
- **StrReplace(<string>, <pattern to remove>, <replacement pattern>)**: returns the resulting string.
- **Args(<k>)**: to be used within a macro. It is eveluating alphanumerically the argument number k and returns the resulting string. If there aren't any argument, then it's the whole argument's list that is treated.
- **StrArgs(<k>)**: to be used within a macro. It returns the argument number k as string. If there is no argument, then the whole argument list is treated.

2.4 Macros returning a string

The definition of the following macros can be found in the file *TeXgraph.mac*.

- **coord(<z> [, decimal places])**: returns the point coordinates whose affix is <z> as a couple (x, y) with the maximum <*decimal places*> asked (4 by default). This macro can be used as a string in functions or macros that can handle strings as argument. Example : Label(z,@coord(z)).
- **engineerF(<x>)**: returns the real <x> as a string in engineer size, that is to say $\pm m \times 10^n$ with m in the interval $[1; 1000[$ and n an integer multiple of 3. This macro can be used as a string in functions or macros that can handle strings as argument.
- **epsCoord(<z> [, decimal places])**: returns the point coordinates whose affix is <z> in the format:x y (eps format coordinates) with the maximum <*decimal places*> asked (4 by default). This macro can be used as a string in functions or macros that can handle strings as argument.
- **label(<expression>)**: the expression is alphanumerically evaluated delimited with the symbol $ if the variable *dollar* has the value 1. The macro is returning the resulting string. For example : [dollar:=1, @label(2+2)] returns "4".

  This macro is used by the macro: *GradDroite* (p. 101).
- **svgCoord(<z> [, decimal places])**: returns the point coordinates whose affix is <z> with the format x y (svg coordinates) with the maximal <*decimal places*> asked (4 by default). This macro can be used as a string in functions or macros that can handle strings as argument.

  That macro is handling the current transforming matrix.
- **texCoord(<z> [, decimal places])**: returns the point coordinates whose affix is <z> as the couple (x, y) (tex format coordinates tex) with the maximal <*decimal places*> as asked (4 by default).This macro can be used as a string in functions or macros that can handle strings as argument.

  That macro is handling the current transforming matrix.

- **ScriptExt()**: returns the string `".bat"` under windows and `".sh"` if not. (shell script files extension).
- **StrNum(<numeric value>)**: replace the decimal point by a comma if the predefined variable *usecomma* is set to 1 and return the resulting string. The number of decimal places is determined by the variable *nbdeci*, and the display format with *numericFormat* (0: default format, 1: scientific, 2: engineer).

  Example: [usecomma:=1, nbdeci:=10, Message(@StrNum(10000*sqrt(2)))] displays: 14142,135623731.

  Example: [usecomma:=1, nbdeci:=10, numericFormat:=1, Message(@StrNum(10000*sqrt(2)))] displays: 1,4142135624E4.

  Example: [usecomma:=1, nbdeci:=10, numericFormat:=2, Message(@StrNum(10000*sqrt(2)))] displays: 14,1421356237E3.

  That macro is used by the macro *GradDroite* (p. 101).

2.5 Constants and variables

2.6 Predefined constants

- Maths constants: i, π, e.
- The TeXgraph version number is stored in the constant called **version**.
- The constant **Windows** comes with the value 0 or 1 according to your operating system.
- The constant **GUI** comes with the value 0 or 1 showing that you are using TeXgraph's gui or not.
- The jump constant: *jump* . This constant is used to separate connex components of a polyline. We must also add that the polylines are automatically clipped by TeXgraph according to a rectangle that is the current window when drawing.
- Example(s): the equation $y = 1/x$ can be drawn from the following polyline with the command: [Seq(t+i/t,t,-5,0,0.1), jump, Seq(t+i/t,t,0,5,0.1)].
- The *Nil* constant. This is a non-value constant, that can be used to make comparisons. For example if we want to know if the variable x is not empty : if x<>Nil then ....
- Constants that are strings:
  - **InitialPath**: Path to the TeXgraph directory (here are the executables and scripts).
  - **DocPath**: path to the doc TeXgraph directory. That directory contains docs int pdf format (eg: TeXgraph.pdf)
    Example: The command Exec("xpdf","TeXgraph.pdf",DocPath), will open the TeXgraph.pdf file with the program xpdf.
  - **UserMacPath**: path to the user macros directory. Under linux this is the directory :
    `$HOME/TeXgraphMac` and under windows it has to be created by the user and the path must be in the environment variable *TeXgraphMac*. When the user is loading a macro file (*.mac) or a model file (*.mod) TeXgraph is seeking in the current directory, then in the directory *UserMacPath* and finally in the macros subdirectory of the directory indicated in the string : *InitialPath*.
  - **TmpPath**: Path to a temporary directory. This is the directory `$HOME/.TeXgraph` under linux, and `c:\tmp` under windows.
  - **JavaviewPath**: Path to the file *javaview.jar* if you have installed it. Its value has to be defined in the config file: menu *Preferences/Configuration file*.
  - **LF**: Insert a line feed when the string is displayed.
  - **Diese**: returns the hash character (used as delimiter in TeXgraph source files).
  - **DirSep**: returns the separating character used by the system for paths to files.
  - **ND**: means " non defined". It is containing the string `"_ND"`. It is used when opening *csv* files to indicate empty elements.
- The exporting constants: **tex**, **teg**, **pst**, **pgf**, **eps**, **psf**, **tkz**, **pdf**, **epsc**, **pdfc**, **svg** et **bmp**, These are the possible values of the constant **ExportMode** (set by TeXgraph at exporting time). Added to those, for the 3D, the constants: **obj**, **geom**, **jvx**.

- Constants: **Xmin**, **Xmax**, **Ymin**, **Ymax**: those are setting the graphical window. **Xscale** and **Yscale**: représentents (in cm) the scale on Ox and Oy axes. The constants can only be modified by the menu or the function *Fenetre* (p. 44).
- Constants : **margeG**, **margeD**, **margeH**, **margeB**: those are defining the margins around the graphic (in cm). Those constants can only be modified with the menu or the function *Marges* (p. 49).
- The constants **line**, **linearc**, **bezier**, **curve**, **arc**, **ellipticArc**, **ellipse**, **circle**, **closepath**, **move**: Those are needed to build paths in the command *Path* (p. 91).
- The graphic constants:
  - Colors: there is a dedicated *dedicated* (p. 11) chapter.
  - Line styles:
    - **noline** [=-1],
    - **solid** [=0],
    - **dashed** [=1],
    - **dotted** [=2],
    - **userdash** [=3], this style is using the variable **DashPattern** that is defining the pattern, which is a length list (unit: points) using the form: *[line length, jump length, line length, jump length, ...]*. For example DashPattern:=[2,3,0.1,3] will alternatively give lines and points.
  - Line endings:
    - **butt**: straight end at the last point (default),
    - **round**: rounded end after last point,
    - **square**: square end after the last point.
  - Join lines:
    - **miter**: miter join. The variable *Miterlimit* (set to 10) permits to handle the spikes length.
    - **round**: rounded join (default),
    - **bevel**: bevel join.
  - Line thickness (using the **tenth of a** TEX point):
    - **thinlines** [=2],
    - **thicklines** [=8],
    - **Thicklines** [=14], the variable *Width* can also set the thickness.
  - Point style (à la pstricks):
    - **dot** [=0],
    - **dotcircle** [=1],
    - **square** [=2],
    - **square'** [=3] (filled square),
    - **plus** [=4],
    - **times** [=5],
    - **asterisk** [=6],
    - **oplus** [=7],
    - **otimes** [=8],
    - **diamond** [=9],
    - **diamond'** [=10],
    - **triangle** [=11],
    - **triangle'** [=12],
    - **pentagon** [=13],

    * **pentagon'** [=14],

  – Label styles (by default the text is horizontally and vertically centered)

    * **left** : the reference point is on the left of the text,
    * **right** : the reference point is on the right of the text,
    * **top** : the reference point is at the top of the text,
    * **bottom** : the reference point is at the bottom of the text,
    * **baseline** : the reference point is the text baseline,
    * **framed**: the text is framed,
    * **special**: the text is written as is in the exported file (it is not displayed on the screen). This is allowing to write directly in a LaTeX or pgf or pstricks file (even eps).
    * **stacked**: the text can be written on several paragraphs (ie: paragraph jumps are allowed).
      Example: LabelStyle := top+framed, the text is horizontally centered, the reference point is at the top of the text, and the text is framed.

  – Fill styles for polygons (hatches are calculated by TeXgraph for the LaTeX output):

    * **none** [=0]: empty.
    * **full** [=1]: the polygon is filled with the color given in *FillColor*, without effect in the *tex* export.
    * **bdiag** [=2]: oriented hatches SW -> NE (45 degrees angle),
    * **hvcross** [=3]: combined horizontal and vertical styles ,
    * **diagcross** [=4]: combined bdiag and fdiag styles,
    * **fdiag** [=5]: oriented hatches NW -> SE (45 degrees),
    * **horizontal** [=6]: horizontal hatches,
    * **vertical** [=7]: vertical hatches.

  – Labels size:

    * **tiny**,
    * **scriptsize**,
    * **footnotesize**,
    * **small**,
    * **normalsize**,
    * **large**,
    * **Large**,
    * **LARGE**,
    * **huge**,
    * **Huge**.

  – constants linked to 3D:

    * **ortho**: kind of projection,
    * **central**: kind of projection,
    * **sep3D**: separator for the command *Build3D* (p. 147).

2.7 Global predefined variables

The following variables are considered as predefined. So are all the variables that are loaded from a macros file that is loaded as the TeXgraph program is starting. The predefined variables do not appear in TeXgraph's window, and are not recorded with the graphic.

The following variables correspond to the different "static fields" of the graphical elements:

- **Arrows**: Arrows number, first set to 0

- **AutoReCalc**: automatic calculation of the graphic elements, initial value:1 (for *True*), it can also get the value 0 (*False*). In the case of the value is zero for one graphic element, only the function **ReCalc()** (button **R**) can force that element to be recalculated.
- **ForMinToMax**: the variable value is 0 or 1, if it's 1 then the variable t for the curves is following the interval `[Xmin,Xmax]`, else this is the interval `[tMin,tMax]`.
- Variables related to axes
  - **xylabelpos**: labels position on the axes, default value set to *bottom+left* (left to the Oy axis and bottom of the Ox axis).
  - **xylabelsep**: distance (cm) between the labels and the end of the graduations, default value set to *0.1 cm*.
  - **xyticks**: length (cm) of the graduations on the axes, default value set to *0.2 cm*.
- **Color**: color. Default value set to *black*,
- **DashPattern**: defining the drawing lines pattern in the style **userdash**. That variable is length list following the format: *[line length, jump length,line length, jump length, ...]*. For example DashPattern:=[2,3,0.1,3] will give a line-points succession.
- **DotStyle**: dot style. Default value : *dot*,
- **DotAngle**: rotating angle of the points (degrees). default value set to 0,
- **DotScale**: dot scale. Default value set to [1,1] (Ox and Oy scale).
- **DotSize**: dot size. Default value set to 2+2i. the value of this variable is a complex $x+iy$. x is the size (unit: points) and y is a positive number: the point diameter calulated with the formula: x+y*(line thickness).
- **Eofill**: default: 0. The value can only be 0 or 1. The value 1 indicate that the fill mod is following the even-odd rule, and the value 0 indicate the contrary case. The even-odd mode (*Eofill*=1) is not always very vell handled by the GUI version of TeXgraph but there are no problems with the exports.
- **FillColor**: The default fill color is *white*,
- **FillOpacity**: when the variable *FillStyle*=*full*, the FillOpacity is a value between 0 and 1 (default: 1 - no transparency). The transparency is not handled by the TeXgraph screen, but it is in the exports.
- **FillStyle**: default fillstyle : *none*,
- **IsVisible**: boolean value (0 or 1), showing or not the graphic element. The default value is 1 (visible)
- **LabelAngle**: direction of the labels relative to the horizontal, this is a degrees angle (default: 0).
- **LabelSize**: size of the labels. Default value: *small*,
- **LabelStyle**: the label style is by default set to 0 (horizontally and vertically centered).
- **LineCap**: this is defining the line ends. Default value: *butt*.
- **LineJoin**: defining the line join type. Default value: *round*.
- **LineStyle**: line styles. Default: *solid*. In the current windows version the lines are displayed as solid lines as soon as the thickness is above 1 pixel, even if the style is dashed or dots. This problem doesn't occur in the exports.
- **MiterLimit**: this limits the length of the joints as soon as *LineJoin* is set to *miter*. Default value: 10.
- **NbPoints**: number of points for the curves. default : 50.
- **PenMode**: drawing mode, 0=normal mode, 1=NotXor mode, if a graphical element created using NotXor mode is redrawn, it is removed. Only the background is left. We can then modify the position of that element and redraw it. This technique is usefull to move objects without redrawing the others (avoiding trembling images). The default value of this variable is 0

- **StrokeOpacity**: Opacity/transparency handling for the lines when *LineStyle* is not equal to *noline*, this is a value between 0 and 1, defaulted to 1, the value 1 means no transparency. The transparency is not handled by the TeXgraph screen, but it is in the exports.

- **TeXLabel**: boolean variable (0 or 1) showing that the labels have (or not) to be displayed as images in the graphic interface after a TeX compilation. The default value is 0

- **tMax**: maximal value of the t parameter. Default :5.

- **tMin**: minimal value of the t parameter. Default value: -5.

- **Width**: line thickness in **an integer of the tenth of point** of TEX. Default value: *thinlines*.

Creating a graphic element does not implies that a constant with the same name is created. Though, it is possible to get the list of the points compositing the graphical element with the command *Get* (p. 44). But the graphical element whose name we are using has to be **already created** or the function *Get* will return the *Nil* value.

3D related variables:

- **theta** and **phi**: these are used for surfaces projections calculations. Default values 30 and 60 degrees, respectively. The first represents the lattitude and the second is the colatitude. Those can be also modified using a button in the toolbar.

- **AngleStep**: represents the angular step (radians) when we turn a 3D object with the buttons with arrows on the toolbar. The default value is $\pi/36$ (5 degrees).

2.8 Variable declaration

As soon as TeXgraph encounters a name in an expression, it is examining if it's followed by parenthesis [ex: *toto(..)*]:

- If it's the case: TeXgraph is testing if it is a predefined function, else it is considering this is a macro [1] (even if it is not existing yet).

- If it's not the case: then it **first** tests if exists a **local** variable with that name. If not, it tests if exists a **global** variable with that name . If not, it **creates** a **local**[2] variable with that name [and set to *Nil*]

It's then not necessary to declare local variables, the first occurence is handled as a declaration. Though, we may need a *x1* variable [for example] to be local though there is already a global variable with that name. We can tell TeXgraph to consider *x1*, by just adding the $ character before its name: *$x1* (adding it to the first occurence is enough).

2.9 Global variables

- Global variables declaration is done using the menu or the *New* button in the global variable zone (on the right of the window). Those wear a name and are defined by a command and will be saved with the graphic in the source file.

- While right-clicking on a point in the window, TeXgraph propose to save that point affix as a global variable, that can be useful to place labels, or create a figure without taking care of the coordinates.

- When the user modifies a variable (double-click on the name in the global variable zone), the graphical elements are automatically updated. This can be desactivated by unticking the corresponding option in the attributes.

2.10 Automatical recalculation

Creation or modification of a global variable leads to an automatical recalculation of the whole graphic, that is to say:

- of all the non predefined global variables,

- of all the non predefined macros,

- of all graphical elements that are in the *Recalcul Automatique* mode.

NB: Modifying the window using the menu also leads to the automatical recalculation.

[1] A macro without parameter has though two parenthesis: *toto()*.

[2] local to the current analysed expression, that analysis is transforming the expression into a tree, as soon as the tree is destroyed, the corresponding local variables are also destroyed.

2.11 Variables in the TeXgraph.mac and interface.mac files

Those files are automatically loaded at program startup (with *color.mac* and *scene3d.mac*). Its content is considered predefined, not displayed on the screen, not saved with the graphics and is present in the memory until the program is closed.

Here are the main variables (the variables used as options in some macros are not cited):

- **stock**, **stock1** to **stock5** (=*Nil*): storage variables.
- **mm** (=Ent(7227/254)): integer tenth of points (of TeX) corresponding to 1 millimeter. Useful for the lines thickness given in integer tenth of points. For example : Width:=1.5*mm will give 1.5 mm thickness.
- **backcolor** (=white): the background color, updated with the macro Helprefbackgroundmacbackground, and used in some exports.
- **deg** (=pi/180): degrees towards radians conversion, for example: alpha:=40*deg.
- **rad** (=180/pi): radians towards degrees conversion, for example: LabelAngle:=pi/16*rad.
- **tailleB** (=145+i*30): button length and height (pixels).
- **DeltaB** (=32*i): difference between two buttons + button's height.
- **RefPoint** (=2+5*i): reference point for the first button.
- **NbBoutons** (=0): button counter.
- **Xfact** (=1.1) and **Yfact** (=1.1): variables used for zooms (+ and - buttons in the toolbar).
- **usecomma** (=0): this variable is handled by the macro *GradDroite* (p. 101). When its value is 1, the point is replaced with a comma in the numerical displays for the graduations. The replacement itself is made by the macro *StrNum* (p. 32).
- **numericFormat** (=0): this variable is handled by the macro *StrNum* (p. 32). It shows if numbers have to be displayed using default format (value 0), or scientific format (value 1) or engineer format (value 2).
- **nbdeci** (=2): number of decimal places in the numerical displays. That variable is handled by the macro *StrNum* (p. 32), itself used by the macro *GradDroite* (p. 101).
- **maxGrad** (=100): this variable is handled by the macro *GradDroite* (p. 101), and shows the maximal number of graduations.

3D linked variables:

- **Origin** (= [0,0]): the origin,
- **vecI** (= [1,0]): first base vector,
- **vecJ** (= [i,0]): second base vector,
- **vecK** (= [0,1]): third base vector,
- **Xinf** (= -5), **Xsup** (= 5), **Yinf** (= -5), **Ysup** (= 5), **Zinf** (= -5), **Zsup** (= 5): 3D window.

3) Macros

A macro is a user-created function that is returning a result (a complexes list, or strings, or *Nil*). TeXgraph admits three types of macros:

- those that are loaded at program startup: considered as **predefined** and not shown in the editable macros list, not removable and not saved in the source **.teg* files either.
- those that are loaded using the menu with the option *File/ load macros*, or via the instruction *InputMac* (p. 47): considered as **predefined**, not shown in the editable macros list, not saved in the **.teg* source files, but they will be removed from memory at next file loading.
- those that are created during program running: those are editable and saved in the source files **.teg*.

A macros file is a text file **.mac* that only contains macros and eventually global variables. It is possible to create/edit a macro file directly in TeXgraph or using a text editor of your choice provided to use UTF-8 encoding.

3.1 Macro creation

- A macro is defined by a name and a command. A macro can have local variables and parameters called : *%1, %2, ...* Declaring parameters is not necessary.

- So that the macro text will not be recorded in one line in the file **.teg*, it has to be formated using line feeds [with the *Enter* key] [3] therefore increasing the lisibility. Added to that, it is also possible to add comments. Two methods for the comments: between braces : `{this is a comment}`, or a comment line beginning with `//`.

- Example(s): here is the command defining a macro called *racine* that is giving the n-th roots of a complex number:

```
{usage: racine(n,z), gives the n-th root list of z}
if (Ent(%1)=%1) And %1>0
then  $a:= abs(%2)^(1/%1),
      for $k from 0 to %1-1 do a*exp(i*(Arg(%2)+$k*2*pi)/%1) od
fi
```

  - It is tested if the first parameter (represents n) is a positive integer, then we store in a local variable the n-th root of the modulus of z (second parameter) then we give the solutions list (else the macro returns *Nil*).
  - The execution of [$a:=3, racine(a,i)] gives: [0.866025+0.5*i, -0.866025+0.5*i, -i].

- TeXgraph doesn't test the number of arguments, the implicit value of the missing arguments is *Nil*. If there are too many arguments, surplus are ignored.

3.2 Immediate or deferred development

- As a command is witten as a string, right before executing it, TeXgraph has to analyze that string before transforming it into a tree. During that analysis, a macro may be immediately developed or not.

- When **analyzing** [$a:=3, racine(a,i)]: TeXgraph builds the corresponding tree, while conserving the word *racine*. When the tree is evaluated, a copy of the macro's expression *racine* replacing the $1 parameter by the a variable [4] and the parameter $2 by i, then evaluate the resulting expression [5] and destroy the copy: **this is the deferred development**.

- When **analyzing** [$a:=3, \racine(a,i)]: TeXgraph replaces *racine* with the macro's expression, replacing the parameter %1 by the a variable and the parameter %2 by i, this is equivalent to analyze the command:

```
[$a:=3,
if (Ent(a)=a) And a>0
then  $a:= abs(i)^(1/a),
      for $k from 0 to a-1 do a*exp(i*(Arg(i)+$k*2*pi)/a) od
fi]
```

  This is the immediate development. We see that this time there is only one variable a. Therefore, that command won't give the right result (it gives i). On the contrary the command [$b:=3, \racine(b,i)] gives the right result ([0.866025403784+0.5*i,-0.866025403784+0.5*i,-i]). The immediate development can take place only if the macro already exists, else it is deferred development.

- Immediate development should be avoided when the macro has local variables and there is a risk of homonymous with the variables of the calling expression. Though, there are case where it is more interesting than the deferred development. For example, the macro called *f* by the command %1*arctan(%1)/(1+%1^2) and if the graphical element *Curve/Parametric* is created by the expression t+i*\f(t), then the expression will in reality be t+i*t*arctan(t)/(1+t^2) and as that expression will be many times evaluated, it will be faster than the expression t+i*f(t), because in a deferred development, the macro *f* will be called at each evaluation of the expression.

  On the other side, immediate development also permits to use macros as variables or as **shortcuts**.

- Macros can be recursive.

[3] This is also the case with the *User-defined* graphical elements.
[4] This is not the value of a that is replacing $1 but its adress
[5] in that expression there are in fact two variables but there is not ambiguity because one is "plugged" in the local variables of the macro, and the other in the local variables of the "calling" expression.

Chapter V

Commands

Note:
<argument>: means that the argument is **mandatory**.
[argument]: means that the argument is **optional**.

1) Args

- **Args(<integer>)**.
- Description: this function only works within a macro, it evaluates and returns the <*integer*>-th argument of the calling macro, otherwise (outside this context) it returns *Nil*. See also the command *StrArgs* (p. 57).
- Example(s): see the function *Nargs* (p. 50).

2) Assign

- **Assign(<expression>, <variable>, <value>)**.
- Description: that function evaluate the <*value*> and assign it to the variable called <*variable*> in <*expression*>[1]. The function *Assign* returns the *Nil* value. That function is useful to write macros that take an expression as parameter and has to evaluate it
- Example(s): here is a macro Bof that takes a function f(t) as a parameter and calculate the list $[f(0), f(1), ..., f(5)]$:

```
for $k from 0 to 5 do Assign(%1,t,k), %1 od
```

%1 represents the first parameter of the macro (that is f(t)), the loop: for k from 0 to 5 executes the command [Assign(%1, t, k), %1], that is assigning the value k to the variable t in the expression %1, then evaluate %1. The execution of Bof(t^2) gives : [0,1,4,9,16,25]. The execution of Bof(x^2) gives *Nil*.

3) Attributes

- **Attributs()**.
- Description: that function opens the window to edit the attributes of a graphical element. That function returns 1 if the user has chosen *OK*, it returns 0 if the *Cancel* was chosen. If the user has chosen *OK*, then the global variables corresponding to the attributes are updated.

4) Border

- **Border(<0/1>)**
- Description: that function is used to know if a frame has to be drawn or not around the graphic margins in the exports. When the argument value is 0 (default value), the frame is not drawn.

  When the argument is empty, that function returns the state of the border at export (0 or 1). Else, it returns the *Nil* value.

[1] This is the first occurence of <*variable*> in <*expression*> that is assigned, because all occurences point to the same <memory cell>, except for the macros after parameters assignment.

5) ChangeAttr

- **ChangeAttr(<element1>, ..., <elementN>)**
- Description: this function allows editing attributes of the graphical elements <*element1*>, ...,<*elementN*>, by assigning the current value of the attributes. Arguments are interpreted as strings. That function returns *Nil*.

6) Clip2D

- **Clip2D(<polyline>, <convex contour> [, close(0/1)]).**
- Description: that function allows to clip the <*polyline*> that has to be a variable containing a list of complexes, with the <*convex contour*>, that is also a complexes list. The function calculates the resulting polyline, then modify the variable <*polyline*>. The last argument <*close*> shows if the <*polyline*> has to be closed or not (0 by default). The function returns *Nil*.

7) CloseFile

- **CloseFile()** or **CloseFile(<"file 1">, <"file 2">,...,<"file N">).**
- Description: this function close the files whose name (strings) are cited as arguments. In the case of there isn't any argument, only the last opened file will be closed. At this time the physical writing in the file is done. The files have to be opened with the command *OpenFile* (p. 52).

8) ComposeMatrix

- **ComposeMatrix(<[z1, z2, z3]>)**
- Description: this function is used to compose the current matrix (it has effects on all the graphics elements except the axes and grids in the actual version) with the matrix <*[z1, z2, z3]*>. That matrix represents the analytic expression of an affine transformation, this is a three complexes list: $z1$ is the vector's affix of the translation, $z2$ is the first column vector's affix of the matrix of the linear part in the base (1,i), and $z3$ is the second column vector's affix of the matrix of the linear part. For example the matrix of the identity is :[0,1,i] (this is the default matrix). (See also commands *GetMatrix* (p. 45), *SetMatrix* (p. 56), et *IdMatrix* (p. 46)).

9) Concat

- **Concat(<argument 1>, <argument 2>, ..., <argument n>).**
- Description: this command reads each argument as a string, concatenate the differents results and returns the resulting string.

10) Copy

- **Copy(<list>, <start index>, <number>).**
- Description: that function returns the <*number*> elements in the <*liste*> from the element number <*start*> [included]. If <*number*> is zero, then the function returns all the elements in the list from the element number <*start*>.

  If the <*start*> number is negative, then the list is browsed from the right to the left starting from the last element. The last element index is -1, the penultimate's is -2 $\cdots$ etc. The function returns the <*number*> elements of the list (or the whole list if <*number*> is zero) browsed to the left, but the list is returned in the same direction than the given <*list*>, and the last is not modified.
- Example(s):
  - Copy([1,2,3,4],2,2) returns [2,3].
  - Copy([1,2,3,4],2,5) returns [2,3,4].

- Copy([1,2,3,4],2,0) returns [2,3,4].
- Copy([1,2,3,4],-1,2) returns [3,4].
- Copy([1,2,3,4],-2,2) returns [2,3].
- Copy([1,2,3,4],-2,0) returns [1,2,3].

NB: for compatibility reasons with the old release, the index 0 is also corresponding to the last element of the list.

11) DefaultAttr

- **DefaultAttr()**
- Description: this function assigns all the attributes variables (*Color*, *Width*, ...) to the default value, and returns *Nil*.

12) Del

- **Del(<list>, <start>, <number>)**.
- Description: removes from the <*list*> <*number*> elements from the <*start*> [included]. If <*number*> is zero, then the function removes all the elements from the <*start*>-th.

  If the <*start*> number is negative, then the list is browsed from the right to the left starting from the last element. The last element index is -1, the penultimate's is $-2\cdots$ etc. The function removes the <*number*> elements from the list (or the whole list if <*number*> is zero) to the left.

  The parameter <*list*> has to be a **variable name**, it is modified and the function returns *Nil*.
- Example(s): the command [x:=[1,2,3,4], Del(x,2,2), x] returns [1,4].

  The command [x:=[1,2,3,4], Del(x,-2,2), x] returns [1,4].

NB: for compatibility reasons with the older version, the index 0 also corresponds to the last element of the list.

13) Delay

- **Delay(<milliseconds>)**
- Description: pause the program during the given delay (milliseconds).

14) DelButton

- **DelButton(<text1>, ..., <textN>)**
- Description: that functions removes, in the column located at the left of the drawing, the buttons with inscriptions <*text1*>, ..., <*textN*>. If the list is empty (*DelButton()*), then all the buttons are removed. Arguments are interpreted as strings. The function returns *Nil*.

15) DelGraph

- **DelGraph(<element1>, ..., <elementN>)**
- Description: that function removes the graphical elements called <*element1*>, ..., <*elementN*>. If the list is empty(*DelGraph()*), then all the elements are removed. The arguments are interpreted as strings. This function returns *Nil*.

16) DelItem

- **DelItem(<name1>, ..., <nameN>)**
- Description: the function removes from the pull-down list at the left of the drawing zone, items called :<*name1*>, ..., <*nameN*>. If the list is empty (*DelItem()*), then the whole list is removed. Arguments are interpreted as strings. This function returns *Nil*.

17) DelMac

- **DelMac(<mac1>, ..., <macN>)**
- Description: that function removes the (non predefined) macros called <*mac1*>, ..., <*macN*>. If the list is empty (*DelMac()*), the command has no effect. The arguments are interpreted as strings. That function returns *Nil*.

18) DelText

- **DelText(<text1>, ..., <textN>)**
- Description: that function removes in the column located at the left of the drawing zone, the labels <*text1*>, ..., <*textN*>. If the list is empty (*DelText()*), then all the labels are removed. The arguments are interpreted as strings. That function returns *Nil*.

19) DelVar

- **DelVar(<var1>, ..., <varN>)**
- Description: this function removes the non predefined global variables called <*var1*>, ..., <*varN*>. If the list is empty (*DelVar()*), the command has no effect. The arguments are intrepreted as strings. That function returns *Nil*.

20) Der

- **Der(<expression>, <variable>, <list>).**
- Description: this function calculate the derivative of the <*expression*> with respect to <*variable*> and evaluate it by giving to <*variable*> the values of the <*list*>. The function *Der* returns the list of results. But if we need the *derivative's expression*, then we prefer the function *Diff* (p. 42).
- Example(s):
  - the command Der(1/x,x,[-1,0,2]) returns [-1,-0.25].
  - here is the text of a macro called tangente that takes an expression $f(x)$ as first parameter, a real value $x0$ as second parameter an that draw the tangent to the curve at the point x0 abscissa:

    [Assign(%1,x,%2), $A:=%2+i*%1, $Df:=Der(%1,x,%2), Droite(A, A+1+i*Df)]

    We assign the value $x0$ to the variable x in the $f(x)$ expression, we store in the variable A the point $(x0, f(x0))$ (in affix format), we store in a variable Df the derivative at $x0$ ($f'(x0)$), then we draw the straight line passing through A with the direction of the vector whose affix is $1+if'(x0)$.

21) Diff

- **Diff(<name>, <expression>, <variable> [, param1,..., paramN])**
- Description: that function create a macro called <*name*>, if it already existed then the old macro will be overwritten unless it is a predefined macro then there will be no effect. The created macro body corresponds to the derivative of the <*expression*> with respect to <*variable*>. Optional parameters are variable names. The variable name <*param1*> is replaced in the derivative's expression with the parameter %1, the name <*param2*> is replaced with %2 ... etc. That function returns *Nil*.
- Example(s): after the command execution (in the command line at the bottom of the window): Diff(df, sin(3*t), t), a macro called df is created and its content is: 3*cos(3*t), This is a macro without parameter that contain a local variable t, it will have to be used with immediate development (ie: preceded by the symbol: \) [2]. On the contrary, after the command Diff(df,sin(3*t),t,t), the content of the macro df is: 3*cos(3*%1) that is a one parameter macro.

[2] For example, if you want to plot the graph of that function, in the menu option *Courbe/Paramétrée* (Curve/parametric), you'll have to enter the command *t+i*\df* and not *t+i*df(t)*.

22) Echange (Exchange)

- **Echange(<variable1>, <variable2>)**.
- Description: the function exchange the two variables, in fact only the adresses are exchanged. The content are not duplicated but it would if we use the command:

  [aux:=variable1, variable1:=variable2, variable2:=aux]

  The function *Echange* returns *Nil*.

23) EpsCoord

- **EpsCoord(<affix>)**
- Description: returns the affix exported in eps form. For other forms, see the macros : *TeXCoord* (p. 73) and *SvgCoord* (p. 73).

24) Eval

- **Eval(<expression>)**.
- Description: that function evaluate the <*expression*> and returns the result. The <*expression*> is interpreted as a *string* (p. 29).
- the function *Input* (p. 46) returns the input as a string in the macro called *chaine()*. The function *Eval* evaluate that string (like every TeXgraph command) and returns the result.
- Example(s): here is a command asking for a value for the variable *x*:

```
if Input("x=", "Please input a value for x", x )
then x:= Eval( chaine() )
fi
```

25) Exec

- **Exec(<program> [, <argument(s)>, <working directory>, <wait>, <show window>])**.
- Description: the function executes a <*program*> (or a script) specifying any <*arguments*> and a <*working directory*>, the three other arguments are interpreted as strings. The argument <*wait*> whose value is 0 (default) or 1, indicate that the program has to wait or not the end of the process. The last argument <*show window*> whose value is 0 (default) or 1, indicate if the executing window has to be visible or not, that argument is valid under windows. The function returns *Nil*. An error message is displayed if: resources are insufficient, or the program is invalid, or the path is invalid.
- the predefined string *TmpPath* contains the path to a temporary directory. The macro *Apercu* exports the current graphic in that directory using the pgf format in the file *file.pgf*, then execute *pdflatex* on the file *apercu.tex*, and wait until the end of the execution before launching the pdf reader.
- Example(s): the macro *Apercu* contained in interface.mac is:

```
  [Export(pgf,[TmpPath,"file.pgf"]),
  Exec("pdflatex", ["-interaction=nonstopmode apercu.tex"],TmpPath,1),
  Exec(PdfReader,"apercu.pdf",TmpPath,0)
]
```

26) Export

- **Export(<mode>, <file>)**.
- Description: the function exports the current graphic, <*mode*> is a numerical value among the following constants: tex, pst, pgf, tkz, eps, psf, pdf, epsc, pdfc, svg, bmp, obj, geom, jvx or teg. The exports are done in the <*file*> that is containing the file name, with (or not) the path.

  The predefined string *TmpPath* contains the path to a temporary directory. The macro *Apercu* exports the current graphic in that directory using the pgf format in the file *file.pgf*, then execute *pdflatex* on the file *apercu.tex*, and wait until the end of the execution before launching the pdf reader.
- Example(s): the macro *Apercu* contained in interface.mac is:

```
  [Export(pgf,[TmpPath,"file.pgf"]),
  Exec("pdflatex", ["-interaction=nonstopmode apercu.tex"],TmpPath,1),
  Exec(PdfReader,"apercu.pdf",TmpPath,0)
]
```

27) ExportObject

- **ExportObject(<argument>)**
- Description: this command has only effect during exports . It exports the<*argument*> in the output file, that <*argument*> is the name of a graphic element or a graphic command (like the function *Get* (p. 44)). It can be useful to write personal exports. See *that section* (p. 24).

28) Fenetre (window)

- **Fenetre(<A>, <B> [, C])**.
- Description: that function modifies the graphical window, equivalent to the option *Paramètres/Fenêtre* (preferences), **but the graphical elements are not automatically recalculated**. The parameter <*A*> and the parameter <*B*> are affixes of two opposite corners of the view window, and the optional parameter <*C*> represents the two scales, more precisely, the real part is for the Ox axis and the imaginary part gives the scale on the Oy axis (in centimeters for both axes), those two values must be positive. That function returns *Nil*.

29) FileExists

- **FileExists(<file name>)**
- Description: that command returns 1 if the given file name exists, 0 if not.

30) Free

- **Free(<expression>, <variable>)**.
- Description: the function returns 1 if the <*expression*> contains the <*variable*>, 0 if not. when the second argument is not a variable name, the function returns *Nil*.

31) Get

- **Get(<argument> [, clip(0/1), current matrix (0/1)])**.

- Description: when the parameter <*argument*> is an *identifier*, the function is looking for a graphical element whose name is <*argument*>, if it is the case, then the function returns the list of the points of that element, else it returns *Nil*. In that case, the optional argument is ignored.

  When the <*argument*> is not an indentifier, it is considered as a *graphical function*, the function *Get* returns the list of the points of that graphical element built with that graphical function, without creating that element in question. The first optional argument <*clip*> (1 by default) shows that the element has to be clipped (value 1) or not (value 0) by the current window. The second optional argument <*current matrix*> (0 by default) shows that the element has to be modified (or not) by the current matrix.

  When the argument is empty: *Get()*, the function returns the list of the points off all the graphical elements already built. Those that are hidden are ignored.

- Example(s): Get(Cercle(0,1)) returns the list of the points of the circle centered in 0, with radius 1, without creating the circle. The list is clipped by the graphic window.

- Example(s): How is handled such a list of points of a graphical object:

```
\begin{texgraph}[name=Get, file]
Cmd view(0,6.25,-1.5,2);
   size(7.5);
   [tMin:=0, tMax:=6.25];
Graph axe = Axes(0,1+i);
    C1 = [Color:=blue,Width:=8,
       Cartesienne(1/x) ];
    C2 = [Color:=red,Width:=8,
       Cartesienne(sin(2*x)*1.5)];
    inter = [DotScale:=2,DotStyle:=dotcircle,
          Color:=forestgreen,
          Point(Get(C1) InterL Get(C2)),
          Color:=blue,
          LabelDot(5.5+0.1*i,"$C_1$","N"),
          Color:=red,
          LabelDot(3-i,"$C_2$","E"),
         ];
\end{texgraph}
```

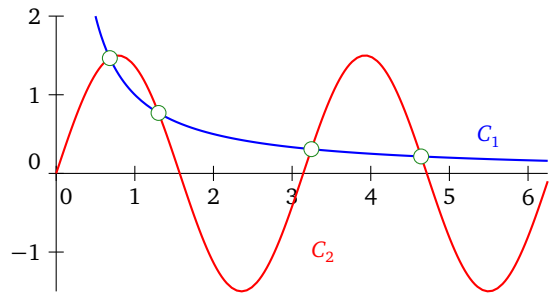


Figure 1: *Get*

32) GetAttr

- **GetAttr(<argument>)**

- Description: when the parameter <*argument*> is an *identifier*, the function is looking for a graphical element whose name is <*argument*>, if this is the case, then the attributes of that element become the current attributes and the function returns *Nil*. Else the argument is interpreted as a string and the function performs the same search.

33) GetMatrix

- **GetMatrix()**

- Description: the function returns the current matrix. (See also the commands : *ComposeMatrix* (p. 40), *SetMatrix* (p. 56), and *IdMatrix* (p. 46))

34) GetSpline

- **GetSpline(<V0>, <A0>,..., <An>, <Vn>)**

- Description: returns the list of the control points corresponding to the cubic spline passing through the points <*A0*> to <*An*>. <*V0*> and <*Vn*> are the speed vectors at the ends [constraints], if one of them is zero then the corresponding end is considered as free (without constraint). The result has to be drawn with the graphic command *Bezier* (p. 86).

35) GetStr

- **GetStr(<name>)** or **GetStr(<name(arguments)>)**
- Description: evaluate alphanumerically the macro called <*name*> and returns the resulting string. There is a shortcut for that command, by adding the operator @ in front of the <*name*>.
- Example(s): Message(@nom) will have the same effect as Message(GetStr(nom)).

36) GrayScale

- **GrayScale(0/1)** or **GrayScale()**.
- Description: that function activate/desactivate the conversion of the colors into grayscale. It has the same effect as the menu option *Paramètres/Gérer les couleurs*(ie: Preferences/manage the colors) in the graphic interface.

  When the argument is empty, the function returns the actual state (0 or 1). Else it returns *Nil*.

37) HexaColor

- **HexaColor(<hexadecimal value>)**
- Description: that function returns the color corresponding to the <*hexadecimal value*>, the value has to be passed as a string. See also the commadnd *Rgb* (p. 55).
- Example(s): `Color:=HexaColor("F5F5DC").`

38) Hide

- **Hide(<element1>, ..., <elementN>)**
- Description: the function hide the graphical elements called <*element1*>, ..., <*elementN*> by assigning its attribute *IsVisible* to false. The arguments are interpreted as strings. The function returns *Nil*.

  To hide everything, we use the command without any arguments: *Hide()*.

  To hide everything but one or several arguments, we use the command: *Hide(except, element1, ..., elementN)*. See also the command *Show* (p. 56).

39) IdMatrix

- **IdMatrix()**
- Description: change the current matrix into identity matrix. (See also the commands *ComposeMatrix* (p. 40), *SetMatrix* (p. 56), and *GetMatrix* (p. 45))

40) Input

- **Input(<message> [, title, string])**.
- Description: the function opens a dialog box with <*title*> in the title bar (empty by default), in which the parameter <*message*> is displayed, the parameter <*string*> is displayed in the input area. Those parameters are interpreted as *strings* (p. 29), the user is asked to type an input. If the user validate then the function *Input* returns 1 and the inputed string is **stored in the macro** *chaine()*. If the user does not validate or if the string is empty, thent the function *Input* returns 0.
- Example(s): see the function *Eval* (p. 43).

41) InputMac

- **InputMac(<file name>)** or **Load(<file name>)**.
- Description: the function loads into memory a macro file (**.mac*), or a model file (**.mod*), or any TeXgraph source file (**.teg*).

  In the first case (fichier **.mac*), the global variables and the macros will be considered as **predefined** (not shown on screen, not saved with the graphic, removed from memory as soon as a new graphic is started). The parameter <*file name*> is a string representing the file to be loaded with eventually its path. The function returns *Nil*, and if the file is already loaded, it has no effect. If the file to be loaded is in the *TeXgraph macros* directory , or in the *TeXgraphMac* directory ,then giving the path is useless.
- Example(s): InputMac("MesMacros.mac").

42) Inc

- **Inc(<variable>, <expression>)**.
- Description: the function evaluate the <*expression*> and adds the result to <*variable*>. That function is more useful than the command: variable := variable + expression, because in that command the <*variable*> is evaluated [ie: duplicated] to calculate the sum. The function *Inc* returns *Nil*.

43) Insert

- **Insert(<liste1>, <liste2> [, position])**.
- Description: the function inserts the <*liste2*> in the <*liste1*> at position number <*position*>. When position is 0 [default value], the <*liste2*> is added at the end. The <*liste1*> must be a variable and is modified. The function *Insert* returns *Nil*.
- Example(s): if the variable *L* contains the list [1,4,5], then after the command Insert(L,[2,3],2), the variable *L* will contain the list [1,2,3,4,5].

44) Int

- **Int(<expression>, <variable>, <lower bound.>, <upper bound.>)**.
- Description: the function calculate the integral of <*expression*> with the respect of the <*variable*> on the **real** interval defined by <*lower bound*> et <*upper bound*>. The calculation is done using the SIMPSON method two times accelerated with the ROMBERG method, <*expression*> is supposed definite and enough regular on the integration interval.
- Example(s): Int(exp(sin(u)),u,0,1) gives 1.63187 (Maple gives 1.631869608).

45) IsMac

- **IsMac(<name>)**.
- Description: the function shows if exists a macro called <*name*>. It returns 1 if it is the case, 0 if not.

46) IsString

- **IsString(<arg>)**.
- Description: the function returns 1 if <*arg*> is a string, 0 if not. When <*arg*> is a list, only the first argument is tested.

47) IsVar

- **IsVar(<name>)**.
- Description: the function shows if there is a global variable called <*name*>. It returns 1 if it is the case, 0 if not.

48) Liste (list)

- **Liste(<argument1>, ..., <argumentn>)** or **[<argument1>, ..., <argumentn>]**.
- Description: that function evaluate each argument and returns the results list **different from *Nil***.
- Example(s): Liste(1, Arg(1+2*i), sqrt(-1), Solve(cos(x)-x,x,0,1)) returns the result [1,1.107149,0.739085].

49) ListFiles

- **ListFiles()**.
- Description: that function is only available with the GUI version of TeXgraph, and is used in the command bar at the bottom of the window. It displays then the list of the macros files (**.mac*) loaded into memory.

50) ListWords

- **ListWords()**.
- Description: that fonction is available only with the GUI version of TeXgraph, and can be used in the command bar at the bottom of the window. It displays then the list of the words in memory (constants names, macros, commands, variables,...).

51) LoadImage

- **LoadImage(<image>)**.
- Description: the function loads the file <*image*>, that has to be a png, jpeg or bmp image. It is displayed as background, is a part of the graphic, and exported in the formats : tex (only visible in the postscript version), pgf, pst and teg. For the pgf format, this is the png or jpg that will be in the document, but for the pst or tex versions we need an eps version of the image. The argument is interpreted as a string and the function returns *Nil*.

  When the image is loaded, its size is fitted to the window, but it (the window) can be modified to the image proportions. From then the image position and size are freezed. We can then resize the window if we do not want the image to take the whole space. To modify the position or size of the image, we have to reload it.

52) Loop

- **Loop(<expression>, <condition>)**.
- Description: that function is a **loop** that build a list by evaluating the <*expression*> and <*condition*> until the result of <*condition*> is 1 (for *True*) or *Nil*, The function *Loop* returns then the list of the results of <*expression*>. That command is the internal representation of the loop *repeat* (p. 28) whose usage is preferable on lisibility grounds.
- Example(s): the commands (équivalent):

  [n:=1, m:=1, n, Loop([aux:=n, n:=m, m:=aux+n, n], m>100)]

  or

  [n:=1, m:=1, n, while m<=100 do aux:=n, n:=m, m:=aux+n, n od]

  or

  [n:=1, m:=1, n, repeat aux:=n, n:=m, m:=aux+n, n until m>100 od]

  return the list: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89] (terms of a sequence of FIBONACCI below 100).

53) LowerCase

- **LowerCase(<string>)**.
- Description: returns the <*string*> in lowercase.

54) Map

- **Map(<expression>, <variable>, <list> [, mode])**.
- Description: that function is a **loop** that build a list using the following manner: <*variable*> browses the elements of the <*list*> and for each of them, <*expression*> is evaluated, the function *Map* returns the results list. That command is the internal representation of the loop *for* (p. 28) whose usage is preferable on lisibility grounds.

  The optional parameter <*mode*> is a complex whose default value is *Nil* when <*mode*>$= a + ib$, then:

  - if a is an integer and $b = 0$: the elements in the <*list*> are handled a by a,
  - if a is an integer and $b = 1$: the <*list*> is handled by components (two components are separated with the constant *jump*) and the elements of each component are operated by complete packets of a elements, when the *jump* constant is encountered in the list, it is returned in the result. An incomplete packet is not treated.
  - if a is an integer and $b = -1$: the <*list*> is treated by component (two components are separated by the constant *jump*) and the elements of each component are treated by complete packets of a elements, when the constant *jump* is encountered in the list, it is not returned in the result. An incomplete packet is not treated.
  - if a =Re(jump): the <*list*> is treated by component (two components are separated by the constant *jump*), when the constant *jump* is encountered in the list, it is returned in the result if $b = 1$ and not returned if $b = -1$.
  - Example(s): see the loop *for* (p. 28) for examples.

- Example(s): if L is a variable containing a points list, then the command:
  - [sum:=0, Map(Inc(sum,z), z, L), sum] returns the sum of the elements of L.
  - the command Map(z*exp(i*π/3), z, L) returns the list of the images of the points of L by the rotation : center O and angle $\pi/3$.

55) Marges (margins)

- **Marges(<left>, <right>, <top>, <bottom>)**
- Description: that function sets the margins around the drawing (in centimeters).The updated values are copied in the constants *margeG*, *margeD*, *margeH* and *margeB*.

56) Merge

- **Merge(<list>)**.
- Description: the function is used to merge list pieces to obtain maximal length components, it returns the resulting list.
- Example(s): Merge([1, 2, jump, 3, 5, jump, 3, 4, 2]) returns [1, 2, 4, 3, 5]. And Merge([1, 2, jump, 3, 5, jump, 3, 4]) returns [1, 2, jump, 4, 3, 5].

  Warning: to merge two ends, they must be equal for the machine.

57) Message

- **Message(<string>)**.
- Description: that function displays the parameter <*string*> [so interpreted as a *string* (p. 29)] in a window. As soon as the user has clicked on *OK*, the window is closed and the function returns *Nil*.

58) Mix

- **Mix(<list 1>, <list 2> [, [<packet 1>, <packet 2>]])**.
- Description: that function mixes the two *<lists>* by inserting an element of the second list after each element of the first, and returns the resulting list. By default the elements are computed by packets of 1, but we can take packets of 2 or more, by editing the last argument (optional) *<[paquet 1, paquet 2]>*, if one of those numbers equals *jump*, the considered packets will be the connex components of the list.
- Example(s): Mix([1,2,3], ["a","b",jump,"c","d","e",jump,"f"], [1,jump]) give [1,"a","b",jump,2,"c","d","e",jump,3,"f"].

59) Move

- **Move(<element1>, ..., <elementN>)**.
- Description: the functions applies only on graphical elements created using the mode NotXor, corresponding to the value 1 of the variable *PenMode* (p. 35). It redraws the graphical elements *<element1>*, ..., *<elementN>*, then recalculates it, then redraws it and returns *Nil*.

  When redrawing a graphical element created using NotXor mode, it is cleared while giving back the background, then we can modify its position and redraw it. This technique allows to move objects without redrawing all the others (then avoiding the image to jump).
- Example(s): see function *Stroke* (p. 59).

60) Mtransform

- **Mtransform(<list>, <matrix>)**.
- Description: that function applies the *<matrix>* to the *<list>* and returns the result. If the *<list>* contain the constant *jump*, it is returned in the result without been modified. The *<matrice>* represents the analytic expression of an affine plane transformation, this is a three complexes list $[z1, z2, z3]$: $z1$ is the translation vector affix, $z2$ is the affix of the first column vector of the matrix of the linear part in the base (1,i), and $z3$ is the affix of the second column vector of the matrix of the linear part. For example, the identity matrix is written like this: [0,1,i] (This is the default matrix).

61) MyExport

- **MyExport(<"name">, <parameter 1>, ..., <parameter n>)** or **draw(<"name">, <parameter 1>, ..., <parameter n>)**
- Description: that command is useful to add new graphical elements with personalized export. This is described in *that section* (p. 24).

62) Nargs

- **Nargs()**.
- Description: this function has only effect within a macro, it returns the number of arguments with those the macro has been called. In all other cases, it returns *Nil*. See also the function *Args* (p. 39).
- Example(s): Here is the macro's body *MyLabel(affixe1, texte1, affixe2, texte2, ...)* taking any number of arguments:

```
for $k from 1 to Nargs()/2 do
     Label(Args(2*k-1), Args(2*k))
od
```

63) NewButton

- **NewButton(<Id>, <name>, <affix>, <size>, <command> [, help])**.
- Description: this function create in the gray area on the left in the window, a button whose identification number is the **integer** <*Id*>, the text on the button is the parameter <*name*> that is so interpreted as a *string* (p. 29), the upper left corner position is given by the parameter <*affix*> that must be in the form X+i*Y with X and Y **integers** because these are coordinates in **pixels**, the size of the button is given by the parameter <*size*> that must be in the form *len+i*height* where *len* is the button length (pixels) and *height* the height (those are also integers), the parameter <*command*> is interpreted as a string, this is the command linked to the button. Each click on the button will launch the command. The last parameter <*help*> is optional, it contains the pop-up help message when the mouse passes over the button.

  If we create a new button whose Id number already exists, then the old button is destroyed **unnless this is a predefined button** (ie: at program startup). Every file change, the non predefined button are destroyed. The function **NewButton** returns *Nil*.

64) NewGraph

- **NewGraph(<string>, <string2> [, code])**.
- Description: that function creates a *User*-defined graphical element whose name is : <*string1*> and defined by the command: <*string2*>. The two arguments are so interpreted as strings. That function returns *Nil*. If it already exists a graphical element with the same name, then the old element is overwritten. The element is created but not drawn, this is the function *ReDraw()* that is updating the display.

  The third parameter <*code*> is a positive integer (optional), a left click with the mouse on that element will launch the special macro *ClicGraph(<code>)*, That macro doesn't exists by default and can be created by the user, it is in particular created in the model file *Mouse.mod* (draw with the mouse).
- Suppose the user click on the affix point 1+i, then a dialog opens with the message *Label=* and an input line to fill. Suppose the user enter the string *Test* and validate, then the macro wil create a *user* graphical element with the name *Label1* and defined by the command Label(1+i,"Test").
- We can also use the predefined macro *NewLabel* and create the macro *ClicG()* with the code: NewLabel(%1) .
- Example(s): Here is a macro *ClicG()* that can be used to create labels "on the fly". We created before a global variable called *num* initialized to 1:

```
if Input("Label=")
then NewGraph( ["Label",num], ["Label(", %1,",", """",chaine(),""")"] ),
     ReDraw(), Inc(num,1)
fi
```

65) NewItem

- **NewItem(<name>, <command>)**.
- Description: that function adds an item called <*name*> in the pull-down list located in the gray area at the left of the window. The second parameter <*command*> is the command associated with the item. Selecting the item will launch that command. the two argument are interpreted as strings. If there already exist an item with the same name, then the old one is destroyed **unless it's a predefined item** (ie: created at program startup). Every time a file is loaded (or a new one created), non predefined items are destroyed. The function *NewItem* returns *Nil*.

66) NewMac

- **NewMac(<name>, <body> [, param1, ..., paramN])**.
- Description: that function create a macro called <*name*> whose containt is <*body*>. The two arguments are interpreted as strings. the optional parameters are variable names, the variable name <*param1*> is replaced in macro's expression with the parameter %1, the name <*param2*> is replaced with %2 ... etc. this function returns *Nil*. If it already exist a macro with the same name, then the old one is destroyed **unless it is a predefined macro**. If the name is invalid, or there is already a predefined macro with that name, or if the expression <*body*> is not correct, then the function has no effect.

67) NewVar

- **NewVar(<name>, <expression>)**.
- Description: this function create a global variable called <*name*> whose value is <*expression*>. The two arguments are interpreted as strings. The function returns *Nil*. If a variable with the same name already existed, then it is overwritten. If the name is invalid, or there already exists a constant with that name, then the function has no effect. If the <*expression*> is not correct, then the value assigned to the variable will be *Nil*.

68) Nops

- **Nops(<list>)**.
- Description: that function evaluate the <*list*> and returns the number of complexes it contains.
- Example(s): `Nops([1,2,3] )` returns `3`.

69) NotXor

- **NotXor(<element1>, ..., <elementN>)**.
- Description: this function only applies to graphical elements that were created in normal mode (ie : the variable *PenMode* (p. 35) value is 0) . It change the <*element1*>, ..., <*elementN*> elements mode into NotXor mode, recalculate them, and return *Nil*.

  When a NotXor mode graphical element is redrawn, it is cleared and give back the background. We can then change its position and redraw it. This technique allows to move objects without redrawing all the others (thus avoiding a "jumping" image)
- Example(s): See the function *Move* (p. 50).

70) OpenFile

- **OpenFile(<"file name">)**.
- Description: this function opens a file in writing mode. WARNING: If a file with the same name already exist, it will be overwritten.

  In combination with the commands *WriteFile* (p. 60) and *CloseFile* (p. 40), this allows the user to create its own text files.

71) OriginalCoord

- **OriginalCoord(<0/1>)** or **OriginalCoord()**
- Description: that function determine if the affine frame at export using *pstricks* and *tikz/pgf* equals to the one on screen(corresponds to the value 1 of the argument, default value), or not. When the argument is zero, the frame at export (*pstricks* and *tikz/pgf*) will have its origin on the bottom left, and the unit will be the centimeter on the two axes. It is usefull when we are working in frames where some numerical values are not valid for TEX.

  When the argument is 1, the point coordinates in the exported file are the same as on the screen.

  When the argument is 0, the exported point coordinates (tex, pst, tikz/pgf) of affix z on the screen are: x=Re(TexCoord(z)) and y=Im(TeXCoord(z)) (and EpsCoord instead of TeXCoord for eps export).

  When the argument is empty, the function returns the actual state of the frame (0 or 1). If not, it returns *Nil*.

72) PermuteWith

- **PermuteWith(<index list>, <list to permute>, [, packets size or jump])**
- Description: the <*list to permute*> has to be a variable, it will be permuted following the <*index list*> that is a positive integer list. The <*liste to permute*> is handled by component if it contain the constant *jump*, elements of each component are treated by packet (of 1 by default) of by complete component (ie: ended by *jump*), the list is then modified.
- Example(s)::
  - [L:=[-1,0,3,5], PermuteWith([4,3,2,1], L), L] returns [5,3,0,-1].
  - [L:=[-1,0,3,5], PermuteWith([4,3,4,1], L), L] returns [5,3,5,-1].
  - [L:=[-1,0,3,5,6,7,8], PermuteWith([4,3,2,1], L, 2), L] returns [6,7,3,5,-1,0].
  - [L:=[-1,jump,0,3,jump,5,6,7,jump,8,jump], PermuteWith([4,3,3,1,2], L, jump), L] returns [8,jump,5,6,7,jump,5,6,7,jump,-1,jump,0,3,jump].
  - [L:=[-1,jump,0,3,jump,5,6,7,jump,8], PermuteWith([4,3,3,1,2], L, jump), L] returns [5,6,7,jump,5,6,7,jump,-1,jump,0,3,jump].
  - [L:=[-1,1,5,jump,0,3,jump,5,6,7,jump,8,9], PermuteWith([2,1], L), L] returns [1,-1,jump,3,0,jump,6,5,jump,9,8].

73) ReadData

- **ReadData(<file> [, read type , separator])**.
- Description: that function opens a text <*file*> in read mode, it is supposed to contain one or more lists of numerical values and/or strings. The first argument is interpreted as a *string* (p. 29) that contains the file name (and eventually its path). The (optional) argument <*read type*> is a numerical value that can be:
  - <*read type*>=0: the function read the file as a text file, ie: only one string if no <*separator*> has been specified, else the command returns a strings list.
  - <*read type*>=1: the function read the file real by real and returns the read list (or lists): `[x1, x2, ...]`,
  - <*read type*>=2: the function read the file complex by complex, that is to say **by packets of two reals** and returns the read list (or lists) as affixes: `[x1+i*x2,x3+i*x4,...]`. This is the default value.
  - <*read type*>=3: the function read the file by packets of 3 reals (3D space points) and returns the read list (or lists) under the form: `[x1+i*x2, x3, x4+i*x5, x6, ...]`.
  - <*read type*>=4: applies to the *csv* files (values separated by a comma), those values can be numerical (reals) or alphanumerical (strings delimited by the symbol "). The <*separator*> character is by default `","`. At the end of a line the constant *jump* is inserted into the list. Empty elements will be represented in the list by the *ND* constant (ie: ND stands for Non Defined).

  In the 1, 2, 3 and 4 modes, the command can read a string if it is delimited by the character ", it will be then inserted in the list. In the modes 1, 2 and 3, a line containing # will be considered as a commentar starting from that character until the end of the line. That ending part will then be ignored.

  In the modes 1, 2, 3: the third argument <*separator*> is interpreted as a string and is supposed to contain the end list character. Between two lists, the constant *jump* will be inserted (except while reading in text mode). That argument is optional and by default there is no separator (then there is only one list). When the separator is the end of line in the file, we will use the string "LF" (*line feed*) as a third parameter. If a separator exists and when the reading is done by packets of 2 or 3 reals, an incomplete packet is ignored.
- Example(s): suppose a text file *test.dat* contains exactly this:

```
1 2 3 4 5/ 6
7 8 9 10 11/ 12
13 14 15 16 17/ 18
```

Then execution of:

- ReadData("test.dat") give: [1+2*i,3+4*i,5+6*i,7+8*i,9+10*i,11+12*i,13+14*i,15+16*i,17+18*i],
- ReadData("test.dat",1,"/") give: [1,2,3,4,5,jump,6,7,8,9,10,11,jump,12,13,14,15,16,17,jump,18],
- ReadData("test.dat",2,"/") give: [1+2*i,3+4*i,jump,6+7*i,8+9*i,10+11*i,jump,12+13*i,14+15*i,16+17*i,jump],
- ReadData("test.dat",3,"/") give: [1+2*i,3,jump,6+7*i,8,9+10*i,11,jump,12+13*i,14,15+16*i,17,jump],
- ReadData("test.dat",3,"LF") give: [1+2*i,3,4+5*i,6,jump,7+8*i,9,10+11*i,12,jump,13+14*i,15,16+17*i,18,jump].

74) ReadFlatPs

- **ReadFlatPs(<file>)**.
- Description: that function opens a <*file*> in read mode, that file is supposed to be a *flattened postscript* written file. The function returns the paths list contained in the file. The first complex of the list is *width+i*height* in cm, and the first complex of each path is *Color+i*Width* (width or thickness ?). Each path ends by a *jump* whose imaginary part is a negative integer: -1 for eofill, -2 for fill, -3 for stroke and -4 for clip.

  It is possible to convert a pdf file or a postscript file into a *flattened postscript* by using the tool *pstoedit* (http://www.pstoedit.net/). The macro *conv2FlatPs* (p. 82) is useful to do that conversion if the tool is installed in your system.

  The function *ReadFlatPs* is mainly used internally by the macro *loadFlatPs* (p. 83) that is added to the loading, adapt the point coordinates before returning the paths list that can then be drawn using the macro *drawFlatPs* (p. 82).

  That system is used by the macro *NewTeXLabel* (p. 83) to get the compiled TeX formulae.

75) ReCalc

- **ReCalc(<name1>, ..., <nameN>)** or **ReCalc()**.
- Description: that function forces the recalculation of the graphical elements whose name are in the list even if those aren't in the *Recalcul Automatique (automatic recalculation)* mode . If the list is empty (*ReCalc()*) then the whole graphic is recalculated. After that, the display is updated and the function returns *Nil*.

Warning: Using *ReCalc()* in a graphical element implies an infinite recursion then a program crash !

76) ReDraw

- **ReDraw(<name1>, ..., <nameN>)** or **ReDraw()**.
- Description: that function (re)draw the graphical elements whose name are in the list. If the list is empty (*ReDraw()*) then all the elements are redrawn. That function returns *Nil*.

77) RenCommand

- **RenCommand(<name>, <new>)**.
- Description: that function rename the command called <*name*>. The two arguments are then interpreted as strings. That function returns *Nil*. If the <*name*> is not valid, or if there isn't any command with that <*name*>, or if there aready exists a command with the name <*new*>, then the function has no effect.

78) RenMac

- **RenMac(<name>, <new>)**.
- Description: that function rename the macro called <*name*>. The two arguments are then interpreted as strings. That function returns *Nil*. If the <*name*> is invalid, or if there isn't any macro with that <*name*>, or if there already exists a macro with the name <*new*>, then the function has no effect.

79) RestoreAttr

- **RestoreAttr()**.
- Description: restore the whole set of attributes saved in a stack by the command *SaveAttr* (p. 55).

80) Reverse

- **Reverse(<list>)**.
- Description: returns the inverted <*list*>.

81) Rgb

- **Rgb(<red>, <green>, <blue>)**.
- Description: that function returns an integer representing the color whose three components are<*red*>, <*green*> et <*blue*>, those three values must be numbers **between 0 and 1**. See the command *HexaColor* (p. 46).
- Example(s): Color:= Rgb(0.5,0.5,0.5)) select the gray.

82) SaveAttr

- **SaveAttr()**.
- Description: save on a stack the whole set of current attributes. See also *RestoreAttr* (p. 55).

83) ScientificF

- **ScientificF(<real> [, <decimal places>])**.
- Description: converts the <*real*> into scientific format and returns the result as a string.

84) Seq

- **Seq(<expression>, <variable>, <start>, <end> [, step])**.
- Description: that function is a **loop** that builds a list following the rule: <*variable*> is initialized to <*start*> then, while the <*variable*> is in the (closed) interval defined by <*départ*> and <*end*>, we evaluate the <*expression*> and increment <*variable*> by the <*step*> value. The step can be negative, but can't be zero. If the step is not given, its default value is 1. When the <*variable*> is out of the interval, the loop stops and the function *Seq* returns the results list. That command is the inner representation of the loop *for* (p. 28) whose usage is preferable for lisibility reasons.
- Example(s): Seq(exp(i*k*pi/5,k,1,5) returns the fifth roots of the unity. The command:

  Ligne(Seq(exp(2*i*k*pi/5, k, 1, 5), 1)

  will return the value *Nil* but will draw a pentagon (see *Ligne* (p. 90)) if it is used in a *user-defined* graphical element.

85) Set

- **Set(<variable>, <value>)**.
- Description: that function assign to <*variable*>[3] the <*value*>. The function *Set* returns *Nil*.

  That command is the internal representation of the assignement:= whose usage is preferable on lisibility grounds.

[3] It is not necessary to declare the variables, they are implicitly locally declared and set to *Nil* unless this is the name of a global variable or a predefined constant like (comme i, π, e, ...).

86) SetAttr

- **SetAttr()**.
- Description: that function has only effects on a user-defined graphical element. It assigns to element's attributes the current attributes values. The function *SetAttr* returns *Nil*.

87) SetMatrix

- **SetMatrix(<[z1, z2, z3]>)**.
- Description: that function modifies the current matrix (The matrix affects all the graphical elements except axes and grids in the actual version). That matrix represents the analytic expression of an affine plane map, this is a three complex list: $z1$ is the translation vector affix, $z2$ is the matrix's first column vector affix of the linear part in the base (1,i), and $z3$ is the matrix's second column vector of the linear part. For example, the identity matrix is written: [0,1,i] (this is the default matrix). (See also *GetMatrix* (p. 45), *ComposeMatrix* (p. 40), and *IdMatrix* (p. 46))
- Example(s): si $f : z \mapsto f(z)$ is an affine map, then its matrix is $[f(0), f(1)-f(0), f(i)-f(0)]$, the calculation can be done by the macro *matrix()* from TeXgraph.mac: SetMatrix(matrix(i*bar(z))) affects the orthogonal symmetry matrix relative to the first bisecting.

```
\begin{texgraph}[name=SetMatrix, file]
Graph image = [
view(-5,5,-3,3), size(7.5),
SetMatrix([0,1,1+i]), axes(0,1+i),
tMin:=-5, tMax:=5,
Color:=red, Width:=8, Cartesienne(2*sin(x)),
Color:=black, Arrows:=2,
tangente(2*sin(x), pi/2, 1.5),
Arrows:=0, LineStyle:=dotted,
Ligne( [2*i,pi/2+2*i, pi/2], 0),
Point(pi/2+2*i),
LabelDot( pi/2, "$\frac{\pi}2$","S",1),
IdMatrix()
];
\end{texgraph}
```

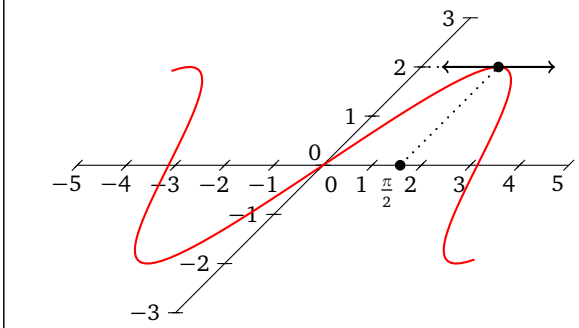


Figure 2: *Non orthogonal frame*

88) Show

- **Show(<element1>, ..., <elementN>)**.
- Description: that function shows the graphical elements called <*element1*>, ..., <*elementN*> by setting their attribute *IsVisible* to true. The arguments are interpreted as strings.that function returns *Nil*. If we use the command without any arguments : *Show()*, everything is shown. If we want to show everything but one or several elements, we use: Show(except, element1, ..., elementN). See also the command *Hide* (p. 46).

89) Si (if)

- **Si(<condition1>, <expression1>, ..., <conditionN>, <expressionN> [, else])**.
- Description: the function evaluates <*condition1*>. A condition is an expression whose result must be 0 [for *False*] or 1 [for *True*], else it fails and the function returns *Nil*. If the condition returns 1 then the function avaluate the <*expression1*> and returns the result, If it is 0, it evaluate the <*condition2*>, if it returns 1 then the function evaluate the <*expression2*>, else ...etc. When none of the conditions is true, the function evaluate the argument <*else*>, if there is one, and returns the result else the function returns *Nil*. That function is the internal representation of the alternative *if* (p. 28) whose syntax is preferable for lisibility reasons.
- Example(s): definition of a macro f(x) representing a piecewise function f of a variable x :

  Si(%1<-1,1-exp(pi*(%1+1)),%1<0,sin(π*%1),sh(pi*%1)),

  ie: $f(x) = 1 - exp(\pi(x+1))$ if $x < -1$, $f(x) = sin(\pi x)$ if $-1 \leqslant x < 0$, $f(x)$ =sh(πx) if not.

90) Solve

- **Solve(<expression>, <variable>, <lower bound>, <upper bound> [, n]).**
- Description: that function "solve" the equation <*expression*>=0 in respect with the **real** <*variable*> in the interval defined by the <*lower bound*> and <*upper bound*>. That interval is divided into <*n*> parts [n=25 by default] and the NEWTON method is used on each part. The function returns the results list.
- Example(s):
  - Solve(sin(x), x, -5, 5) give [-3.141593, 0, 3.141593].
  - Equation: $\int_0^x \exp(t^2)dt = 1$: Solve(Int(exp(u^2),u,0,x)-1, x, 0, 1) give 0.795172 and executing Int(exp(u^2), u, 0, 0.795172) give 1.
  - Solve(x^2+x+1, x, -1, 1) returns *Nil*.

91) Sort

- **Sort(<complexes list> [, option]).**
- Description: the function sorts the given list in lexicographical order, if the argument <*option*> is 0 (default value), or in the reverse lexicographical order if the argument <*option*> is 1. That list must be a variable, it will be modified.If the list contain a *jump* constant, then it is copied as is in the result, and the “connex components” of the list are each other separately sorted. The function returns *Nil*.
- Example(s): If the variable *L* contains the list [-2,-3+i,1,1-2*i, jump, 3,5,-6] then after the execution of Sort(L), the variable will contain the list [-3+i,-2,1-2*i,1,jump,-6,3,5], and after the execution of Sort(L,1), the variable will contain the list [1,1-2*i,-2,-3+i,jump,5,3,-6]. The method used is the Quick Sort..

92) Special

- **Special(<string>).**
- Description: that function has only effects in a User graphical element (like graphical functions). The argument must be a string (delimited by " and "), If that string contains the tags \[, and \], then the text between those tags is intepreted by TeXgraph and the result inserted in the string. Then that string will be written as is in the export file (In fact this is a Label created with LabelStyle= special).
- Example(s): Special("\psdot(\[1+\],\[2^3\])"), will write in the export file: \psdot(2,8).

93) Str

- **Str(<macro name>).**
- Description: when alphnumerical evaluation, that function returns the definition of the macro called <*macro name*> (unless this is a predefined macro). Outside that context, the function *Str* returns *Nil*. The argument <*macro name*> is itself interpreted as a string.
- Example(s): suppose the macro f is defined by %1+i*sin(%1), then the command Label(0,["f(%1)=",Str("f")]) will display on screen at affix 0, the text: : f(%1)=%1+i*sin(%1).

94) StrArgs

- **StrArgs(<integer>).**
- Description: that function has only effects within a macro , it returns the argument number <*integer*> of the calling macro, as a string. Outside this context, it returns *Nil*. See also the command *Args* (p. 39).
- Example(s): see the function *Nargs* (p. 50).

95) StrComp

- **StrComp(<string1>, <stringe2>)**.
- Description: the two arguments are interpreted as a string, and the strings are been compared, if they are equal, then the function returns 1, 0 if not. Since release 1.97, it is possible to compare strings with the symbol =.
- Example(s): the key combination : Ctrl+Maj+<letter> automatically launch the execution of the special macro: *OnKey(<letter>)*. The user can define that macro with for example that command:

```
if StrComp(%1, "A") then Message("Letter A") fi
```

96) StrCopy

- **StrCopy(<string>, <start index>, <quantity>)**.
- Description: returns the string resulting of the extraction (works like the command *Copy* (p. 40)).

97) StrDel

- **StrDel(<variable>, <start index>, <quantity>)**.
- Description: modifies the <*variable*> by removing <*quantity*> characters from the <*start index*> (works like the command *Del* (p. 41)). If the <*variable*> contains a strings list, only the first one is modified. If the <*variable*> doesn't contain any string, the command has no effect.

98) StrEval

- **StrEval(<expression>)**.
- Description: that function evaluates the <*expression*> (that must be a string), and returns the result as a string.

99) String

- **String(<mathematical expression>)**.
- Description: when an alphanumerical evaluation is done, that function returns the <*mathematical expression*> as a string. Outside that context, the function *String* returns *Nil*.

100) String2Teg

- **String2Teg(<expression>)**.
- Description: that function does an alphanumerical evaluation of the <*expression*> and returns the result as a string doubling all the " encountered characters. The resulting string is then readable by TeXgraph.

101) StrLength

- **StrLength(<string>)**.
- Description: returns the number of characters in the string.

102) Stroke

- **Stroke(<element1>, ..., <elementN>)**.
- Description: that function recalculate the graphical elements : <*element1*>, ..., <*elementN*>, then redraw them using NORMAL mode and returns *Nil*.
- Example(s): we created two global variables: *a* and *drawing*. We will create the circle with center a and radius 1 called *object1*, we want to be able to move it with the mouse. To do that, we create the macro *ClicG* with the command:

```
[PenMode:=1, {NotXor mode}
 NewGraph("object1", "Cercle(a,1)"),
 PenMode:=0, {normal mode}
 ReDraw(), {we show}
 drawing:=1]
```

we create the macro MouseMove with the command: if drawing then a:=%1 {we move the center}, Move(object1) fi,
then the macro LButtonUp with the command: if drawing then Stroke(object1), drawing:=0 fi.
The macro *ClicG* creates the object1 in NotXor mode, update the graphical display and switch to the "drawing" mode. The macro *MouseMove* places the center a at the mouse place, then moves the object1. When the left button is released, we draw the object1 in normal mode and quit the "drawing" mode.

103) StrPos

- **StrPos(<pattern>, <string>)**.
- Description: returns the position (integer) of the first pattern encountered in the string.

104) StrReplace

- **StrReplace(<string>, <pattern to be removed>, <replacement pattern>)**.
- Description: returns the string resulting of the replacement.

105) TeX2FlatPs

- **TeX2FlatPs(<"formula"> [, dollar(0/1)])**.
- Description: that command returns a TEX <*formula*> as a paths list, the result has to be drawn with the macro *drawFlatPs* (p. 82). The <*formula*> is written in a file called *formula.tex*. That file is called by the file *TeX2FlatPs.tex* that is located in the TeXgraph working directory, to be compiled by TEX. If the option <*dollar*> is 1 then the formula will be delimited by \] and \], else it is written as is. For further details about how these paths are build, see the command *ReadFlatPs* (p. 54).

106) Timer

- **Timer(<milli-seconds>)**.
- Description: sets the time interval for the timer, the one that is regularly executes a particular macro (it can be defined with the command TimerMac). To stop the timer, you only have to set the time interval to 0.

107) TimerMac

- **TimerMac(<body of the macro to execute>)**.
- Description: that command create a macro attached to the timer. The argument is interpreted as a string and must correspond to the macro body (it will be called TimerMac). For performances reasons, it is preferable to avoid too many calls to other macros from this one. That function returns 1 if the macro is valid, 0 if an error occurs. Warning, executing TimerMac don't start the timer! Use the Timer command to do that.
- Example(s): let be *A* a global variable (a point), and *dotA* a graphical element that draws the point. Here is a command that moves *A*:

  [TimerMac("[Inc(A,0.1), if Re(A)>5 then Timer(0) else ReCalc(dotA) fi]"), A:=-5, Timer(10)]

108) UpperCase

- **UpperCase(<string>)**.
- Description: returns the <*string*> to uppercase.

109) VisibleGraph

- **VisibleGraph(<0/1>)** or **VisibleGraph()**.
- Description: that function activate or desactivate the drawing area in the graphic interface. When it is desactivated, its content do not change anymore because it the area is not updated. Desactivation of the graphical display can, in some cases, permit for example to save time to record an animation.

  When the argument is empty, the function simply returns the actual graphical display state (0 or 1). Else, it returns *Nil*.

110) WriteFile

- **WriteFile(<argument>)** or **WriteFile(<file name>, <argument>)**.
- Description: in its first version, the function write either in the last text file opened by the command *OpenFile* (p. 52), or in the export file if the execution of that command takes place while an export with the macros *Bsave* (p. 107) and/or *Esave* (p. 107). In the second version, the writing is done in the given name file, if it is already opened. The two arguments are alphanumerically evaluated.
- Example(s): here is how the macro *Bsave* could look like to modify the arrows sizes using pstricks:

```
if ExportMode=pst then WriteFile("\psset{arrowscale=3}") fi
```

Chapter VI

Mathematical functions and operations

1) Operations

1.1 Usual operations

- These are the operations: *+, -, *, /*. Those symbols are mandatory in the expressions, for example: 2x instead of 2*x will generate an error.
- We can add two lists: [1,2,3]+[4,5] will give [5,7,3].
- We can subtract two lists: [1,2,3]-[4,5,6,7] will give [-3,-3,-3,-7].
- A list may be multiplied by a complex: 5*[1,2,3] will give [5,10,15], but [1,2,3]*5 will give 5.
- A list may be divided by a complex: [1,2,3]/2 will give [0.5,1,1.5].
- The operation x^y is corresponding to the power function x^y. The exponent has to be real, but when x is a non real complex, y must be an integer.

1.2 Logic operations

- These are the operations **And** and **Or**, the boolean values True and False corresponding respectively to the numerical values 1 and 0. The macro *not()* (p. 65) take the negation of an expression.
- Example(s): 1 And 0 gives 0, but 2 Or 1 gives *Nil*.

1.3 Comparisons

These are operations whose result is a boolean value (0 or 1). Here is the list:

- **Egal** (or =): tests if two objects are equal. Those objects can be a list or the *Nil* value.
- **Negal** (ou <>): tests if two objects are not equal.Those objects can be a list or the *Nil* value.
- **Inf** (or <): Tests the "strictly less than" relation (between two reals).
- **InfOuE** (or <=): tests the "less than or equal to" relation.
- **Sup** (or >): tests the "strictly greater than".
- **SupOuE** (or >=): tests the "greater than or equal to" relation.
- **Inside**: tests if the first argument (that must be an point's affix) is inside (edge excluded) the polygon given by the second argument (that must be a closed list).
- Example(s): 1 Inside [-1,2+3*i,4-i,-1] give 1 and i Inside [-1,2+3*i,4-i,-1] give 0.

1.4 Intersection operations

There are two of them:

- **Inter**: the two arguments must be lists with two elements (there can be more elements, but only the two first will be handled), that will be interpreted as two straight lines [defined by two points], the operation determine and returns the intersection point. When the two lines are parallel, the result is *Nil*.
- **InterL**: the two arguments must be lists with at least two elements, then interpreted as two polylines, the *InterL* operation determine and returns the points list that intersects the two lines. **The intersection's points are sorted following the same way (direction) as the first argument** (and if several points are on the same segment, then they are sorted following the direction of travel of the second argument).

1.5 Cut operation

There are two cut operations:

- **CutA**: (cut after) the first argument must be a list and the second, a complex (that should be on the polyline given by the list's points). The *CutA* operation returns the list's points located **before the complex**.

  Example(s): [1,2,3,4,5] CutA 3.5 gives [1,2,3,3.5] and [1,2,3,4,5] CutA 6 gives *Nil*.
- **CutB**: (cut before) the first argument must be a list and the second, a complex (that should be on the polyline given by the list's points). The *CutB* operation returns the list's points located **after the complex**.

2) The predefined mathematical functions

These are functions of a **real** or **complex** variable that are returning a complex.

2.1 abs

- **abs(<argument>)**.
- Description: this is the modulus of a complex number.

2.2 arccos, arccsin, arctan, arccot

- **arccos(<argument>)**, **arcsin(<argument>)**, **arctan(<argument>)** and **arccot(<argument>)**.
- Description: there are the usual inverse circular functions of a real variable.

2.3 Arg

- **Arg(<argument>)**.
- Description: this is the main argument of a complex number (in the interval]-π;π]).

2.4 argch, argsh, argth, argcth

- **argch(<argument>)**, **argsh(<argument>)**, **argth(<argument>)** and **argcth(<argument>)**.
- Description: these are the usual inverse hyperbolic functions of a real variable.

2.5 bar

- **bar(<argument>)**.
- Description: this is the conjugate of a complex number.

2.6 ch, cos

- **ch(<argument>)** and **cos(<argument>)**.
- Description: trigonometric and hyperbolic cosinus of a real variable.

2.7 Ent

- **Ent(<argument>)**.
- Description: this the integer part of a real variable.

2.8 exp

- **exp(<argument>)**.
- Description: this is the exponential function of a **complex** variable.

2.9 Im

- **Im(<argument>)**.
- Description: this is the imaginary part of a **complex** number.

2.10 ln

- **ln(<argument>)**.
- Description: this is the natural logarithm, the argument is a real.

2.11 M

- **M(<a>, <b>)** or **M(<a>, <b>, <c>)**.
- Description: the arguments are reals, that function returns the a+ib complex or the space point [a+ib,c]. The main advantage of that function is that the coding takes less place in memory.

2.12 opp

- **opp(<argument>)**.
- Description: opposite function, the argument is a **complex**.

2.13 Rand

- **Rand([argument])**.
- Description: that function gives a random number: if there is no <*argument*> (*Rand()*) then the returned value is a number of the interval [0;1[, else the returned value is an integer between 0 and the absolute value of the <*argument*> (excluded).
- Example(s): Rand(256) returns an integer between 0 and 255.

2.14 Re

- **Re(<argument>)**.
- Description: returns the real part of the given **complex**.

2.15 Round

- **Round(<complex> [, decimal places])**.
- Description: that function returns the <*complex*> by rounding to the nearest the real and imaginary parts with respect to the given decimal places (0 by default).

2.16 sh, sin

- **sh(<argument>)** and **sin(<argument>)**.
- Description: hyperbolic and trigonometric sinus of the real variable.

2.17 sqr

- **sqr(<argument>)**.
- Description: square function. The argument is a **complex**.

2.18 sqrt

- **sqrt(<argument>)**.
- Description: square root function. The argument is a real.

2.19 tan, th, cot, cth

- **tan(<argument>)**, **th(<argument>)**, **cot(<argument>)** and **cth(<argument>)**.
- Description: trigonometric tangent, hyperbolic tangent, trigonometric cotangent and hyperbolic cotangent. The argument is real.

Chapter VII

Mathematical macros from TeXgraph.mac

1) Arithmetic and logic operations

1.1 Ceil

- **Ceil(<x>)**.
- Description: returns the lowest integer that is greater than or equal to the real <x>.

1.2 div

- **div(<x>, <y>)**.
- Description: returns the only k integer so that x-ky is in the interval $[0;|y|[$.

1.3 mod

- **mod(<x>, <y>)**.
- Description: returns the only integer r in the interval [0;|y|[so that x=ky+r with k integer.

1.4 not

- **not(<boolean expression>)**
- Description: returns the boolean value of the negation.

1.5 pgcd (gcd)

- **pgcd(<a>, <b> [, <u>, <v>])**
- Description: returns d the value of the gcd of <a> and <b>, and assigns two Bézout coefficients to the variables <u> and <v> (if those arguments have been given), so that $au + bv = d$.

1.6 ppcm (lcm)

- **ppcm(<a>, <b>)**
- Description: returns the lcm of <a> and <b>.

2) Operations on the variables

2.1 Abs

- **Abs(<affix>)**.
- Description: that macro gives the norm in cm.

2.2 free

- **free(<x>).**
- Description: frees the variable <*x*> by assigning it to *Nil*. From version 1.93 direct assignment to *Nil* is possible, for example : x:=Nil.

2.3 IsIn

- **IsIn(<affix> [, <epsilon>]).**
- Description: returns 1 if the <*affix*> is in the graphical window, 0 if not. That macro takes in count the current matrix, the test is made within <*epsilon*> cm. By default <*epsilon*> is set to 0.0001 cm.

2.4 nil

- **nil(<x>).**
- Description: returns 1 if the variable <*x*> is *Nil*, 0 if not. Since version 1.93 it is possible to directly test if <*x*> is equal to the *Nil* constant.

2.5 round

- **round(<list> [, decimal places])**
- Description: cut the complexes from the <*list*> by rounding to the nearest number the real and imaginary part with the given <*decimal places*> (0 by default). If the <*list*> contain the constant *jump*, then it is returned in the results's list. That macro uses the command *Round* (p. 64) (which applies only to a complex, not a list).

3) Operations on the lists

3.1 bary

- **bary(<[affix1, coef1, affix2, coef2, ...]>).**
- Description: returns the centroid of the weighted system: <*[(affix1, coef1), (affix2, coef2),...]*>.

3.2 del

- **del(<list>, <indexes's list to remove>, <remove quantity>).**
- Description: returns the list after removing the elements whose index are in the <*indexes's list to remove*>. The <*remove quantity*> (each time) is 1 by default, that argument can also be a list.
- Example(s):
  - del([1,2,3,4,5,6,7,8], [2,6,8]) gives [1,3,4,5,7].
  - del([1,2,3,4,5,6,7,8], [2,6,8], 2) gives [1,4,5].
  - del([1,2,3,4,5,6,7,8], [2,6,8], [1,2]) gives [1,3,4,5].
  - del([1,2,jump,3,4,5,jump,6,7,8],[3,7]) gives [1,2,3,4,5,6,7,8].

3.3 getdot

- **getdot(<s> , <polyline>).**
- Description: returns the point from the <*polyline*> where <*s*> is its curvilinear abscissa. The <*s*> parameter must be in the interval [0; 1], 0 for the first point, and 1 for the last.

3.4 IsAlign

- **IsAlign(<2D points list> [, epsilon]).**
- Description: returns 1 if the points are on the same straight line, 0 if not. Default tolerance <*epsilon*> is 1E-10. The <*list*> must not contain the *jump* constant.

3.5 isobar

- **isobar(<[affix1, affix2, ...]>).**
- Description: returns the isobarycenter of the system <*[affix1, affix2, ...]*>.

3.6 KillDup

- **KillDup(<list> [, epsilon]).**
- Description: returns the list without duplicates. Comparisons tolerance is <*epsilon*> (0 by default).
- Example(s): KillDup([1.255,1.258,jump,1.257,1.256,1.269,jump] ,1.5E-3) returns [1.255,1.258,1.269].

3.7 length

- **length(<list>).**
- Description: returns the <*list*> length in cm.

3.8 permute

- **permute(<list>).**
- Description: moves the first element of the <*list*> to the end. The <*list*> must be a variable.
- the command [x:= [1,2,3,4], permute(x), x] returns [2,3,4,1].

3.9 Pos

- **Pos(<element>, <list>, [, epsilon]).**
- Description: returns the positions's list of the <*element*> (string or complex) in the <*list*>, comparison tolerance is <*epsilon*> for numerical values, (<*epsilon*> is 0 by default).
- the command Pos(2, [1,2,3,2,4]) returns [2,4], but Pos(5, [1,2,3,2,4]) returns *Nil*.

3.10 rectangle

- **rectangle(<list>).**
- Description: determines the smallest rectangle containing the list. The macro returns a two complexes list representing the bottom left corner affix followed by its upper right corner.

3.11 replace

- **replace(<list>, <position>, <new>).**
- Description: change the <*list*> by replacing the element number <*position*> by the <*new*>, that macro returns *Nil*.

3.12 reverse

- **reverse(<list>).**
- Description: returns the list after inverting each component (two components are separated by *jump*).

3.13 SortWith

- **SortWith(<keys list>, <list>, <packets size> [, mode])**.
- Description: sorts the **variable** <*list*> following the <*keys*>). Elements in the <*list*> are handled by <*packets*>, the element number *i* in the keys list, is the key of the number *i* packet from the <*list*>. The <*packet size*> is 1 by default; it can be *jump* then each packet handled is a whole component. If a packet is not complete, it is not treated. If the list contains the constant *jump*, then all the components are also sorted. The last parameter is the sort type: <*mode*>=0 for ascending order (default value), <*mode*>=1 for descending order.

4) Handling lists by components

The adopted convention is that two components are separated by the *jump* constant. A component can be empty.

4.1 CpCopy

- **CpCopy(<list>, <start index>, <number>)**.
- Description: that function returns the list built with the <*number*> components of the <*list*> from the component number <*start*> [included]. If <*number*> is 0, then the function returns all the components in the list from the component number <*start*>.

  If the <*start*> number is negative, then the list is read from the right to the left starting from the last, the last component's index is -1, the penultimate's is $-2\cdots$ etc. The function returns the <*number*> components from the list (or the whole list if <*number*> is 0) read towards left, but the list is returned in the same direction as the <*list*> (that list is unchanged).
- Example(s):
  - CpCopy([1,2,jump,3,7,8,jump,4,jump,5,6], 2, 1) returns [3,7,8].
  - CpCopy([1,2,jump,3,7,8,jump,4,jump,5,6], -1, 2) returns [4,jump,5,6].
  - CpCopy([1,2,jump,3,7,8,jump,4,jump,5,6], -3, 0) returns [1,2,jump,3,7,8].

4.2 CpDel

- **CpDel(<variable list>, <start index>, <number>)**.
- Description: that function removes in the <*list*>, the <*number*> components from the number <*start*> [included] component .If <*number*> is 0, then the function removes all components in the <*list*> from the number <*start*> component.

  If the number <*start*> is negative, then the list is read from the right to the left starting from the last component. The last component's index is -1, the penultimate's is $-2\cdots$ etc. The function removes the <*number*> components in the <list> (or the whole list if <*number*> is 0) towards left.

  The parameter <*list*> must be a **variable name**, it is modified and the macro returns *Nil*.

4.3 CpNops

- **CpNops(<list>)**.
- Description: that function evaluates the argulist and returns the number of components in that list.
- Example(s):
  - CpNops([1,2,jump,3]) returns 2.
  - CpNops([1,2,jump,3,jump]) returns 3.
  - CpNops([jump]) returns 2.

4.4 CpReplace

- **CpReplace(<variable list>, <position>, <new>).**
- Description: modifies the <*list*> by replacing the component number <*position*> by <*new*>, the macro returns *Nil*.

4.5 CpReverse

- **CpReverse(<list>).**
- Description: returns the <*list*> with the components in the reverse order.
- Example(s): CpReverse([1,2,jump,3,4,5,jump]) returns [jump,3,4,5,jump,1,2].

5) Managing string lists

This part is still here for backward compatibility because since TeXgraph 1.97, strings are natively handled. In the old releases of TeXgraph, such a list was a **macro**, elements are indexed from 1, and the string number *k* is given by `@ListName(k)`, and the list length is given by `ListName(0)`.

5.1 StrListInit

- **StrListInit(<ListName>, <"string1">, <"string2">, ...).**
- Description: create a list of strings called <*ListName*> (ListName is a macro), the following arguments are interpreted as strings, are the elements of the list, indexed from 1.
- Example(s): With the command StrListInit(test, "toto", ["toto",2/4], 24), a macro called **test** is created and contains:

```
for $z in Args() do
  if z<0 then Inc(z,4) fi,
  if z=0 then 3
  elif z=1 then "toto"
  elif z=2 then "toto0.5"
  elif z=3 then "24"
  fi
od
```

- Example(s): Display labels with an affix and a direction for each, by using LabelDot.

```
\begin{texgraph}[name=StrListInit, file]
Graph image = [
 view(-2,2,-2,2), Marges(0,0,0,0), size(7.5),
 StrListInit( name, -1,"$A$","O", i,"$B$","N",
          1,"$C$","E", -i,"$D$","S"),
 for k from 1 to name(0) step 3 do
   LabelDot( Eval(@name(k)), @name(k+1), @name(k+2), 1)
 od,
 StrListKill(name)
];
\end{texgraph}
```

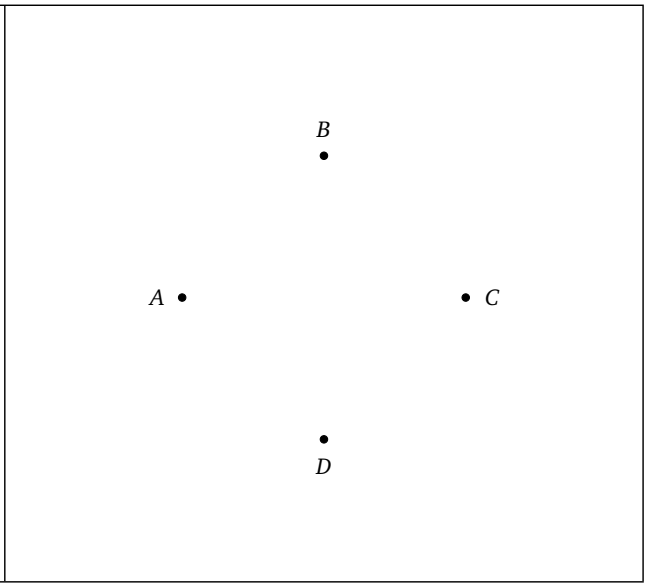


Figure 1: *StrListInit Usage*

Another solution is to do three lists: name, position, orientation:

```
view(-2,2,-2,2), Marges(0,0,0,0), size(7.5),
StrListInit( name, "$A$", "$B$", "$C$", "$D$"),
StrListInit(orientation, "O", "N", "E", "S"),
position:=[-1, i, 1, -i],
for k from 1 to name(0) do
    LabelDot( position[k], @name(k), @orientation(k), 1)
od,
StrListKill(name, orientation)
```

5.2 StrListAdd

- **StrListAdd(<ListName>, <"string1">, <"string2">, ...)**.
- Description: that macro adds at the end of the string list called <*ListName*>, the following arguments (interpreted as strings). The list <*ListName*> must already exist. In fact, that list is a macro that will be entirely overwritten to add the other elements. It is faster to create the list in one time using the macro *StrListInit* (p. 69) when it's possible.

5.3 StrListCopy

- **StrListCopy(<ListName>, <NewList> [, start index, number])**.
- Description: that macro creates a new string list called <*NewList*> by copying <*number*> elements from the <*NameList*> starting at <*start index*>. The argument <start index> can be negative (−1 is the last element's index, −2 the penultimate's, ...). The elements are always read from the left to the right when <number> is positive, in the inverse direction when <number> is negative. By default, the <*start index*> is 1 and the <*number*> is 0 (that means "all the elements").

5.4 StrListDelKey

- **StrListDelKey(<ListName>, <start index>, <number>)**.
- Description: That macro removes <*number*> elements from the <*ListName*> starting at <*start index*>. Like with the Del command, the argument <*start index*> can be negative (−1 indexes the last element, −2 the penultimate, ...). The elements are always read from the left to the right when <number> is positive, and in the reverse direction when <number> is negative. That macro returns *Nil*.

5.5 StrListDelVal

- **StrListDelVal(<ListName>, <val1>, <val2>, ...)**.
- Description: that macro removes from the <*ListName*> the strings <*val1*>, <*val2*>..., without giving its indexes.

5.6 StrListGetKey

- **StrListGetKey(<ListName>, <string>)**.
- Description: that macro returns the <*string*>'s index in the list <*ListName*>, if it's not, then the macro returns *Nil*.

5.7 StrListInsert

- **StrListInsert(<ListName1>, <string> [, <index>])**.
- Description: that macro modifies the string list <*ListName*>, by inserting a new <*string*> at position <*index*>. By default <*index*> is 0 (end of the list), the value can be negative (−1 is the last element's index, −2 the penultimate's, ...).

5.8 StrListKill

- **StrListKill(<ListName1>, <ListName2>, ...)**.
- Description: the macro removes the string lists <*ListName1*>, <*ListName2*>, ...

5.9 StrListReplace

- **StrListReplace(<ListName>, <old string>, <new>)**.
- Description: That macro replaces the <*old string*> by the <*new*> in the list <*ListName*>.

5.10 StrListReplaceKey

- **StrListReplaceKey(<ListName>, <index>, <new string>)**.
- Description: That macro modifies the list called <*nomListe*>, by replacing the string number <*index*>, with the <*new string*>.

5.11 StrListShow

- **StrListShow(<ListName> [, <start index>, <number>])**.
- Description: that macro returns the string obtained by copying <*number*> elements from the list <*ListName*> starting at <*start index*>, without concatenation. The resulting string is returned using the form: `"string", "string", ...`. The argument <start index> can be negative (-1 is the last element's index, -2 the penultimate's, ...). The elements are always read from the left to the right when <*number*> is positive, and in the reverse direction when <number> is negative. By default the <start index> is 1 and the <*number*> is 0 (meaning "all elements").

6) Statistical functions

6.1 Anp

- **Anp(<n>, <r>)**.
- Description: returns the permutations of <*r*> objects taken among <*n*> different objects (nPr).

6.2 binom

- **binom(<n>, <p>)**.
- Description: returns the binomial coefficient <*p*> among <*n*>.

6.3 ecart

- **ecart(<reals list>)**.
- Description: returns the standard deviation of the <reals list>, strings and the constant *jump* are ignored.

6.4 fact

- **fact(<n>)**.
- Description: returns the value of $n!$ (factorial function).

6.5 max

- **max(<complexes list>)**.
- Description: returns the greatest element from a numerical list (lexicographical order), strings and *jump* constant are ignored.

6.6 min

- **min(<complexes list>)**.
- Description: returns the lowest element from a numerical list (lexicographical order), strings and *jump* constant are ignored.

6.7 minmax

- **minmax(<complexes list>)**.
- Description: returns the lowest and the greatest element from a numerical list (lexicographical order), strings and *jump* constant are ignored.

6.8 median

- **median(<complexes list>)**.
- Description: returns the median element from a numerical list (lexicographical order), strings and *jump* constant are ignored.

6.9 moy

- **moy(<complexes list>)**.
- Description: returns the average of a numerical list, strings and the *jump* constant are ignored.

6.10 prod

- **prod(<complexes list>)**.
- Description: returns the element's product from a numerical list, strings and *jump* constant are ignored.

6.11 sum

- **sum(<complexes list>)**.
- Description: returns the elements's sum from a numerical list, strings and *jump* constant are ignored.

6.12 var

- **var(<reals list>)**.
- Description: returns the variance of a numerical list, strings and *jump* constant are ignored.

7) Conversion functions

7.1 Anchor

- **Anchor(<position>)**
- Description: returns the point's affix in the current graphical window, according to the <*position*>:
  - <*position*>=*center* (or *c*, default value), This is the window's center,
  - <*position*>=*top* (or *t*), this is the top middle of the window,
  - <*position*>=*bottom* (or *b*), this is the bottom middle of the window,
  - <*position*>=*left* (or *l*), this is the middle of the left side of the window,
  - <*position*>=*right* (or *r*), this is the middle of the right side of the window,
  - <*position*>=*top+right* (or *tr*), this is the top right corner of the window,
  - <*position*>=*top+left* (or *tl*), this is the top left corner of the window,
  - <*position*>=*bottom+right* (or *br*), this is the bottom right corner of the window,
  - <*position*>=*bottom+left* (or *bl*), this is the bottom left corner of the window.

7.2 RealArg

- **RealArg(<affix>)**
- Description: returns the argument (radians) of the real affix of a vector. The function takes the current matrix in count.

7.3 RealCoord

- **RealCoord(<screen affix>)**
- Description: returns the point's real affix given the scales and the current matrix.

7.4 RealCoordV

- **RealCoordV(<screen affix>)**
- Description: returns the vector's real affix given the scales and the current matrix.

7.5 ScrCoord

- **ScrCoord(<real affix>)**
- Description: returns the screen affix of a point given the scales and the current matrix.

7.6 ScrCoordV

- **ScrCoordV(<real affix>)**
- Description: returns the screen affix of a vector given the scales and the current matrix.

7.7 SvgCoord

- **SvgCoord(<screen affix>)**
- Description: returns the svg format affix given the scales and the current matrix.

7.8 TeXCoord

- **TeXCoord(<screen affix>)**
- Description: returns the affix exported in tex, pst and pgf given the scales and the current matrix. For eps, there is the command *EpsCoord* (p. 43).

8) Plane geometric transformations

8.1 affin

- **affin(<list> , <[A, B]>, <V>, <lambda>)**
- Description: returns the point's images list of the <*list*> by scaling in the vector <*V*> direction with respect to the straight <*(AB)*> direction , using factor <*lambda*>.

8.2 defAff

- **defAff(<name>, <A>, <A'>, <linear part>)**
- Description: that function creates a macro called <*name*> representing the affine map transforming <*A*> into <*A'*>, whose linear part is the last argumeent. That linear part is a two complexes list: [Lf(1), Lf(i)]. Lf is the linear part of the transformation.

8.3 ftransform

- **ftransform(<list>, <f(z)>)**
- Description: returns the images list of the points from the <*list*> by the function <*f(z)*>, this can be an expression function of z or a macro with argument z.

8.4 hom

- **hom(<list>, <A>, <lambda>)**
- Description: returns the images's list of points from the <*list*> by the homothety with center <*A*> and ratio <*lambda*>.

8.5 inv

- **inv(<list>, <A>, <R>)**
- Description: returns the image's list of the points from the <*list*> by the inversion with reference circle: center <*A*> and radius <*R*>.

8.6 mtransform

- **mtransform(<list>, <matrix>)**
- Description: returns the images list of the points from the <*list*> by the affine map f defined by the <*matrix*>. that *matrix* (p. 75) is like : [f(0), Lf(1), Lf(i)] with the linear part Lf.

8.7 proj

- **proj(<list>, <A>, <B>)** or **proj(<list>, <[A,B]>)**
- Description: returns the orthogonal projections of the points from the <*list*> on the straight line (*AB*).

8.8 projO

- **projO(<list>, <[A,B]>, <vector>)**
- Description: returns the list of the projections of the points from the <*list*> on the straight line <*(AB)*> following the direction <*vector*>.

8.9 rot

- **rot(<list>, <A>, <alpha>)**
- Description: returns the images list of the points from the <*list*> by the rotation with center <*A*> and angle <*alpha*>.

8.10 shift

- **shift(<list>, <vector>)**
- Description: returns the list of the translated points of the <*list*> by the <*vector*>.

8.11 simil

- **simil(<list> , <A>, <lambda>, <alpha>)**
- Description: returns the images list of the points from the <*list*>, by the similitude with center <*A*>, ratio <*lambda*> and angle <*alpha*>.

8.12 sym

- **sym(<list>, <A>, <B>)** or **sym(<list>, <[A,B]>)**
- Description: returns the symetric of the points from the <*list*>, with respect to the straight line (*AB*).

8.13 symG

- **symG(<list>, <A>, <vector>)**
- Description: glide reflexion: returns the list of the points's images from the <*list*>,by the glide reflexion combining the reflexion on the line passing through <*A*> directed by the <*vector*>, with the translation of <*vector*>.

8.14 symO

- **symO(<list>, <[A, B]>, <vector>)**
- Description: returns the symetrics's list of the points from the <*list*> with respect to the line <*(AB)*> and the direction of the <*vector*>.

9) 2D transformation matrices

An affin transformation of the complex plane called f can be represented by its analytic expression in the canonical base $(1, i)$, the general form of that expression is:

$$\begin{cases} x' &= t_1 + ax + by \\ y' &= t_2 + cx + dy \end{cases}$$

That analytic expression will be represented by the list [t1+i*t2, a+i*c, b+i*d], ie: [f(0), f(1)-f(0), f(i)-f(0)], that list will be briefly called (improperly) *matrix* of the transformation f. The two last elements of that list: [a+i*c, b+i*d] represent the matrix of the linear part of f :Lf= $f - f(0)$.

9.1 ChangeWinTo

- **ChangeWinTo(<[xinf+i*yinf, xsup+i*ysup]> [, ortho])**
- Description: modifies the current matrix so that the current window is transformed into the window's great diagonal: <*[xinf+i*yinf, xsup+i*ysup]*>, the window will be orthonormal or not with respect to the optional <*ortho*> parameter's value.(0 by default).

```
\begin{texgraph}[name=ChangeWinTo, file]
Graph image = [
view(-10,10,-5,5),size(7.5),NbPoints:=100,
LabelSize:=footnotesize, SaveWin(),view(-10,-1,-5,5),
ChangeWinTo([-2-2*i,2+2*i]),
Arrows:=1, axes(0,1+i),Arrows:=0,
tMin:=-2, tMax:=2, Color:=red, Width:=8,
Cartesienne(x*Ent(1/x)-1,5,1),
Color:=blue,A:=(1+i)/4, Dparallelo(A,bar(A),-A),
dep:=RealCoord(i*Im(A)), RestoreWin(), SaveWin(),
//zoom
view(1,10,-5,5), background(full,white),
ChangeWinTo([-A,A]), Color:=black,
arr:=RealCoord(-Re(A)+i*Im(A)*0.75),
Arrows:=1, axes(0,A), Arrows:=0,
tMin:=-0.25, tMax:=0.25, Color:=red, Width:=8,
Cartesienne(x*Ent(1/x)-1,5,1),
Color:=blue, Dparallelo(A,bar(A),-A),
RestoreWin(),
//trait
Color:=blue, Arrows:=1,
A:=ScrCoord(dep), B:=ScrCoord(arr),
Bezier(A,A+3*exp(i*pi/2),B-3,B)
];
\end{texgraph}
```

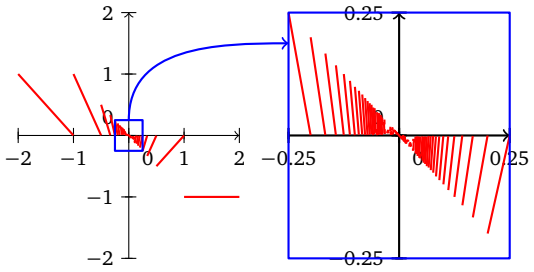


Figure 2: *ChangeWinTo Usage*

9.2 invmatrix

- **invmatrix(<[f(0), Lf(1), Lf(i)]>)**
- Description: returns the inverse of the matrix *<[f(0), Lf(1), Lf(i)]>*, ie the matrix: $[f^{-1}(0), Lf^{-1}(1), Lf^{-1}(i)]$ if it exists.

9.3 matrix

- **matrix(<affin function>, [variable])**
- Description: returns the matrix of the *<affin function>*, by default the *<variable>* is z. The matrix is the like: [f(0), Lf(1), Lf(i)], f is the affine map, and Lf its linear part, more precisely : Lf(1)=f(1)-f(0) and Lf(i)=f(i)-f(0).
- Example(s): matrix(i*bar(z)) returns [0,i,1].

9.4 mulmatrix

- **mulmatrix(<[f(0), Lf(1), Lf(i)]>, <[g(0), Lg(1), Lg(i)]>)**
- Description: returns the matrix of the composite: $f \circ g$, with f and g two affine maps defined by matrices passed as arguments.

10) Plane geometric constructions

Those macros define graphic objects but not draw them : they return a point list representing the objects.

10.1 bissec

- **bissec(<B>, <A>, <C>, <1 or 2>)**
- Description: returns a two points list of the bisector, 1=internal.

10.2 cap

- **cap(<set1>, <set2>)**
- Description: returns the outline of the intersection of the <*set1*> with <*set2*> in the form of two point lists. These two sets are two closed polylines of same orientation with a quite simple shape.The macro *set* (p. 104) is useful to define an draw sets.
- Example(s): intersection of two sets:

```
\begin{texgraph}[name=cap, file]
Graph image = [
Marges(0,0,0,0),size(7.5),
A:=set("A", 0, [rotation:=30]),
B:=set("B", 0, [rotation:=-30]),
C:= cap(A,B),Color:=red,FillStyle:=full,
FillOpacity:=0.5, FillColor:=pink,
Ligne(C,0)
];
\end{texgraph}
```

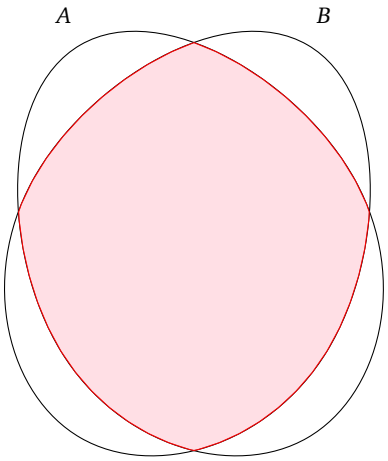


Figure 3: *cap macro*

10.3 capB

- **capB(<set1>, <set2>)**
- Description: returns the outline of the intersection of <*set1*> with <*set2*> in the form of a control points list that has to be drawn using the macro *drawSet* (p. 100). The two given sets have also to be two control point lists representing closed curves of the same orientation, with a quite simple shape. The macro *setB* (p. 104) is useful to define and draw sets.
- Example(s): intersection of two sets:

```
\begin{texgraph}[name=capB, file]
Graph image = [
Marges(0,0,0,0),size(7.5),
A:=setB("A", 0, [rotation:=30]),
B:=setB("B", 0, [rotation:=-30]),
C:= capB(A,B),Color:=red,FillStyle:=full,
drawSet(C, [FillOpacity:=0.5, FillColor:=pink])
];
\end{texgraph}
```

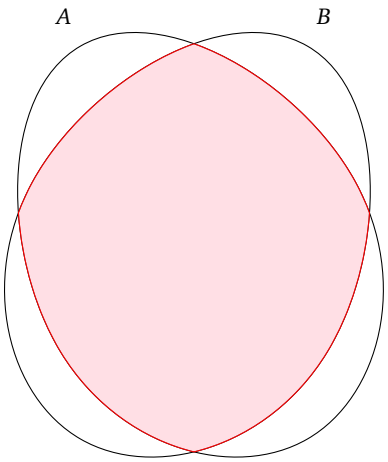


Figure 4: *capB macro*

10.4 carre

- **carre(<A>, <B>, <1 ou -1>)**
- Description: returns the the four point list of the four vertices of the square with consecutive vertices A and B, , 1=counterclockwise.

10.5 cup

- **cup(<set1>, <set2>)**
- Description: returns the outline of the union of <*set1*> with <*set2*> as a point list. Those two sets must be closed curves with the same orientation and a quite simple shape. The macro *set* (p. 104) is useful to define and draw sets.
- Example(s): union of two sets:

```
\begin{texgraph}[name=cup, file]
Graph image = [
Marges(0,0,0,0),size(7.5),
A:=set("A", 0, [rotation:=30]),
B:=set("B", 0, [rotation:=-30]),
C:= cup(A,B),Color:=red,FillStyle:=full,
FillOpacity:=0.5, FillColor:=pink,
Ligne(C,0)
];
\end{texgraph}
```

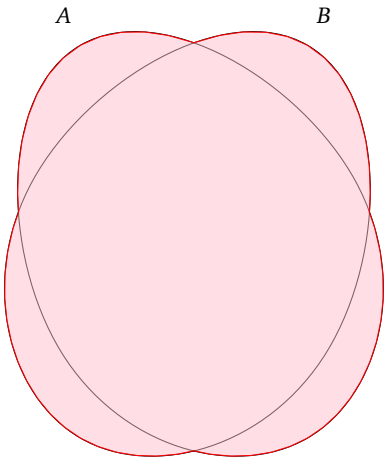


Figure 5: *cup macro*

10.6 cupB

- **cupB(<set1>, <set2>)**
- Description: returns the outline of the union of <*set1*> with <*set2*> as control point list that has to be drawn with the macro *drawSet* (p. 100). Those two sets have also to be two control point lists representing closed curves in the same orientation with a quite simple shape. The macro *setB* (p. 104) is useful to create and draw sets.
- Example(s): union of two sets:

```
\begin{texgraph}[name=cupB, file]
Graph image = [
Marges(0,0,0,0),size(7.5),
A:=setB("A", 0, [rotation:=30]),
B:=setB("B", 0, [rotation:=-30]),
C:= cupB(A,B),Color:=red,FillStyle:=full,
drawSet(C, [FillOpacity:=0.5, FillColor:=pink])
];
\end{texgraph}
```

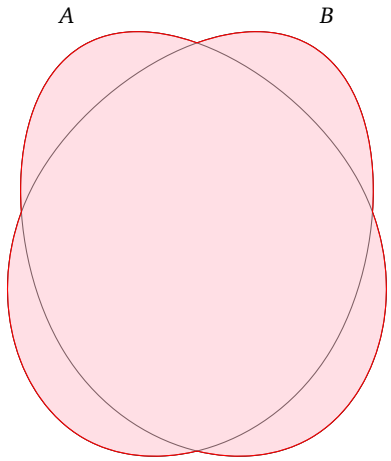


Figure 6: *cupB macro*

10.7 cutBezier

- **cutBezier(<Bézier curve>, <point>, <before(0/1)>)**
- Description: returns a bézier arc corresponding to the <*Bézier curve*> cut before or after the <*point*>, according to the parameter <*before*>. The <*Bézier curve*> is a point list [A1,C1,C2,A2,C3,C4,A3,...] representing successive Bézier curves with two control points : $[A_i, C_k, C_{k+1}, A_{i+1}]$. The result has to be drawn with the command *Bezier* (p. 86).

10.8 Cvx2d

- **Cvx2d(<list>)**
- Description: returns the convex hull of the <*list*> following the RONALD GRAHAM algorithm. The <*list*> does not contain the *jump* constant.
- Example(s): 10 points are randomly chosen in the pavement $[-4,4] \times [-4,4]$ and placed in the variable *P* and each is drawn with its number, then the convex hull is drawn.

```
\begin{texgraph}[name=Cvx2d, file]
Graph image = [
Marges(0,0,0,0),size(7.5),
P:= for k from 1 to 10 do
   z:=8*Rand()-4+i*(8*Rand()-4),
   LabelDot(z, k,"N",1), z
   od,
Ligne(Cvx2d(P),1)
];
\end{texgraph}
```

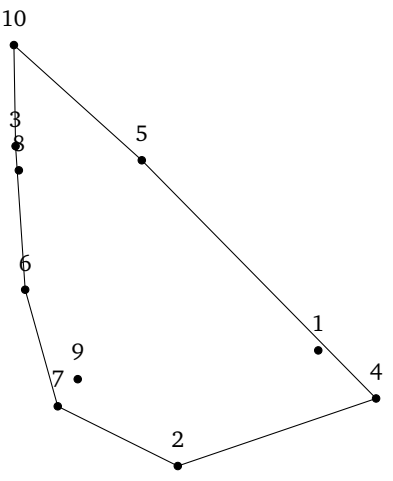


Figure 7: *Cvx2d macro*

10.9 Intersec

- **Intersec(<object1>, <object2>)**
- Description: returns the intersection point list of the two given graphical objects. Those objects can either be grahical commands (Cercle(), Droite(), ...) or a name of an already existing graphical object.
- Example(s): the command Intersec(Cercle(0, 1), Droite(-1,i/2)) returns:
  [0.59851109463416+0.79925554731708*i, -0.99794539275033+0.00102730362483*i].

10.10 med

- **med(<A>, <B>)**
- Description: returns a two point list from the line segment bisector of $[A,B]$.

10.11 parallel

- **parallel(<[A,B]>, <C>)**
- Description: returns a two point list from the parallel to (AB) passing through C.

10.12 parallelo

- **parallelo(<A>, <B>, <C>)**
- Description: returns the parallelogram vertices list whose consecutive vertices are A, B, C.

10.13 perp

- **perp(<[A, B]>, <C>)**
- Description: returns a two point list of the perpendicular to (AB) passing through C.

10.14 polyreg

- **polyreg(<A>, <B>, <edges number>)**
- Description: returns the regular polygon vertices list of center A, passing through B with the given number of edges.

or

- **polyreg(<A>, <B>, <edges number + i*orientation>)** with orientation = +/-1
- Description: returns the vertices list of the regular polygon of consecutive vertices A and B, with the given edges number and orientation (1 for counterclockwise).

10.15 pqGoneReg

- **pqGoneReg(, <vertice>, <[p,q]>)**
- Description: returns the vertices list of the regular $<p/q>$-gon defined by the $<center>$ and a $<vertice>$.
- Example(s): See *here* (p. 100).

10.16 rect

- **rect(<A>, <B>, <C>)**
- Description: returns the rectangle vertices list with consecutive vertices A, B, the opposite edge passing through C.

10.17 setminus

- **setminus(<set1>, <set2>)**
- Description: returns the outline of the difference <*set1*> - <*set2*> as a point list. Those two sets must be closed lines in the same orientation with a quite simple shape. The macro *set* (p. 104) is useful to define and draw sets.
- Example(s): différence of two sets:

```
\begin{texgraph}[name=setminus, file]
Graph image = [
Marges(0,0,0,0),size(7.5),
A:=set("A", 0, [rotation:=30]),
B:=set("B", 0, [rotation:=-30]),
C:= setminus(A,B),Color:=red,FillStyle:=full,
FillOpacity:=0.5, FillColor:=pink,
Ligne(C,0)
];
\end{texgraph}
```

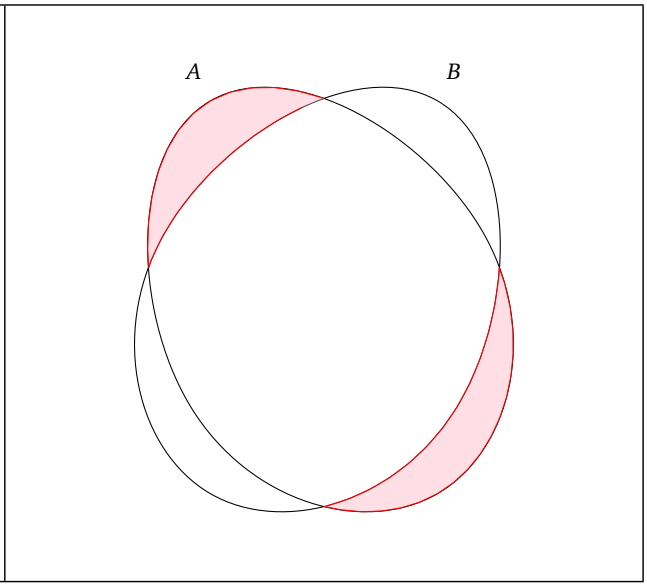


Figure 8: *setminus macro*

10.18 setminusB

- **setminusB(<set1>, <set2>)**
- Description: returns the outline of the difference <*set1*> - <*set2*> as a control point list that must be drawn with the macro *drawSet* (p. 100). Those two sets must also be two control point lists representing closed curves in the same orientation wit a quite simple shape. The macro *setB* (p. 104) is useful to define and draw sets.
- Example(s): difference of two sets:

```
\begin{texgraph}[name=setminusB, file]
Graph image = [
Marges(0,0,0,0),size(7.5),
A:=setB("A", 0, [rotation:=30]),
B:=setB("B", 0, [rotation:=-30]),
C:= setminusB(A,B),Color:=red,FillStyle:=full,
drawSet(C, [FillOpacity:=0.5, FillColor:=pink])
];
\end{texgraph}
```

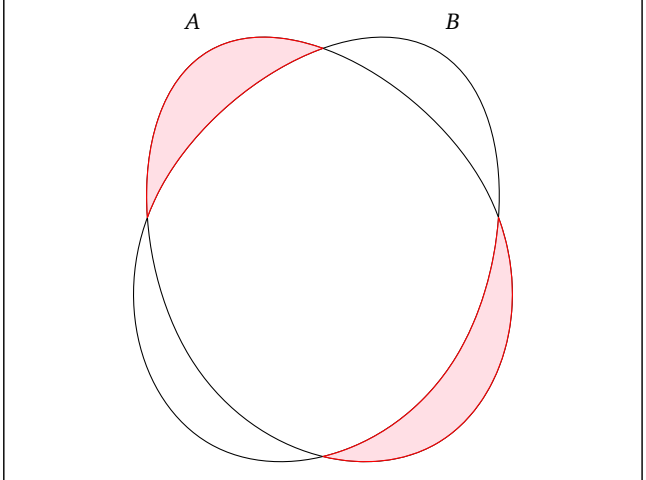


Figure 9: *setminusB macro*

11) Managing flattened postscript

It is possible to transform a pdf or postscript file into a *flattened postscript* using the tool *pstoedit* (http://www.pstoedit.net/). In the output file, everything is a path, including text. TeXgraph can get all the paths from a file written in *flattened postscript*. That is what are proposing macros of that section.

11.1 conv2FlatPs

- **conv2FlatPs(<input file>, <output file> [, working directory])**
- Description: that macro is using the tool *pstoedit* to transform the <*input file*> into *flattened postscript* in the <*output file*>. The <*input file*> must be a ps or pdf file.

11.2 drawFlatPs

- **drawFlatPs(<affix>, <paths read by loadFlatPs> [, options])**
- Description: that macro draw on screen all the paths read in a *flattened postscript* file by the macro *loadFlatPs* (p. 83). The display is done at the requested <*affix*>. The parameter <*options*> is a list (optional) like: *[option1:= value1, ..., optionN:=valueN]*, options are:
  - scale := ⟨ positive number ⟩: scale, 1 by default.
  - position := ⟨ center/left/right/... ⟩: affix position with respect to the image, center by défaut (works like the LabelStyle variable).
  - color := ⟨ color ⟩: to set a color, *Nil* by default means we take the origin color.
  - rotation := ⟨ angle in degrees ⟩: 0 by default.
  - hollow := ⟨ 0/1 ⟩: with the value 0 (default) full filling are done, else the current values of *FillStyle* and *FillColor* are taken in count.
  - select := ⟨ path number list to show ⟩: *Nil* by default, ie: all paths.
  - drawbox := ⟨ 0/1 ⟩: draw the bounding box or not (0 by default).
  - flip := ⟨ 0/1 ⟩: applies or not an horizontal symmetry (0 by default).
  - mirror := ⟨ 0/1 ⟩: applies or not a vertical symmetry (0 by default).

11.3 drawTeXlabel

- **drawTeXlabel(<affix>, <variable containing the TeX formula read by loadFlatPs>, [, options])**
- Description: that macro uses the macro *drawFlatPs* (p. 82) to draw a firstly TEXcompiled expression. The parameter <*options*> is an (optional) list that is like: *[option1:=value1, ..., optionN:=valueN]*, the options are:
  - scale := ⟨ number>0 ⟩: scale, 1 by default.
  - hollow := ⟨ 0/1 ⟩: with the value 0 (default) the fillings are done.

  That macro is internally used by the macro *NewTeXlabel* (p. 83).

11.4 extractFlatPs

- **extractFlatPs(<variable containing a flattened postscript>, <paths number list>, [options])**
- Description: select paths in a variable containing a "flattened postcript" file read by *loadFlatPs* (p. 83), the result is a list: the first complex of the list is width +i*height in cm, then the first complex of each path is Color+i*width. The result can be drawn by *drawFlatPs* (p. 82). The parameter <*options*> is a list (optional) like: *[option1:= value1, ..., optionN:=valueN]*, options are:
  - width := ⟨ number>0 ⟩: width in cm, *Nil* by default for natural width.
  - height := ⟨ number>0 ⟩: height in cm, *Nil* by default for natural height.

11.5 loadFlatPs

- **loadFlatPs(<"flattened postscript file name">, [, options])**
- Description: that macro loads a <*flattened postscript file*>, convert the point coordinates and returns the path list (that can then be drawn with the macro *drawFlatPs* (p. 82)). The parameter <*options*> is a list (optional) like: *[option1:= value1, ..., optionN:=valueN]*, options are:
  - width := ⟨ number>0 ⟩: width in cm, *Nil* by default for the natural width.
  - height := ⟨ number>0 ⟩: height in cm, *Nil* by default for the natural height.
- suppose you have the file *circuit.pdf* in the TeXgraph temporary directory, the following command in a User graphical element:

```
[conv2FlatPs( "circuit.pdf", "circuit.fps", TmpPath),
 stock:= loadFlatPs( [TmpPath,"circuit.fps"] ),
 drawFlatPs( 0, stock, [scale:=1, hollow:=1] )
]
```

will permit to load and draw the content of that file in TeXgraph without the fillings.

11.6 NewTeXlabel

- **NewTeXlabel(<"name">, <affix>, <"text">, [, options])**
- Description: that macro asks TEX to compile the <*"text"*> in a pdf file, that file will then be converted in an eps file using pstoedit, and the result will be loaded by loadFlatPs and stored in a global variable called TeX_+name. A graphical element called <*name*> is created to draw the formula with drawTeXLabel. The parameter <*options*> is a list (optional) like: *[option1:= value1, ..., optionN:=valueN]*, options are:
  - dollar := ⟨ 0/1 ⟩: tells TeXgraph to add or not the tags \[and \] around the formula. (0 by default).
  - scale := ⟨ number>0 ⟩: scale, 1 by default.
  - hollow := ⟨ 0/1 ⟩: with the value 0 (by default) the fillings are done.

  In the options, following attributes can also be used: *LabelSize*, *LabelStyle*, *LabelAngle* and *Color*.

  Here is the definition of that macro:

```
[dollar:=0, scale:=1, hollow:=0, $options:=%4,
     $L:=TeX2FlatPs( %3, dollar), $aux:=NewVar(["TeX_",%1],L),
 NewGraph(%1, ["drawTeXlabel(",%2,", TeX_",%1,", [scale:=",scale,", hollow:=",hollow,"])"]),
 ReDraw()
]
```

  The formula is written in the file *formula.tex*, then we compile the following *tex2FlatPs.tex* file:

```
\documentclass[12pt]{article}
\usepackage{amsmath,amssymb}
\usepackage{fourier}
\pagestyle{empty}
\begin{document}
\input{formula.tex}%
\end{document}
```

  and we convert the result into *flattened postscript* before loading.

  That macro is to be used in the command line or within macros that create graphical elements, but not directly in a User graphical element. example:

```
NewTeXlabel( "label1", 0, "\frac{\pi}{\sqrt{2}}", [scale:=1.5, Color:=blue, LabelAngle:=45])
```

12) Other

12.1 pdfprog

- **pdfprog()**.
- Description: that macro is used internally to store the program needed to convert eps to pdf format. By default the macro contains the string: *"epstopdf"*. By editing the file TeXgraph.mac, you can change the tool to be used.

Chapter VIII

Graphical Functions and macros

Those macros and functions create a graphical element as soon as they are evaluated and return the *Nil* result, can only be used **within a "User" graphical element**[1].

Those can be used within macros, but these ones will be evaluated only if those macro are executed while creating a "User" graphical element.

1) Predefined graphical functions.

Note:
<argument>: shows that the argument is **mandatory**.
[, argument]: shows that the argument is **optional**.

1.1 Axes

- **Axes(<origin>, <scaleX + i*sccaleY> [, origin label position])**.
- Description: draw axes, <*origin*> is the point affix of the axes intersection, <*scaleX*> is the scale on the Ox axis, and <*scaleY*> on the Oy axis. A scale set to zero means no graduations. The ticks length is in the global variable xyticks that is editable. The distance between labels and the ticks is stored in the xylabelsep variable that is also editable.

  The third parameter is optional, it gives the position of the labels for the origins of the two axes. This is a complex number $a + ib$, the real part is about the Ox axis origin, and the imaginary part is about the Oy axis origin. These two numbers can be assigned with three values:

  - 0: the label is hidden,
  - 1: the label is displayed, like the other graduations,
  - 2: tThe label is shifted so that it is not on the other axis (default value).

- In the *Attributes*, those can be modified: Line style, thickness, color an labels size.

[1] Menu Option *Elément graphique/créer/Utilisateur* ie: Graphical element/create/User.

```
\begin{texgraph}[name=Axes, file]
Graph image = [
view(-5,4,-5,5),Marges(0.5,0,0,0.5),
size(7.5), Width:=2,
Color:=lightgray, Grille(-5-5*i,(1+i)/2),
Width:=4,
Color:=gray, Grille(-5-5*i,(1+i)),
Color:=black, Arrows:=1,
Axes(-5-5*i,1+i,1+i),Arrows:=0,
LabelAxe(x,-pi-5*i,"$-\pi$",2-i,1),
LabelAxe(x,pi-5*i,"$\pi$",2+i,1),
SaveAttr(),
FillStyle:=full, FillColor:=lightblue,
FillOpacity:=0.5,
domaine2(3*sin(x) ^ 3, -5,-pi,pi),
RestoreAttr(),
Color:=red, Arrows:=0, Width:=8,
Cartesienne( 3*sin(x) ^ 3 )
];
\end{texgraph}
```

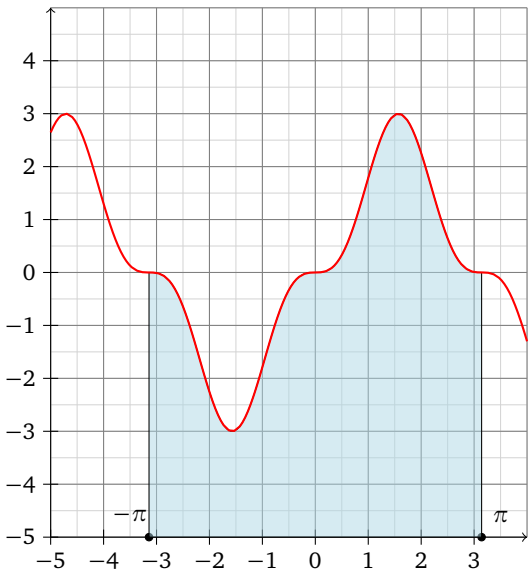


Figure 1: *Axes Command*

1.2 (Poly-)Bézier

- **Bezier(<point list>).**
- Description: draw Bézier curves successively (with eventually some segments). There are several possibilities for the point list:
  1. a three point list $[A, C, B]$, then it is a Bézier curve of origin <A> and end <B> with a control point <C>, this is the curve parametrized with:
  $$(1-t)^2A + 2t(1-t)C + t^2B$$
  2. a 4 or more point list: [A1, C1, C2, A2, C3, C4, A3...]: then it is several Bézier curve with two control points, the first goes from A1 to A2, is controled by C1, C2 (parametrized by $(1-t)^3tA1+3(1-t)^2tC1]+3(1-t)t^2C2+t^3A2$), the second goes from A2 to A3 is controlled by C3,C4 ...etc. An exception though, two control points can be replaced with the *jump* contant. In that case we jump directly from A1 to A2 by drawing a segment.
- The calculated point number (for each curve) is editable in the *Attributes* (NbPoints variable).

```
\begin{texgraph}[name=Bezier, file]
Graph image = [
view(-4,4,-4,5),Marges(0,0,0,0),
size(7.5), Width:=8,
A:=-3+4*i, B:=3+i, C:=3-3*i, D:=-3-3*i,
C1:=4.5*i,C2:=-2*i, C3:=2-i, C4:=-2,
FillStyle:=full, FillColor:=lightblue,Color:=red,
Bezier(A,C1,C2,B,jump,C,C3,C4,D,jump,A),
FillStyle:=none, DotStyle:=cross,
DotScale:=2,Color:=black,
LabelDot(A,"$A$","N",1),
LabelDot(B,"$B$","E",1),
LabelDot(C,"$C$","SE",1),
LabelDot(D,"$D$","SO",1),
LabelDot(C1,"$C_1$","E",1),
LabelDot(C2,"$C_2$","SO",1),
LabelDot(C3,"$C_3$","N",1),
LabelDot(C4,"$C_4$","N",1),
LineStyle:=userdash,
DashPattern:=[5,2,0.5,2], Width:=6,
LineCap:=round,
Ligne([A,C1,C2,B,jump,C,C3,C4,D],0)
];
\end{texgraph}
```

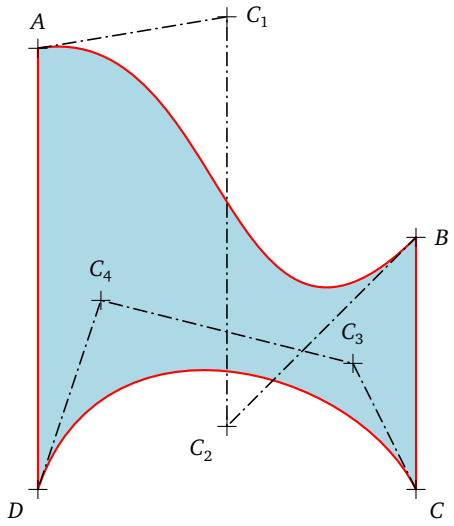


Figure 2: *Bezier Command*

1.3 Cartesienne (cartesian)

- **Cartesienne(<f(x)> [, n, 1]).**
- Description: draw the cartesian curve whose equation is $y = f(x)$. The <*n*> parameter (optional) is an integer (5 by default) is here to set the step: When the distance between two consecutive points is above a limit, then an intermediate point is calculated [using dichotomy], it can be iterated n times. After n iterations, if the distance is still above the limit and the optional value 1 is present, then a discontinuity (*jump*) is inserted in the point list.

```
\begin{texgraph}[name=Cartesienne, file]
Graph image = [
view(-2,2,-0.1,2),Marges(0.5,0.5,0.5,0.5),
size(7.5), tMin:=-2, tMax:=2,
Color:=darkgray, Width:=8,
LineStyle:=dotted, Grille(0,0.5*(1+i)),
Color:=black, LineStyle:=solid,Axes(0,1+i,1),
NbPoints:=100, Width:=8, Color:=darkseagreen,
Cartesienne(x*Ent(1/x),5,1)
];
\end{texgraph}
```

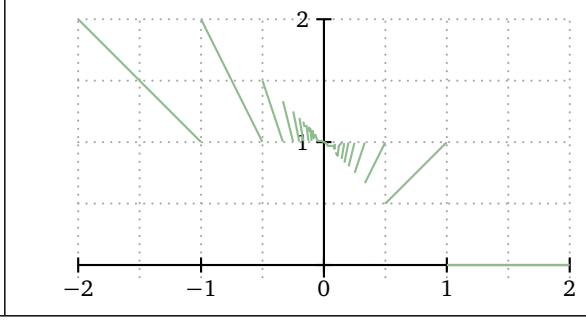


Figure 3: *Curve and discontinuity*

1.4 Courbe (curve)

- **Courbe(<f(t)> [, n, 1]).**
- Description: draw the curve parametrized by <*f(t)*> where f comes with complex values.

  The optional parameter <*n*> is an integer (5 by default) that permits to change the step the following way: If the distance between two consecutive points is above a limit, then an intermediate point is calculated (using dichotomy), it can be repeated n times. If after n iterations the distance between two consecutive points is still above the limit, and if the optional value 1 is present, then a dicontinuity (*jump*) is inserted in the point list.

1.5 Droite (Straight Line)

- **Droite(<A>, <B> [, C]).**
- Description: draw the line (AB) only if the <*C*> argument is missing. Else this is the line with cartesian equation : <*A*>x+<*B*>y=<*C*>.

```
\begin{texgraph}[name=Droite, file]
Graph image = [
view(-5,5,-5,5),Marges(0,0,0,0),
size(7.5),
F:=sqrt(7), F':=-F, {foyers}
Width:=1, Color:=darkgray,
for t from -pi to pi step 0.1 do
 M:=4*cos(t)+3*i*sin(t),
 Vn:=(M-F)/abs(M-F)+(M-F')/abs(M-F'),
 Droite(M,M+Vn),{normal to the ellipse}
od,
Width:=8, Color:=red,
Ellipse(0,4,3),
LabelDot(F,"$F$","S",1),
LabelDot(F',"$F'$","S",1)
];
\end{texgraph}
```

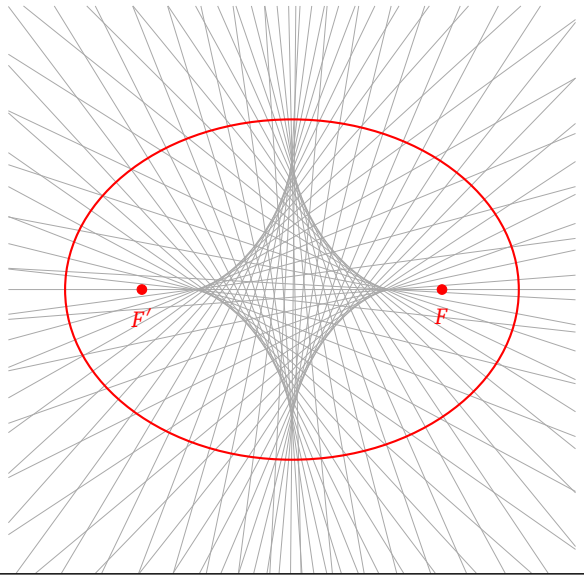


Figure 4: *Evolute of an ellipse*

1.6 Ellipse

- **Ellipse(<A>, <Rx>, <Ry> [, direction]).**
- Description: draw an ellipse with center <*A*> radius <*Rx*> and <*Ry*> on respective axes Ox and Oy. The last parameter <*direction*> is an angle (in degrees), zero by default that is showing the direction of the ellipse with respect to the horizontal.

```
\begin{texgraph}[name=ellipse, file]
Graph image = [
view(-5.25,5.25,-5.25,5.25),
Marges(0,0,0,0), size(7.5),
background(full,blue),
Width:=4, Color:=white,
inclin:=[0,35,-35],
for z in inclin do
 Ellipse(0,5,2,z)
od,
Width:=2*mm, Ellipse(0,1.5,4.5),
Label(-0.1,
  "\resizebox{6cm}{3.5cm}{R\ T\ F}")
];
\end{texgraph}
```

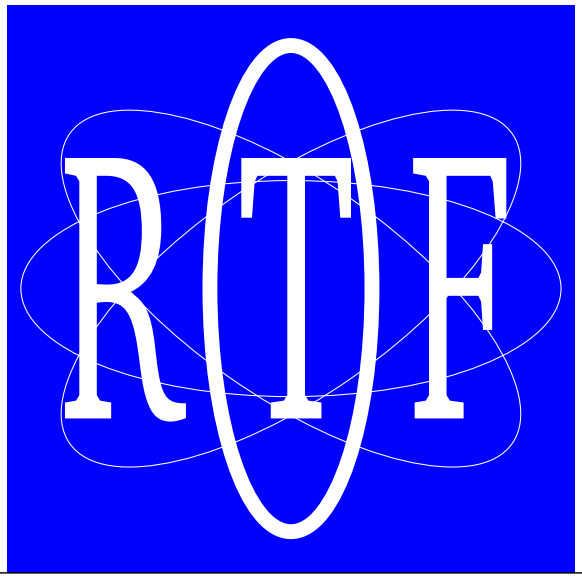


Figure 5: *Ellipses*

1.7 EllipticArc

- **EllipticArc(<B>, <A>, <C>, <Rx>, <Ry> [, direction]).**
- Description: draw an elliptic arc with Ox and Oy axes and center <*A*>, Ox radius is <*Rx*>, Oy radius is <*Ry*>. The arc is drawn from the line (AB) until the line (AC). If the optional argument <*direction*> is 1 (default value) then the arc is drawn counterclockwise, and clockwise if its vaue is -1.

```
\begin{texgraph}[name=EllipticArc, file]
Graph image = [
view(-2.25,3.75,-2,5),Marges(0,0,0,0),size(7.5),
A:=0, B:=3+i, C:=2+4*i,
DotScale:=2, Width:=8,
Ligne([B,A,C],0), Color:=red,
LabelDot(A,"$A$","S",1),
LabelDot(B,"$B$","N",1),
LabelDot(C,"$C$","SE",1),
Arrows:=1, Color:=blue,
EllipticArc(B,A,C,2,1,-1),
EllipticArc(B,A,C,2,3,1)
];
\end{texgraph}
```

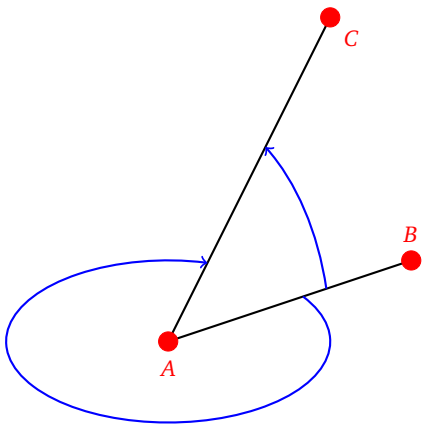


Figure 6: *EllipticArc Command*

Note: for a circle arc, using equals values for <*Rx*> and <*Ry*> is a solution, but it is easier to use the macro *Arc* (p. 95) that is replacing the **Arc** arc command of the old version.

1.8 EquaDif

- **EquaDif(<f(t,x,y)>, <t0>, <x0 + i*y0> [, mode]).**
- Description: draw an approximate solution of the differential equation: $x'(t)+iy'(t) = f(t,x,y)$ with initial condition $x(t_0) = x0$ and $y(t_0) = y_0$. The last parameter (optional) can be 0, 1 or 2:
  - <*mode*>=0: the curve represents the points coordinate $(x(t), y(t))$ (default value).
  - <*mode*>=1: the curve represents the points coordinate $(t, x(t))$.
  - <*mode*>=2: the curve represents the points coordinate $(t, y(t))$.

  This is the RUNGE-KUTTA 4th order method that is used here.
- Example(s): the equation $x'' - x' - tx = \sin(t)$ with initial condition $x(0) = -1$ and $x'(0) = 1/2$ is put in the form:

$$\begin{pmatrix} X' \\ Y' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ t & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \end{pmatrix} + \begin{pmatrix} 0 \\ \sin(t) \end{pmatrix}$$

with $X = x$ and $Y = x'$:

```
\begin{texgraph}[name=EquaDif, file]
Graph image = [
view(-10.5,2.5,-1.5,4.5),Marges(0,0,0,0),
size(7.5,0), Arrows:=1, Width:=4,
Axes(0,1+i), Arrows:=0,
LabelAxe(y,4.25*i,"$x$"),
LabelAxe(x,2,"$t$",2),
Width:=8, Color:=red, tMin:=-10, tMax:=2,
EquaDif(y+i*(t*x+y+sin(t)),0,-1+i/2, 1),
Color:=black, LabelStyle:=stacked,
Label(-6+2*i,
 "$x''-x'-tx=\sin(t)$\\
with $x(0)=-1$ and $x'(0)=\frac12$")
];
\end{texgraph}
```

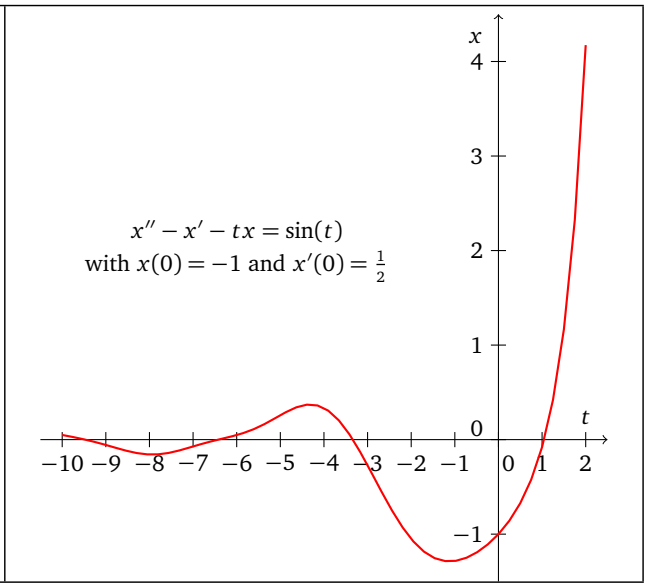


Figure 7: *Differential equation*

1.9 Grille (grid)

- **Grille(<origin>, <scaleX + i*scaleY>).**
- Description: draw a grid, <*origin*> is the origin point affix, <*scaleX*> is the scale on the Ox axis, and <*scaleY*> on the Oy axis, a scale set to zero means no graduations.
- By editing the *Attributes*, linestyle, thickness, color can be modified. There is nothing but the grid that is drawn, axes (and ticks...) can be drawn on top of the grid.

1.10 Implicit

- **Implicit(<f(x,y)> [, n, m]).**
- Description: draw the implicit curve of equation $f(x,y) = 0$. The abscissa interval is divided into <*n*> parts and the ordinate's into <*m*> parts ($n = m = 50$ by default). On each pad, a sign change is tested. If yes, then a dichotomy is applied on the edges of the pad.

```
\begin{texgraph}[name=Implicit, file]
Graph image = [
view(-5,5,-5,5),Marges(0,0,0,0),
size(7.5), Arrows:=1, Width:=4,
Axes(0,1+i), Arrows:=0,
Width:=8, Color:=red,
Implicit( sin(x*y) )
];
\end{texgraph}
```

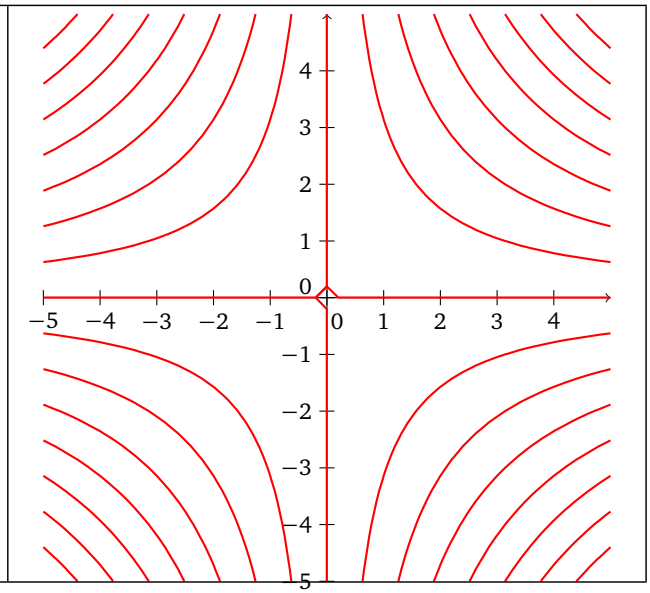


Figure 8: $\sin(xy) = 0$ *equation*

1.11 Label

- **Label(<affix1>, <text1>,..., <affixN>, <textN>)**.
- Description: places the <*text1*> string at position <*affix1*> ... etc. Parameters <*text1*>,..., <*textN*> are then interpreted as *strings* (p. 29).

```
\begin{texgraph}[name=Label, file]
Include "PolyedresII.mac";
Graph image = [
view(-5,5,-5,5),Marges(0,0,0,0),
size(7.5,0),
C:=Cube(Origin, M(3,3,0)),
S:=Sommets(C), Point3D(S),
DrawPoly(C,0), k:=0,
for Z in S by 2 do
 Inc(k,1),
 Label(Proj3D(Z)+
  if k>4 then 0.5*i else -0.5*i fi,
    ["$S_",k,"$"])
od
];
\end{texgraph}
```

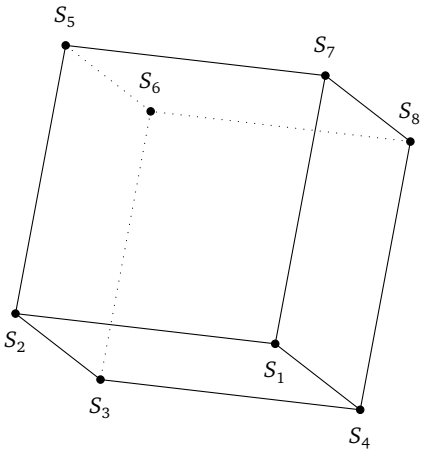


Figure 9: *point label*

1.12 Ligne (polyline)

- **Ligne(<list>, <closed> [, radius])**.
- Description: draw the polyline defined by the list. If the parameter <*closed*> is 1, the polyline will be closed, if the value is 0, the line is open. If the argument <*rayon*> is set (0 by default), then the polyline “angles” are rounded with a circle arc whose radius is <*radius*>.

```
\begin{texgraph}[name=Ligne, file]
Graph image = [
Marges(0,0,0,0), size(7.5),
A:=-5-5*i, B:=5*i, C:=5-5*i,niv:=6,
Tr:=[A,B,C,jump], {initial}
for k from 1 to niv do
 Tr:=[hom(Tr,A,0.5),hom(Tr,B,0.5),
     hom(Tr,C,0.5)]
od,
FillStyle:=full,FillColor:=blue,
Ligne(Tr,1)
];
\end{texgraph}
```

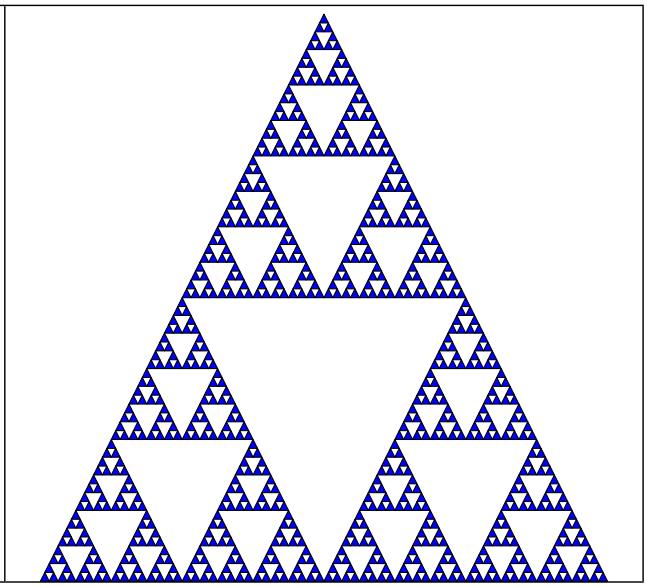


Figure 10: SIERPINSKI*'s triangle*

1.13 Path

- **Path(<list> [, closed (0/1)]**
- Description: draw the path by reading the *<list>* and close the last component of it if the optional argument is 1 (default value: 0). The list gives several points (affixes) and instructions in the following list:
  - **line**: connects the points with a polyline,
  - **linearc**: connects the points with a polyline, but angles are rounded with a circle arc, the value before the linearc command is interpreted as the arcs radius.
  - **arc**: draw a circle arc, four arguments are needed: 3 points (start, center, end) and the radius, plus eventually a fifth argument : the direction (+/- 1). The default value is 1 (counterclockwise).
  - **ellipticArc**: draw an elliptic arc, five arguments are needed: three points, the Xradius, Yradius and eventually an other argument: the direction (+/- 1). The default value is 1 (counterclockwise).Plus eventually a seventh argument: the direction (in degrees) of the great axis with respect to the horizontal.
  - **curve**: connects the points with a natural cubic spline.
  - **bezier**: connects the first and the fourth point with a Bézier curve (the second and third points are control points)
  - **circle**: draw a circle. Two arguments are needed: A point and the center, or three arguments that are three points of the circle.
  - **ellipse**: draw an ellipse. The arguments are: one point, the center, rX radius, rY radius, inclination of the great axis relative to the horizontal (optional).
  - **move**: indicate a movement without drawing.
  - **closepath**: close the curent component.

  By convention, the first point of the number n+1 part is the last point of the number n part.
- Example(s):

```
\begin{texgraph}[name=Path, file]
Graph image = [
view(-5,5,-4,6),Marges(0,0,0,0),size(7.5),
Axes(2*i,1+i),Eofill:=1,
FillStyle:=full,FillOpacity:=0.9,
FillColor:= blue, Width:=8,
Path([-4,i,circle,
    -3+2*i,move,-3,-2,line,
    0,2,2,-1,arc,
    3,3+3*i,0.5,linearc,
    1,-1+5*i,-3+2*i,bezier,
    closepath,
    ])
];
\end{texgraph}
```

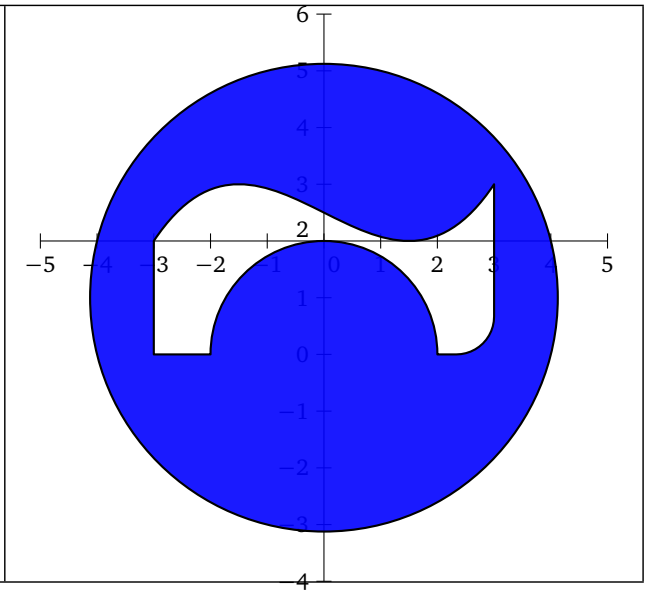


Figure 11: *Path and Eofill commands*

1.14 Point

- **Point(<A1>, ..., <An>).**
- Description: represents the point cloud: <*A1*> ... <*An*>.

```
\begin{texgraph}[name=Point, file]
Graph image = [
view(2.75,4,0,1),
Marges(0.75,0.5,0.5,0.5),size(7.5),
Axes(Xmin+i*Ymin,0.25+0.2*i,1+i),
pas:=0.001, Color:=red,
DotScale:=0.1,
Point(
 for r from Xmin to Xmax step pas do
  u:=0.5,
  for k from 1 to 25 do u:=r*u*(1-u) od,
  for k from 1 to 25 do u:=r*u*(1-u), r+i*u od
 od)
];
\end{texgraph}
```

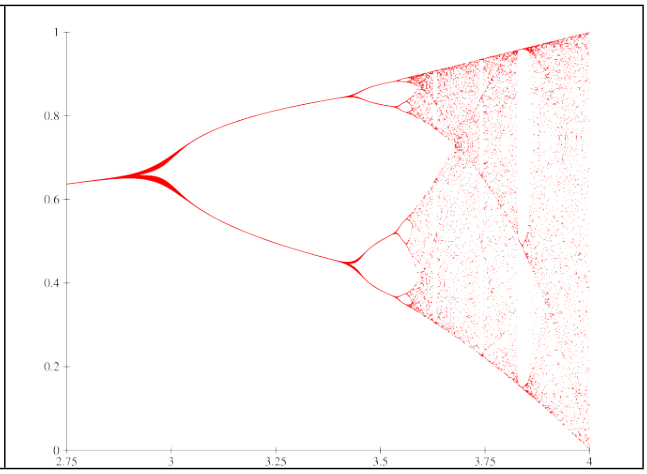


Figure 12: *Bifurcation diagram of the sequence* $u_{n+1} = ru_n(1-u_n)$

1.15 Polaire

- **Polaire(<r(t)> [, n, 0/1]).**
- Description: draw the polar curve of equation $\rho = r(t)$.The optional parameter <*n*> is an integer (5 by default) that permit to vary the step in the following way: when the distance between two consecutive points is greater than a limit, then an intermediate point is calculated (using dichotomy), this can be repeated *n* times. If after *n* iterations the distance between two points is still greater than the limit, and if the optional value 1 is present, then a discontinuity (*jump*) is inserted in the point list.

```
\begin{texgraph}[name=Polaire, file]
Graph image = [
view(-3,2,-2,3),Marges(0.25,0.25,0.25,0.25),
size(7.5),Width:=4,
Axes(0,1+i),NbPoints:=250,tMin:=-25,tMax:=25,
courbe:=Get(Polaire((t+1)/(t-1))),
ptDoubles:= courbe InterL courbe,
Width:=8, Color:= blue, Ligne(courbe,0),
DotStyle:=dotcircle, DotScale:=2,
Point(ptDoubles),
Label(1+2*i,"$r(t)=\dfrac{t+1}{t-1}$")
];
\end{texgraph}
```

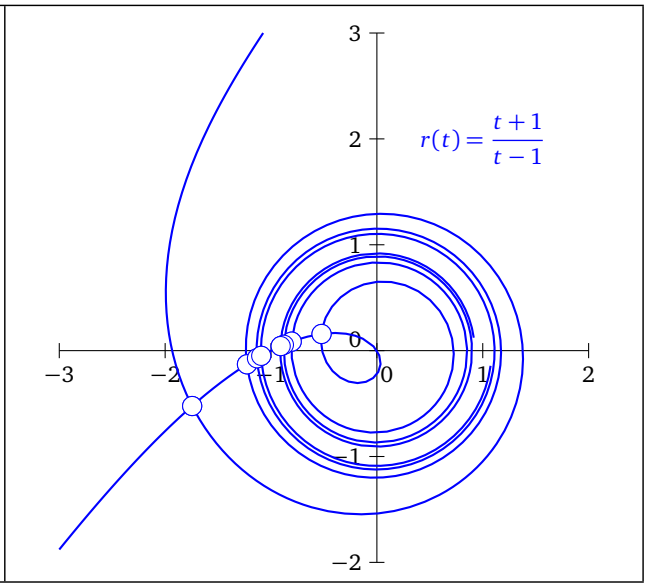


Figure 13: *Polar curve and double point*

1.16 Spline

- **Spline(<V0>, <A0>,..., <An>, <Vn>)**.
- Description: draw the cubic spline passing through the points <*A0*> to <*An*>. <*V0*> and <*Vn*> are the velocity vectors at the ends [restraint], if one of them is zero then the corresponding end is considered as free (no restraint).

```
\begin{texgraph}[name=Spline, file]
Graph image = [
view(-5,5,-5,5),Marges(0.25,0.25,0.25,0.25),
size(7.5),Width:=4,Axes(0,1+i),
A:= -4-3*i, B:=-2+2*i, C:=1-3*i, D:=4+3*i,
LabelDot(A,"$A$","S",1),LabelDot(B,"$B$","N",1),
LabelDot(C,"$C$","S",1),LabelDot(D,"$D$","O",1),
Width:=8,Color:=red, Spline(0,A,B,C,D,0),
Ligne([-4.5+4.5*i,-4+4.5*i],0), LabelStyle:=left,
Label(-3.5+4.5*i,"free"),
Color:=blue,Spline(5,A,B,C,D,5*i),
Ligne([-4.5+3.5*i,-4+3.5*i],0),
Label(-3.5+3.5*i,"restraint"),
Width:=4,
Arrows:=1, Ligne([A,A+2,jump,D,D+2*i],0)
];
\end{texgraph}
```

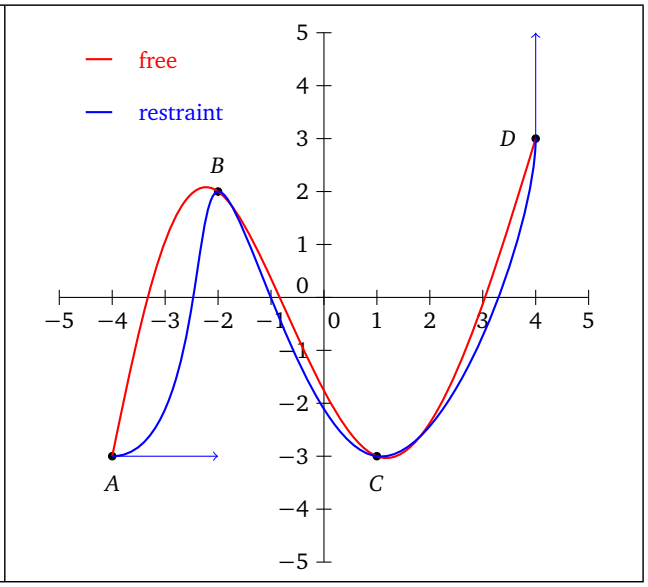


Figure 14: *Spline command*

2) Bitmap drawing commands

The 1.974 TeXgraph version offer a few basic commands to do some bitmap drawing. That drawing can be saved (in the format *bmp*) but it is not included in other exports of the software. Those commands only work with the GUI version of TeXgraph. Each pixel is identified by its affix $x+iy$. x and y are integer. The origin is at the top left corner of the drawing area **margins excluded**, The Ox axix is directed towards right and the Oy towards bottom.

2.1 DelBitmap

- **DelBitmap()**.
- Description: removes the current bitmap.

2.2 GetPixel

- **GetPixel(<affix pixel list>).**
- Description: returns the pixel color list of the pixel <*list*>. The pixel affixes are in the form $a+ib$ with a and b greater than or equal to zero. The origin is the top left corner of the graphic area **margin excluded**.

2.3 MaxPixels

- **MaxPixels().**
- Description: returns the graphic area size in pixels using the form: *maxX+i*maxY*, then for the pixel coordinates (integer), the abscissa interval is *[0 .. maxX]* and the ordinate's *[0 .. maxY]*. Each pixel is identified with integer coordinates, so each pixel affix is like $a+ib$ with a in *[0 .. maxX]* and b in *[0 .. maxY]*. The origin is at the top left corner of the graphic area (**margins excluded**).

2.4 NewBitmap

- **NewBitmap([background]).**
- Description: create a new bitmap (empty). By default the background color is white.

2.5 Pixel

- **Pixel(<pixel affixes list>).**
- Description: permits to light on a pixel <*list*>. That list is in the form: *[affix, color, affix, color, ...]*. Pixel affixes are in the form $a+ib$ with a and b greater than or equal to zero integers. The origin is the top left corner of the graphic area (**margins excluded**).

2.6 Pixel2Scr

- **Pixel2Scr(<affix>).**
- Description: convert a pixel <*affix*> (integer coordinates) into an affix in the user coordinate system (on screen).

2.7 Scr2Pixel

- **Scr2Pixel(<affix>).**
- Description: convert the point <*affix*> in the screen user coordinate system into integer coordinate (pixel affix corresponding to the point).
- Example(s): a Julia set, the command is to be placed in a user graphical element. The *png* image has been gotten using the *snapshot* button of the graphic interface and a *bmp* export then a conversion into *png*:

```
\begin{texgraph}[name=julia, file]
Graph image = [
view(-1.5,1.5,-1,1),Marges(0,0,0,0),size(7.5),
 NewBitmap(),
  T:=100, m:=MaxPixels(), c:=-0.181-0.667*i,
  for x from 0 to Re(m) do
   Pixel(
    for y from 0 to Im(m) do
      N:=0, z:=Pixel2Scr(x+i*y),
      repeat
         z:=z^2+c, Inc(N,1)
      until (N=T) Or (abs(z)>2) od,
      x+i*y, MixColor(darkred,1-N/T,yellow,N/T)
   od )
  od
];
\end{texgraph}
```

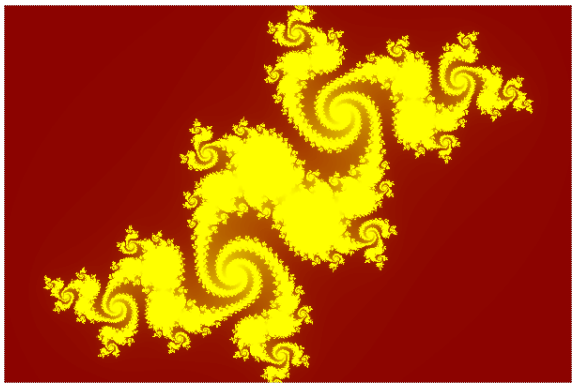


Figure 15: *A Julia set*

3) Graphic macros from TeXgraph.mac

3.1 angleD

- **angleD(<B>, <A>, <C>, <r>).**
- Description: draw the $\widehat{BAC}$ angle with a parallelogram with side r.

3.2 Arc

- **Arc(<B>, <A>, <C>, <R> [, direction]).**
- Description: draw a circle arc with center <*A*> radius <*R*>. The arc is drawn starting from the line (*AB*) to the line (*AC*), the optional argument <*direction*> indicate: counterclockwise if the value is 1 (default value), clockwise if the value is −1.

```
\begin{texgraph}[name=Arc, file]
Graph image = [
view(-2.25,3.75,-2,5),Marges(0,0,0,0),size(7.5),
A:=0, B:=3+i, C:=2+4*i,
DotScale:=2, Width:=8,
Ligne([B,A,C],0), Color:=red,
LabelDot(A,"$A$","S",1),
LabelDot(B,"$B$","N",1),
LabelDot(C,"$C$","SE",1),
Arrows:=1, Color:=blue,
Arc(B,A,C,1,-1),
Arc(B,A,C,2,1)
];
\end{texgraph}
```

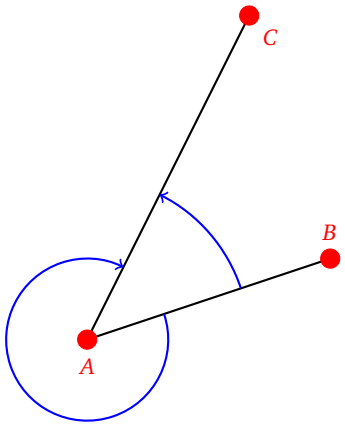


Figure 16: *Arc command*

3.3 arcBezier

- **arcBezier(<B>, <A>, <C>, <r> [,direction]).**
- Description: same effect as the graphic macro *Arc* but the arc is drawn with Bézier curves.

3.4 axes

- **axes(<[origin, extentX, extentY]>, <gradX+i*gradY> [, subdivX+i*subdivY, posOriginX+i*posOriginY, num, "text", den, firstnum]).**
- Description: draw and graduate axes passing through <*origin*> (affix).Usage :like the command *Axes* (p. 85) with the variables *xylabelpos* and *xyticks*. By default axes are drawn on the whole window if the optional parameters <*extentX*> and <*extentY*> are omitted. The parameter <*extentX*> is a complex number representing the abscissa interval : `xmin+i*xmax`, same with the ordinates with the parameter <*extentY*>, axes are then limited to those intervals. Note: if you need to give a value to <*extentY*> only, it suffices to replace the <*extentX*> value with *jump* (not *Nil*!).
- The optional parameter <*subdivX+i*subdivY*> shows the subdivision number per unit on each axis (0 by default)
- The optional parameter <*posOriginX+i*posOriginY*> places the origin label:
  - <*posOriginX*>=0: no label for the origin (idem for <*posOriginY*>=0),
  - <*posOriginX*>=1: ordinary label (idem for <*posOriginY*>=1)„

  - <*posOriginX*>=2: label shifted towards right of the origin and upwards <*posOriginY*>=2 (default values),
  - <*posOriginX*>=-2: label shifted towards left of the origin and downwards for <*posOriginY*>=-2.

- On the two axes each label is multiplicated by the fraction <*num/den*> (1 by default), added to <*firstnum/den*> (default origin) with the <*"text"*> at the numerator. That macro calls the macro *GradDroite* (p. 101), uses the variables: *usecomma* (0/1: so that the decimal separator is a comma or a point), *dollar* (0/1: to add (or not) $ around the graduation labels), *numericFormat* (0/1/2: managing the numerical format decimal(0), scientific(1), ou engineering(2)), and *nbdeci* (sets the decimal places number).

- Example(s): graduating the axes $\pi/2$ by $\pi/2$: axes(0, pi*(1+i)/2, 1+i, 2+2*i, 1, "\pi", 2, 0). Unlike the Axes command, that macro is sensitive to the current matrix. It calls the macro *GradDroite* (p. 101) that is using the variables *dollar*, *numericFormat* and *nbdeci*.

3.5 axeX

- **axeX(<[origin, posOrigin, extent]>, <Xstep> [, Subdiv, labelPos , num, ""text"", den, firstnum]).**

- Description: drawing and graduating an abscissa axis passing through the <*origin*> with a step <*Xstep*>. The parameter <*extent*> is a complex representing the abscissa interval: *xmin+i*xmax*. If it is omitted, then the whole window used.Note: setting the <*extent*> value only (ie without <*posOrigin*>), it suffices to replace <*posOrigin*> with *jump* (and not *Nil*!).

- <*Subdiv*> is the subdivision number per unit, each abscissa is multiplied by the fraction <*num/den*> (1 by default), added to <*firstnum/den*> (the default origin) with the <*"text"*> at the numerator. This macro calls the macro *GradDroite* (p. 101), uses the variables: *usecomma* (0/1: so that the decimal separator is a comma or a point), *dollar* (0/1: add (or not) $ around the graduations labels), *numericFormat* (0/1/2: managing the numeric format: decimal(0), scientific(1), or ingineer(2)), and *nbdeci* (for the decimal places number displayed).

- The optional parameters <*posOrigin*> and <*labelpos*> are used to place the labels:

  - <*posOrigin*>=0: no label at origin.
  - <*posOrigine*>=1: normal label at origin.
  - <*posOrigine*>=2: label shifted to the right of the origin (default value),
  - <*posOrigine*>=-2 label shifted to the left of the origin,
  - <*labelPos*>=0 : no label at all.
  - <*Re(labelpos)*>=top: labels above the axis,
  - Re(<*labelPos*>)=bottom : labels below (default value),
  - Im(<*Im(labelPos*>)=1: labels orthogonal to the axis.

3.6 axeY

- **axeY(<[origin, posOrigin, extent]>, <Ystep> [, Subdiv, labelPos , num, ""text"", den,firstnum]).**

- Description: draw and graduate an ordinate axis passing through the <*origin*> with the step <*Ystep*>. The parameter <*extent*> sets the ordinate interval using a complex number: *ymin+i*ymax*. If it is omitted, then the drawing uses the whole window. Note: giving a value to the <*extent*> only (no value for <*posOrigin*>) is done by giving the *jump* value to <*posOrigin*> (and not *Nil*!).

- <*Subdiv*> is the subdivisions number per unit , each ordinate is multiplied by the fraction <*num/den*> (1 by default), added to <*firstnum/den*> (the default origin) with the <*"text"*> at numerator. That macro calls the macro *GradDroite* (p. 101), is using the variables: *usecomma* (0/1: so that the decimal separator is a comma or a point), *dollar* (0/1: adds (or not) $'s around the graduation labels), *numericFormat* (0/1/2: managing the numeric format: decimal(0), scientific(1), or ingineer(2)), and *nbdeci* (for the decimal places number displayed).

- The optional parameters <*posOrigin*> and <*labelpos*> are used for label positioning:

  - <*posOrigine*>=0: no label at the origin,

- <*posOrigine*>=1: normal label at the origin,
- <*posOrigine*>=2: label shifted upwards at the origin (default value),
- <*posOrigine*>=-2 label shifted downwards at the origin,
- <*labelPos*>=0 : no label at all,
- <*Re(labelpos)*>=left: labels on the left of the axis (default axis),
- Re(<*labelPos*>=right : labels on the right of the axis,
- Im(<*labelPos*>)=1: labels orthogonal to the axis.

```
\begin{texgraph}[name=axeXY, file]
Graph image = [
view(-5,5,-5,5),size(7.5),
LabelSize:=footnotesize,
//top graphic
SaveWin(),
view(-5,5,0.25,5),Width:=6, Arrows:=1,
SetMatrix([2.5*i,2/pi,2*i]),
axeX(0,pi/2,Nil,Nil,1,"\pi",2), axeY(0,1),
Arrows:=0, Color:=red, Width:=8,
tMin:=-2*pi, tMax:=2*pi, Cartesienne(sin(x)),
Label(pi*1.65+i,"$f(x)=\sin(x)$"),RestoreWin(),
//bottom graphic
SaveWin(), Color:=black,
view(-5,5,-5,-0.25), SetMatrix([-5.5*i,2,i]),
Width:=6, Arrows:=1, usecomma:=1,
axeX([i,0],0.5), axeY([i,0],1,Nil,Nil,1,"a"),
Arrows:=0, Color:=blue, Width:=8,
Cartesienne(x^2+1),
Label(1+4.75*i,"$g(x)=x^2+a$"),
RestoreWin()
];
\end{texgraph}
```

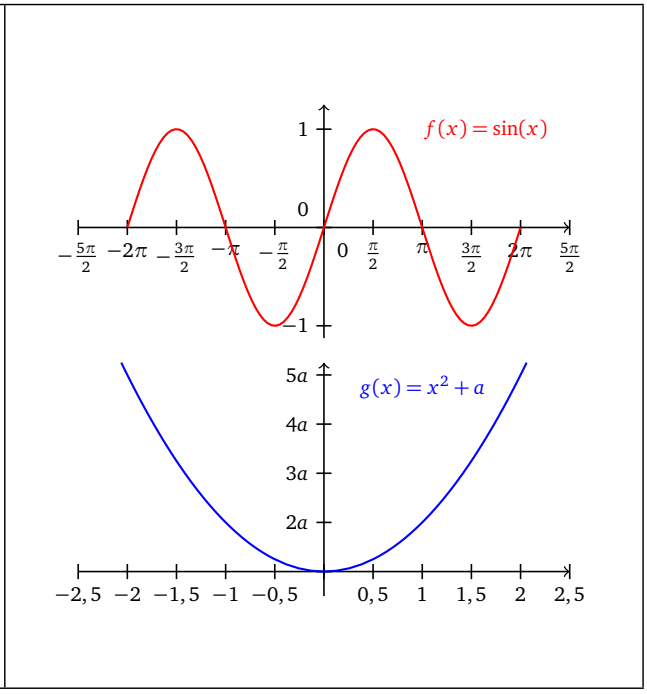


Figure 17: *axeX, axeY usage*

3.7 background

- **background(<fillstyle>, <fillcolor>)**.
- Description: fill the background of the graphical window with the given style and color. That macro updates the *backcolor* variable.

3.8 bbox

- **bbox()**.
- Description: fit the window to the "bounding box" around the current drawing. That macro is designed to be used in the command line at the bottom of the main window, and not in a graphical element.

3.9 centerView

- **centerView(<affix>)**.
- Description: center the graphical window to the given <*affix*>, without changing the current sizes of the graphic. This macro is usually designed to be used in the command line at the bottom of the main window.

3.10 Cercle (circle)

- **Cercle(<A>, <r> [, B]).**
- Description: draw a circle with center <*A*> and radius <*r*> when the third parameter is omitted, else this is the circle defined by the three points : <*A*>, <*r*> and <*B*>.

  Using macros *Arc* and *Cercle* can lead to surprises in the final result if the coordinate system is not orthonormal ! It is orthonormal if *Xscale* and *Yscale* variables are equal, see option *Paramètres/Fenêtre (preferences/window)* (p. 10).

```
\begin{texgraph}[name=cycloide, file]
Graph image = [
view(-5,5,-1,3),Marges(0,0,0,0),
size(7.5), Seg(-5,5),
for t in [-4,-1.85,0,1.85,3] do
  M:=t-sin(t)+i*(1-cos(t)),
  I:=t+i, DotStyle:=cross,
  Point(I), DotStyle:=bigdot,
  Point(M), Cercle(I,1), Seg(M,I),
  Arrows:=1, Arc(M,I,t,0.5,t),
  Arrows:=0, LineStyle:=dashed,
  Seg(I,t), LineStyle:=solid
od,
Width:=8,Color:=red,
Courbe( t-sin(t)+i*(1-cos(t)))
];
\end{texgraph}
```

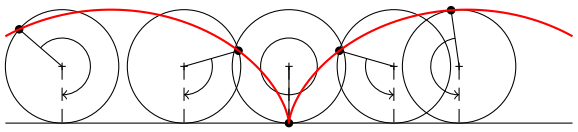


Figure 18: *The cycloid*

3.11 Clip

- **Clip(<list>).**
- Description: clip the already drawn graphical elements (the given <*list*> must be a closed curve). The macro paints the outside of the curve represented by the <*list*>.

3.12 Dbissec

- **Dbissec(<B>, <A>, <C>, <1 or 2>).**
- Description: draw the bisector of the angle $\widehat{BAC}$, internal if the last parameter is 1 and external with the 2 value .

3.13 Dcarre (square)

- **Dcarre(<A>, <B>, <+/-1> [, radius]).**
- Description: draw the square with consecutive vertices <*A*> and <*B*> counterclockwise if the third parameter is 1 (clockwise for −1). If the <*radius*> is present, then the figure “corners” will be rounded by a circle arc with the given radius.

3.14 Ddroite

- **Ddroite(<A>, <B>).**
- Description: draw the half-line $[A, B)$.

3.15 Dmed

- **Dmed(<A>, <B> [, right angle(0/1)]).**
- Description: draw the mediator of the segment $[A, B]$. If the third parameter is 1 (0 by default) then a right angle is drawn.

3.16 domaine1

- **domaine1(<f(x)> [, a, b]).**
- Description: draw the plane part located between the Cf curve, the Ox axis and the lines $x = a$, $x = b$ if a and b were given, else $x =$*tMin* and $x =$*tMax*.

3.17 domaine2

- **domaine2(<f(x)>, <g(x)> [, a, b]).**
- Description: draw the plane part located between the curves Cf, Cg and the lines $x = a$, $x = b$ if a and b were given, else $x =$*tMin* and $x =$*tMax*.

3.18 domaine3

- **domaine3(<f(x)>, <g(x)>).**
- Description: defines the part of the plane located between the curves Cf and Cg with x in the interval *[tMin,tMax]*, by searching the intersection points.

```
\begin{texgraph}[name=domaines, file]
Graph image = [
view(-5,5,-4,5),size(7.5),tMin:=-5,
Axes(-5,1+i,2+2*i), tMax:=-1, Width:=8,
Cartesienne(sin(x)), tMax:=5,
Cartesienne(2*cos(x)),tMin:=-3,tMax:=3,
Cartesienne(x^2), A:=-4.5+(Ymin+1.75)*i, Width:=4,
FillStyle:=full, FillOpacity:=0.5,FillColor:=green,
LabelStyle:=left, LabelSize:=footnotesize,
Dcarre(A+0.25*(-1+i), A+0.25*(1+i), -1),
Label(A+0.4,"domaine2 on $[-4\,;\, -1.5]$ "),
Inc(A,-0.75*i), FillColor:=blue,
Dcarre(A+0.25*(-1+i), A+0.25*(1+i),-1),
Label(A+0.4,"domaine3 between $2\cos(x)$ and $x^2$ "),
Inc(A,-0.75*i), FillColor:=red,
Dcarre(A+0.25*(-1+i), A+0.25*(1+i),-1),
Label(A+0.4,"domaine1 on $[1.1\,;\, 4]$"),
FillColor:=red, domaine1(2*cos(x), 1.1, 4),
FillColor:=green, domaine2(2*cos(x), sin(x),-4,-1.5),
FillColor:=blue, domaine3(2*cos(x),x^2),
Arrows:=2, tangente(2*cos(x),0.5,1.5)
];
\end{texgraph}
```

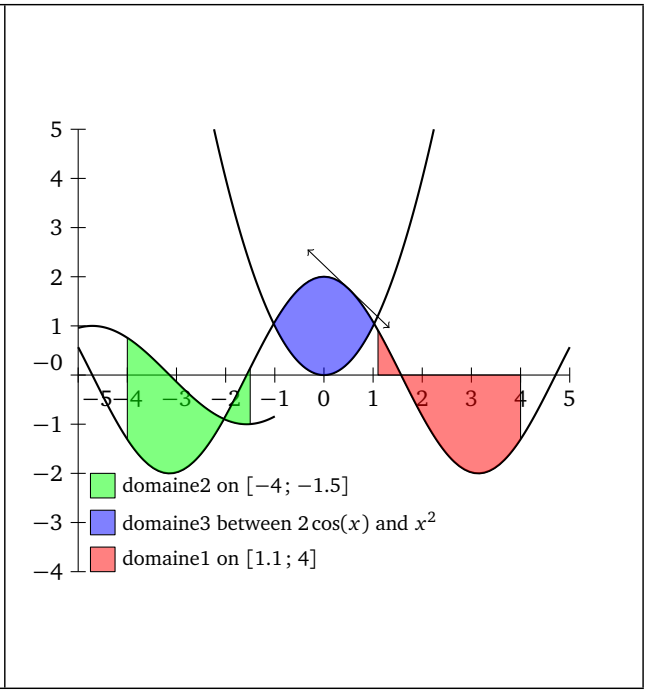


Figure 19: *Example with domaine1, 2 and 3*

3.19 Dparallel

- **Dparallel(<[A, B]> , <C>).**
- Description: draw the parallel to the line <*[A,B]*> passing through <*C*>.

3.20 Dparallelo

- **Dparallelo(<A>, <B> , <C> [, radius]).**
- Description: draw the parallelogram with consecutive vertices <*A*>, <*B*> and <*C*>. If the parameter <*radius*> is present, then the “corners” of the figure are rounded by a circle arc with the given radius.

3.21 Dperp

- **Dperp(<[A, B]> , <C> [, right angle(0/1)]).**
- Description: draw the perpendicular to the line <*(A,B)*> passing through <*C*>. If the third parameter is 1 (0 by default) then a right angle is drawn..

3.22 Dpolyreg

- **Dpolyreg(, <vertice>, <sides number> [, radius]).**
- Description: draw the regular polygon defined by a <*center*>, a <*vertice*> and the <*sides number*>. If the <*radius*> is present, then the figure "corners" are rounded with a circle arc with the specified radius.

or

- **Dpolyreg(<vertice1>, <vertice2>, <sides number+direction*i> [, radius]).**
- Description: draw the regular polygon defined by two consecutive vertices : <*vertice1*> and <*vertice2*>, the <*sides number*>, and the <*direction*> (1 for counterclockwise −1 for clockwise). If the <*radius*> parameter is present, then the figure "corners" are rounded with a circle arc with the spcified radius.

3.23 DpqGoneReg

- **DpqGoneReg(, <vertice>, <[p,q]>).**
- Description: draw the regular <*p/q*>-gon defined by the <*center*> and a <*vertice*>.

```
\begin{texgraph}[name=pqGoneReg, file]
Graph image = [
view(-5,5,-6,6),Marges(0,0,0,0),size(7.5),
FillStyle:=full, FillColor:=lightpink,
Color:=blue, Width:=8,
DpqGoneReg(-2.5+2.5*i,-2.5+0.25*i, [7,3]),
Label(-2.5+5*i,"[7,3]"),
DpqGoneReg(2.5+2.5*i,2.5+0.25*i, [7,2]),
Label(2.5+5*i,"[7,2]"),
DpqGoneReg(-2.5-2.5*i,-2.5-0.25*i, [7,1]),
Label(-2.5-5*i,"[7,1]"),
DpqGoneReg(2.5-2.5*i,2.5-0.25*i, [6,2]),
Label(2.5-5*i,"[6,2]")
];
\end{texgraph}
```

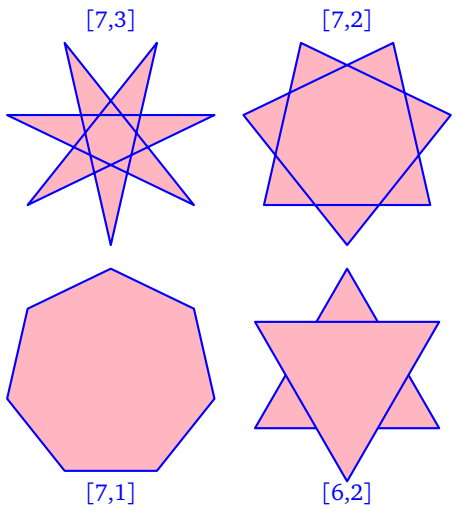


Figure 20: *DpqGoneReg: example*

3.24 drawSet

- **drawSet(<set>).**
- Description: draw a set using the macros *capB* (p. 77), *cupB* (p. 78) or *setminusB* (p. 81).

3.25 Drectangle

- **Drectangle(<A>, <B> , <C> [, radius]).**
- Description: draw the rectangle with consecutive vertices <*A*>, <*B*>, the opposite side passing through <*C*>. If the <*radius*> parameter is present, then the figure "corners" are rounded with a circle arc with the specified radius.

3.26 ellipticArc

- **ellipticArc(<B>, <A>, <C>, <RX>, <RY>, <direction(+/-1)> [, inclination]).**
- Description: draw an elliptic arc with center <*A*>, going from <*B*> to <*C*> with radiuses <*RX*> and <*RY*>, the axis with the radius <*RX*> having an <*inclination*> with respect to the horizontal (in degrees, 0 by default). The <*direction*> parameter indicate the rotating direction : 1 for counterclockwise.

3.27 flecher (arrowing)

- **flecher(<list>, <pos1, ..., posN>).**
- Description: draw arrows along the polyline : <*list*>, each arrow position (pos1, ...) is a number between 0 and 1 (0 for the start of the line and 1 for its end), the arrows are drawn in the polyline drawing direction. To reverse an arrow add +i to the position.
- Example(s): flecher(Get(Cercle(0,3)), [0,0.5])

3.28 GradDroite (graduating a straight line)

- **GradDroite(<[A, origin + i*posOrigin, extent]>, <[u, unit]>, <heightDiv>, <subdiv> [, poslab, direction, num, ""text"", den, firstnum])**
- Description: draduate the straight line passing through <*A*> with the direction vector <*u*> (not necessary a unit vector), <*heightdiv*> is the height of each graduation (cm), , <*subdiv*> is the per unit subdivisions number.

  Optional parameters:

  - <*origin*>: sets the origin graduation <*A*> (0 by default),
  - <*posOrigin*>: shows the label position of the origin <*A*>, following the cases:
    - <*posOrigin*>=0: no label at the origin,
    - <*posOrigin*>=1: normal label at origin (like others)
    - <*posOrigin*>=2: label at origin shifted following the <*u*> vector direction (default value),
    - <*posOrigin*>=-2: label at origin shifted following the oposit direction of the <*u*>.
  - <*extent*>: represents the graduation interval using a complex number: *min+i*max*, the axis drawing will be delimited to that interval. If that parameter is omitted, the drawing will use the whole window.
  - <*unit*>: Indicate the step graduation (1 by default). The value must be positive.
  - <*poslab*> indicate the label position with respect to the axis, that parameter takes the values top or bottom,
  - <*direction*>: labels direction, the value *i* means the labels are orthogonal to the axis, else the direction represents the *LabelStyle* (left, right, top, ...),
  - each graduation is multiplied by the fraction <*num/den*> (1 by default), added to <*firstnum/den*> (the default origin) with the <*"text"*> at the numerator. This macro uses the variables : *usecomma* (0/1: so that the decimal separator is a comma or a point), *dollar* (0/1: adding (or not) $'s around the graduations labels), *numericFormat* (0/1/2: managing the numerical format: decimal(0), scientific(1), or ingineer(2)), *nbdeci* (sets the displayed decimal places) and *maxGrad* (sets the maximal graduations number, 100 by default).

- Example(s): GradDroite([0,1+2*i],[1,0.5], xyticks, 1, bottom, i): means that the origin graduation will be 1 with a label shifted to the right, graduations will go 0.5 by 0.5, labels will be under the axis and orthogonal to the axis.

3.29 LabelArc

- **LabelArc(<B>, <A>, <C>, <R>, <direction>, <"text">, [, options]).**
- Description: that macro draw a circle arc with center <*A*>, radius <*R*> starting from the line (*AB*) until the line (*AC*), the optional argument <*direction*> indicate: counterclockwise if its value is 1 (default value), clockwise if the value is -1. The macro also adds the <*"text"*>. The <*options*> parameter is an optional list in the form *[option1:= value1, ..., optionN:=valueN]*, those options are:

- labelpos := ⟨ inside/outside ⟩: label position (outside by default),
- labelsep := ⟨ distance in cm ⟩: distance between the label and the arc (0.25cm by default).
- rotation := ⟨ nunber ⟩: angle in degrees for the label with respect to the horizontal (0 by default).

It is possible in the option list, to modify the attributes like *Color* for example.

3.30 LabelAxe

- **LabelAxe(<x ou y>, <affix>, <label> [, [labelPos, shift in cm], mark(0/1)]).**
- Description: add a label on one of the axes <*x or y*>, <*affix*> is the point affix where the label will be added.<*label*> is the text to display. Optional parameters <*[labelPos, shift in cm]*> and <*mark*>:
  - Re(<*labelpos*>)=1 means below for Ox and to the right for Oy (default for Ox),
  - Re(<*labelpos*>)=2 means above for Ox and to the left for Oy (default for Oy),
  - Im(<*labelpos*>)=-1 means a shift to the left for Ox, and to the bottom for Oy, if the shift is not given, it is 0.25 cm by default,
  - Im(<*labelpos*>)=1 means a shift to the right for Ox, to the top for Oy, if the shift is not given, it is 0.25cm by default,
  - Im(<*labelpos*>)=0 means no shift (default value),
  - <*mark*>: sets if the point has to be shown or not (using the current dotstyle).

3.31 LabelDot

- **LabelDot(<affix>, <"text">, <direction> [, DrawDot, distance]).**
- Description: that macro displays a text beside the point <*affix*>. The direction can be "N" for north, "NE" for north-east, "NO" for north-west, "SO" for south-west, "S" for south ...etc. , or a list in the form [length, direction] where direction is a complex number, in that second case, the optional parameter <*distance*> is ignored. The point is also displayed when <*DrawDot*> is 1 (0 by default) and the <*distance*> in cm between the point and the text can be redefined (0.25cm by default).

3.32 LabelSeg

- **LabelSeg(<A>, <B>, <"text">, [, options]).**
- Description: that macro draw the segment line defined by <*A*> and <*B*>, and adds the <*"text"*>. The <*options*> optional parameter is a list in the form: *[option1:= value1, ..., optionN:=valueN]*. Those options are:
  - labelpos := ⟨ center/top/bottom ⟩: label positionning (center by default),
  - labelsep := ⟨ distance en cm ⟩: distance from the segment to the label if labelpos is top or bottom (0.25cm by default).
  - rotation := ⟨ number ⟩: angle in degrees of the label with respect to the horizontal (by default the label is parallel to the segment).

  In the options list, it is possible to edit the attributes like *Color* for example.

3.33 markangle

- **markangle(<B>, <A>, <C>, <r>, <n>, <spacing>, <length>).**
- Description: same as *markseg* (p. 102) but to mark a circle arc.

3.34 markseg

- **markseg(<A>, <B>, <n>, <spacing>, <length> [, angle]).**
- Description: marks the segment $[A,B]$ with <*n*> small segments, the <*spacing*> is using the graphical unit, and the length <*length*>, cm. The optional parameter <*angle*> in degrees define the angle of the marks with respect to the line (AB) (45 degrees by default).

3.35 periodic

- **periodic(<f(x)>, <a>, <b> [, divisions, discontinuities]).**
- Description: draw the periodic function curve defined by $y = f(x)$ on the period $[a; b]$, then translates the pattern to cover the whole interval [tMin; tMax]. The two optional parameters are identical to those of parametric curves (divisions number and discontinuities).

```
\begin{texgraph}[name=periodic, file]
Graph image = [
view(-5,5,-5,5), size(7.5),
Axes(0,1+i), Width:=8,
Color:=red,
periodic(if t>0 then 4 else 2 fi,-1,1,5,1),
Color:=blue,
periodic(2*abs(t)-1,-1,1),
Color:=magenta,
periodic(sin(t)-3,-pi/2,pi/2)
];
\end{texgraph}
```

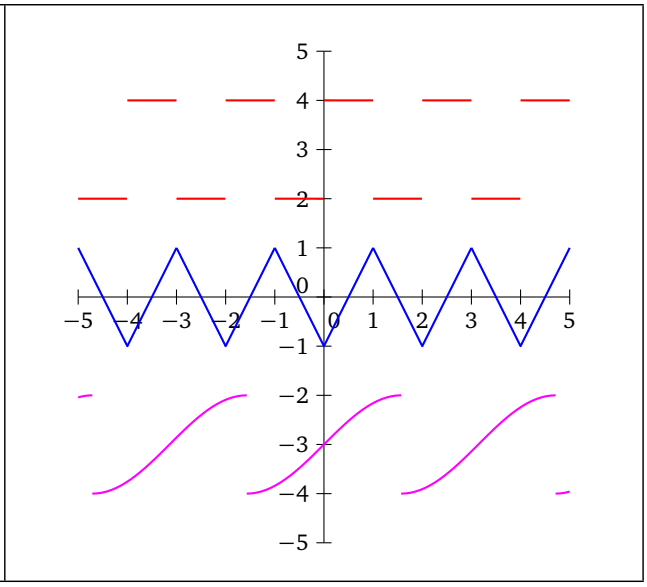


Figure 21: *Periodic Functions*

3.36 Rarc

- **Rarc(<B>, <A>, <C>, <R>, <direction>).**
- Description: works like the macro *Arc* (p. 95) but the circle arc is round even if the coordinate system is not orthonormal, the radius <*R*> is in centimeters.

3.37 Rcercle

- **Rcercle(<A>, <R>)** or **Rcercle(<A>, <B>, <C>).**
- Description: draw a round circle eve if the coordinate system is not orthonormal. In the first form, the radius <*R*> is in centimeters.

3.38 Rellipse

- **Rellipse(<O>, <RX>, <RY> [, inclination]).**
- Description: like the command *Ellipse* (p. 88) but it doesn't take count of the screen coordinate system, radiuses are in centimeters.

3.39 RellipticArc

- **RellipticArc(<B>, <A>, <C>, <RX>, <RY>, <direction(+/-1)> [, inclination]).**
- Description: like the macro *ellipticArc* (p. 101) but this one is not sensitive to the screen coordinate system, radiuses are in centimeters.

3.40 RestoreWin

- **RestoreWin().**
- Description: restore the graphical window and the 2D matrix last saved with the macro *SaveWin* (p. 104).

3.41 SaveWin

- **SaveWin()**.
- Description: saves the currents graphical window and 2D matrix on a stack. That macro is used in parallel with the macro *RestoreWin* (p. 103).
- Example(s): several graphic in one: see *this example* (p. 97).

3.42 Seg

- **Seg(<A>, <B>)**.
- Description: draw the segment line $[A,B]$.

3.43 set

- **set(<name>, <affix center> [, options])**.
- Description: draw a set in a potatoid shape, <*affix center*> is the set center, and the <*name*> is a string with the set name. The optional parameter <*options*> is a list in the form *[option1:= value1, ..., optionN:=valueN]*, the options are:
  - scale := ⟨ positive integer ⟩: represents the scale (1 by default),
  - rotation := ⟨ angle in degrees ⟩: rotate the drawing (0 degree by default),
  - labels := ⟨ 0/1 ⟩: display (or not) the set name.
  - labelsep := ⟨ distance in centimeter ⟩: distance of the label to the edge of the set (0.45cm by default)

  In the options list, it is possible to modify the attributes like LabelStyle for example.
- The macro returns the curve point list that is drawing the set.

3.44 setB

- **setB(<name>, <affix center> [, options])**.
- Description: draw a set in a potatoid shape using Bézier curves, <*affix center*> is the center of that set, and the parameter <*name*> is a string with its name. The <*options*> optional parameter is a list in the form: *[option1:= value1, ..., optionN:=valueN]*. Those options are:
  - scale := ⟨ positive interger ⟩: represents the scale (1 by default),
  - rotation := ⟨ angle in degreed ⟩: gives the drawing inclination (0 degree by default),
  - labels := ⟨ 0/1 ⟩: displays (ot not) the set name.
  - labelsep := ⟨ distance in cm ⟩: distance from the label to the edge of the set (0.45cm by default)

  In the options list, it is possible to modify attributes like coLabelStyle for example.
- The macro returns the control point list of the curve representing the set. That list can be used to determine the intersection between two sets (see *capB* (p. 77)), an union (see *cupB* (p. 78)) or a difference (see *setminusB* (p. 81)).

3.45 size

- **size(<width + i*height> [, ratio(Xscale/Yscale)])**

- Description: sets the graphic sizes: <*width*> and <*height*> (margins included) in cm. If the parameter <*height*> is zero, then it means that width=height.

  If the parameter <*ratio*> is omitted, the scales on the two axes are calulated so that the drawing fits in the framework while keeping the current ratio.

  If the <*ratio*> is 0 then the scales are calculated to get the exact given size (the current ratio is then likely not kept)

  The coordinate system is orthonormal if the parameter <*ratio*> is 1..

  NB: a call to the functions *Fenetre (window) Marges (margins)* or to the macro *view*, will change the graphic size. It is then better to change the margins and the graphical window **before** giving the size.

  The width of a graphic is given with the formula:

  *(Xmax-Xmin)*Xscale+margeG+margeD*

  and the height with:

  *(Ymax-Ymin)*Yscale+margeH+margeB*

3.46 suite (sequence)

- **suite(<f(x)>, <u0>, <n>).**

- Description: graphical representation of the sequence defined by $u_{n+1} = f(u_n)$, the first term <*u0*> till the <*n*>-th term. This macro only represents the “stairs”.

```
\begin{texgraph}[name=suite, file]
Graph image = [
view(-0.1,1.1,-0.1,1.1),size(7.5),
Arrows:=1,Axes(0,1/2+i/4),Arrows:=0,
u0:=0.3,nb:=20, Width:=6,
Color:=darkseagreen,
Droite(1,-1,0), Color:=red,
tMin:=0, tMax:=1,
Width:=8, Cartesienne(4*x*(1-x)),
Width:=6, Color:=blue,
Label(0.5+1.05*i,"$u_{n+1}=4u_n(1-u_n)$"),
LabelDot(u0, "$u_0$","S",1),
suite(4*x*(1-x), u0,nb)
];
\end{texgraph}
```

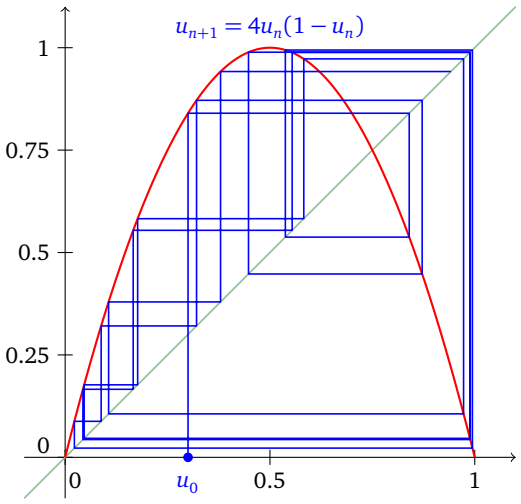


Figure 22: *Suite (sequence) macro usage*

3.47 tangente (tangent)

- **tangente(<f(x)>, <x0> [, length]).**

- Description: draw the tangent to the cartesian curve $y = f(x)$ on the abscissa point <*x0*>, a segment with the given <*length*> is drawn (in cm) or the whole line if the length is omitted.

3.48 tangenteP

- **tangenteP(<f(t)>, <t0> [,length]).**

- Description: draw the tangent to the curve parametrized with <*f(t)*> at the parameter point <*t0*>, a segment of the given <*length*> (in cm) is drawn or the the whole line if the length is omitted.

3.49 view

- **view(<xmin>, <xmax>, <ymin>, <ymax>)** or **view(<[xmin+i*ymin, xmax+i*ymax]>)**
- Description: change the current graphical window and keep the scale. Warning : this is modifying the graphic size, that size can be changed with the macro *size* (p. 105).
- Example(s): in a user graphic element, the command [view(-4, 4,-3, 3), size(12)] will set the window to $[-4,4]\times[-3,3]$, and the graphic size to 12cm while keeping the current ratio. It is important to respect the order (view before size).

3.50 wedge

- **wedge(<B>, <A>, <C>, <r>)**
- Description: draw a wedge defined by the angle $\widehat{BAC}$ with the radius <*r*>.

3.51 zoom

- **zoom(<+/-1>)**
- Description: allows to zoom in/out.

Chapter IX

"Special" macros

1) Special macros

These are the following macros:*Init()*, *Exit()*, *Bsave()*, *Esave()*, *TegWrite()*, *ClicGraph*, *ClicG()*, *ClicD()*, *LButtonUp()*, *RButtonUp()*, *MouseMove()*, *MouseWheel()*, *CtrlClicG()*, *CtrlClicD()* and *OnKey()*. Those macros have a different behaviour from other macros.

1.1 The Init() macro

If a source **.teg* source file, or a **.mod* model file , or a **.mac* macros file, contains a macro called *Init*, then it will be executed as soon as the file is loaded. This macro can be used to initialize some values or for asking the user to give value.

1.2 The Exit() macro

If a file contains a macro called *Exit*, then it is saved in a stack at file loading, and it will be executed next file change, or at program close. that macro is mainly designed to be used within a macro file (**.mac*), for example: restore a given context in its original state.

1.3 Bsave(), Esave() and TegWrite() macros

the macro **Bsave** is automatically executed before the current graphic is exported, while the macro **Esave** is automatically executed after the current graphic is exported.

Those two macros are more likely to be used within the macro files because we have to take count of its existence before redefening them. In fact, they are already defined in the file *TeXgraph.mac*. The first is only calling the macro *UserBsave()*, the second calls *UserEsave()*. The two last do not exist, and can be created by the user in his source file.

The constant *ExportMode* shows the exporting mode, with the following constants: *tex*, *pgf*, *tkz*, *pst*, *eps*, *psf*, *epsc*, *pdf*, *pdfc*, *svg* or *teg*.

The **TegWrite** macro is a very particular one: It is never executed ! More precisely, at saving stage, it is successively saving:

- The window.
- The margins.
- θ and φ values (for the 3D).
- The global variables.
- The macro files to be loaded.
- The macros.
- The graphical elements.

Right before saving the global variables, it is tested if exists a macro called *TegWrite*. If yes, then the command defining the macro is stored in the saved file in the form of a command. Then, at opening of that file, this command will be executed before reading the global variables and all the rest of the code.

1.4 Macros ClicG(), ClicD(), LButtonUp(), RButtonUp(), MouseMove(), MouseWheel(), CtrlClicG() and CtrlClicD()

A left mouse click automatically launch the macro **ClicG(<affix>)** with the affix of the clicked point if the *Ctrl* key is not pressed, else this is the macro **CtrlClicG(<affix>)**. Those macros do not exist by default, can be created by the user.

As soon as the left button is released, the macro **LButtonUp(<affix>)** is executed with the pointer's affix as parameter. This macro, that doesn't exist by default, can be created by the user.

A right-click launch the macro **ClicD(<affix>)** with the pointer's affix as parameter if the *Ctrl* key is not activated, else this is the macro **CtrlClicD(<affix>)**. By default, the macro *ClicD(<affix>)* creates a global variable.

When the right button is released, it launches the macro **RButtonUp(<affix>)** with the pointer's affix as parameter.This macro does not exist by default and can be created by the user.

Moving the mouse launches the macro **MouseMove(<affix>)** with the pointer's affix as parameter. This macro does not exist by default and can be created by the user.

Moving the wheelmouse launches the macro **MouseWheel(<delta>)** with *delta* a positive integer if the wheel is turning forward, negative integer if not. By default, the macro **MouseWheel(<delta>)** is used to zoom in/out on the graphic.

Example(s): draw a polyline using the mouse:

- A global variable *L* is created and set for example to *Nil*.
- A graphical element is created: a *Polyline* called *line* and defined by the command L.
- the macro *ClicG()* is created with the command: [Insert(L, %1), ReCalc(line)].
- The macro *ClicD()* is created with the command: [Del(L, -1, 1), ReCalc(line)] (It removes the last element from the list).

At each left-click, the clicked point is added to the list *L* and the command ReCalc(line) update the graphical element called *line*, the polyline is then built using the mouse.

1.5 The macros ClicGraph() and OnKey()

A left click on one element in the graphical element list (top right) launches the macro **ClicGraph(<code>)** with the clicked element code, that code is defined when the element is created with the function *NewGraph* (p. 51). This macro does not exist by default and can be created by the user.

The key combination : `Ctrl+Maj+<letter>` launches the macro **OnKey(<letter>)**, the argument is a one character string. This macro does not exists by default and can be created by the user.

2) Special macros from interface.mac

Those macros are not to be used in graphical elements, but in the command line or associated with a button or an item in the pull-down list of the graphical interface.

2.1 Apercu (overview)

- **Apercu()**.
- Description: create and display an A4 overview from a pdf export. That macro is associated to the button with an eye in the toolbar: Standard.

2.2 Bouton (button)

- **Bouton(<position>, <name>, <macro>)**.
- Description: button creation, the <*position*> is a complex number $x + iy$ with x and y in pixels, the given name and macro are two strings.
- Example(s): creation (in the CLI) of a button to make a png snapshot and display it:

```
Bouton( RefPoint, "Snapshot", "Snapshot(epsc, 0, ""image.png"", 1)")
```

- To remove buttons, see the command *DelButton* (p. 41).

2.3 geomview

- **geomview()**.
- Description: displays using geomview the current 3D scene built with *Build3d* (p. 147), assuming that this program is installed on your machine and its path is known by your system.
- This macro is associated to a button in the toolbar *Suppléments 3D (3D supplements)*.

2.4 help

- **help(<pdf file> [, directory])**.
- Description: Opens a <*pdf file*> in the given <*directory*>. The file name is without any extension, path and quotes. For example : help(TeXgraph) opens the file *TeXgraph.pdf* located in the directory *DocPath*, this is the default value of the argument <*directory*>. Other example: help(povray, [UserMacPath,"/povray"]).

2.5 javaview

- **javaview()**.
- Description: displays the 3D current scene built with *Build3d* (p. 147) using javaview. This assumes that *java* program is installed on your machine, and the path to the archive *javajview.jar* is correctly set in the config file (menu: *Paramètres/Fichier de configuration (Parameters/ config file)*, restarting the program is necessary).
- That macro is associated with a button from the toolbar *Suppléments 3D (3D supplements)* .

2.6 MouseZoom

- **MouseZoom(<+/-1>)**
- Description: zoom in/out the graphic. This macro is by default linked with the wheelmouse (Mousewheel event).

2.7 NewLabel

- **NewLabel(<affix>)**.
- Description: create a label at the given <*affix*>. The macro opens an input window asking for the text label. That is initially designed to be used within the macro *ClicG()*.

2.8 NewLabelDot

- **NewLabelDot(<affix>, <"name">, <direction> [, DrawDot, distance])**.
- Description: that macro create a global variable called <*"name"*> with the given <*affix*>. It also create a graphical element displaying the variable's name next to the point <*affix*>. The direction is "N" for north, "NE" for north-east,"NO" for north-west, "SO" for south-west...etc, or a list in the form [length, direction] where direction is a compex number, in that second case, the optional parameter <*distance*> is ignored. The point is also displayed when <*DrawDot*> is 1 (default value) and the <*distance*> (in cm) between the dot and the text can be redefined (0.25cm by default). The graphical element calls the macro *LabelDot* (p. 102).
- That macro is linked to a button in the toolbar: Supplément 2D (more 2D tools).

2.9 NewLabelDot3D

- **NewLabelDot3D(<coordinate>, <"name">, <direction> [, DrawDot, distance]).**

- Description: the argument <*coordinate*> is a space point, that can be in the form $M(x,y,z)$ or $[x+iy,z]$. The macro create a global variable called <*"name"*> with the value <*coordinate*>. It also create a graphical element displaying the variable name besides the point <*coordinate*>. The direction (in the screen plane) can be "N" for north, "NO" for north-west, "SO" for south-west, "SE" for south-east...etc, or a list in the form [longueur, direction] where direction is a complex, In that second case, the optional parameter <*distance*> is ignored. The point is also displayed when <*DrawDot*> is 1 (default value) and the <*distance*> (in cm) between the point and the text can be redefined (0.25cm by default). The graphical element calls the macro *LabelDot* (p. 102).

- That macro is associated to a button in the toolbar: Supplément 3D (more 3D tools).

2.10 Snapshot

- **Snapshot(<export>, <screen or printer (0 or 1)>, <"name"> [, show(0/1)]).**

- Description: take a snapshot of the graphic area on the screen, the first argument sets the <*export*> type, one among the following: *eps*, *epsc*, *pdf*, *pdfc* or *bmp*. The second argument sets the image resolution: 0 for the screen (96 dpi) and 1 for the printer (300 dpi), that argument is ignored if the chosen export is *bmp*.The third argument is a string containing the <*"name"*> of the image with one of the extensions (mandatory): *png* ou *jpg*, and with the path. It will be by default the TeXgraph temporary directory. The fourth argument is optional and indicates if the snapshot has to be displayed on screen or not (1 by default). This macro calls the external program : *convert*.

- Example(s): in the command line: Snapshot(epsc, 0, "../capture1.png")

- This macro is associated with a button in the toolbar: Standard.

2.11 VarGlob

- **VarGlob(<affix>).**

- Description: define a global variable with the given <*affix*>. By default, this macro is associated with the mouse right-click.

Chapter X

3D representation

To be fully honest, TeXgraph is not a 3D drawing software, it is working with complex numbers. Though, minimal things can be done in the space:

- A **point** or **vector** with coordinates (x,y,z) is represented by the list: [x+i*y,z] or using the command *M* (p. 63): M(x,y,z). For example the origin is M(0,0,0) or [0,0], the variable *Origin* also exists. It is possible to add or substract two lists, and also multiply a list by a number, ie: linear combinations. Moreover a local or global variable may contain a complexes list, then a variable A could contain a list like [x+i*y,z] representing a so called **3Dpoint** or **3Dvector**.
- A **plane** is represented by one of its points and a normal vector, ie a list: [3Dpoint, 3Dvector].
- A **strait line** is represented by one of its points and a direction vector, ie a list: [3Dpoint, 3Dvector].
- A **facet** is represented by the list of its vertices, that list is ended with the constant *jump*. The vertices order is crucial, defining the facet orientation. Example: face:= [Origin, M(3,0,0), M(0,3,0), jump].
- A **surface** or **polyhedron** is represented by a facet list.

There are two 3D representation types:

1. **individual objects** representation: in this case, the user has to manage the scene: the display order and for example the intersections. This case correspond to the options located on the *3D supplements* toolbar from the graphical interface. That case is suitable if there is only one object or if the scene is very simple. Advantage of the method: the image is lightweight and kept vectorial (circles, arcs,...)
2. **global scene** representation: In that case, the *Build3D()* (p. 147) command is defining the scene and the *Display3D()* (p. 148) command is "calculating" the scene and displaying it. The display order and intersections are then automatically determined. The drawback is that the facets or segments number may explode resulting in a heavy image file and the vectorial aspect for some elements is lost : those are then drawn with segments (arcs, circles,...)

This chapter is dedicated to the first type, the second is described in the following one.

1) Predefined variables

Predefined variables related to 3D representation:

- **theta** and **phi**: used in the calculations of the projections on the screen plane, respectively initialized to $\pi/6$ and $\pi/3$, the first one is the longitud and the second is the colatitude. Those are also editable using a button in the toolbar.
- **sep3D**: constant initialized to *Re(jump)-i*, used as delimiter for the graphics elements in the command *Build3D* (p. 147).
- **AngleStep**: represents the angular step when rotating a 3D object using the arrows buttons. initialized to $\pi/36$ (ie: 5 degrees).
- **Origin**: origin, initialized to *[0,0]*.
- **vecI**: 1st base vector, initialized to *[1,0]*.

- **vecJ**: 2nd base vector, initialized to *[i,0]*.
- **vecK**: 3rd base vector, initialized to *[0,1]*.
- For the 3D window: **Xinf** (=-5), **Xsup** (=5), **Yinf** (=-5), **Ysup** (=5), **Zinf** (=-5) and **Zsup** (=5).
- **HideStyle**: initialized to dotted, for the style of the hidden edges,
- **HideWidth**: initialized to *Nil*, for the thickness of the hidden edges,
- **HideColor**: initialized to *Nil*, for the color of the hidden edges.

2) Commands for 3D

2.1 Aretes (edges)

- **Aretes(<facet list>)**
- Description: that function returns the edges list of the object represented by the *<facet list>*. An edge is itself a list in the form: [point3D1, point3D2, jump] and the imaginary part of the *jump* constant is 0 for a hidden edge, or 1 for a visible edge.
- Example(s): section of a tetrahedron:

```
\begin{texgraph}[name=Aretes, file]
Graph image = [
 view(-2,3,-2,4.5),Marges(0,0,0,0),size(7.5),
 plan:=[M(1.5,0,0), -vecI],
 S:= Section(plan,
     Tetra(Origin,3*vecI,3*vecJ,4*vecK)),
 A:=Aretes(S), Width:=12, DrawAretes(A)
];
\end{texgraph}
```

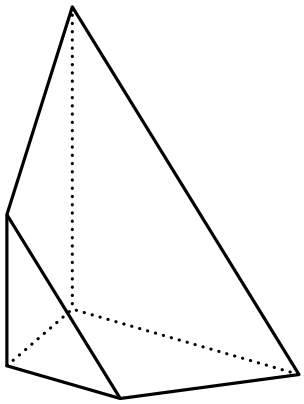


Figure 1: *Aretes (edges)*

2.2 Bord (outline)

- **Bord(<facet list>)**
- Description: that function returns the edges list that is building the outline of the object represented by the *<facet list>*. An edge is itself a list in the form: [point3D1, point3D2, jump] and the imaginary part of the *jump* constant is 0 for a hidden edge, or 1 for a visible edge. An edge is considered as a part of the outline if it belongs to only one facet.

2.3 ComposeMatrix3D

- **ComposeMatrix3D(<[vector3D1, vector3D2, vector3D3, vector3D4]>)**

- Description: that function compose the matrix <*[vector3D1, vector3D2, vector3D3, vector3D4]*> with the 3D current matrix (thus affecting the projection function *Proj3D* (p. 117)). This matrix represents the analytic expression of an affine map of the space, this is a three vectors list: *vector3D1* is the translation vector, *vector3D2* is the first column vector of the matrix of the linear part in the canonical base, *vector3D3* is the second column vector of the matrix of the linear part, and *vector3D4* is the third column vector of the matrix of the linear part. For example, the identity matrix is : [M(0,0,0), M(1,0,0), M(0,1,0], M(0,0,1)] or [Origin, vecI, vecJ, vecK] (this is the default matrix). (See also the commands *GetMatrix3D* (p. 115), *SetMatrix3D* (p. 119), and *IdMatrix3D* (p. 115)).

- If f is an affine map of the space then its linear part is Lf(X)=f(X) - f(Origin), the translation vector is f(Origin), and its matrix is : [f(Origin), Lf(vecI), Lf(vecJ), Lf(vecK)].

2.4 ConvertToObj

- **ConvertToObj(<facet list>, <vertices>, <facets>)**

- Description: that functions converts the <*facet list*> into the format *obj*, more precisely the two last arguments have to be variables. the variable <*vertices*> gets in output the vertices list (without repetitions) and the variable <*facets*> gets the facet list (separated by the *jump* constant) not with the vertices coordinates but the order of appearance in the vertices list. The function returns a complex $a+ib$ where a is the vertices number and b the faces number. That command is used in the *obj*, *geom* and *jvx* exports.

  Warning: for a great number of facets (thousand or more), that command takes a certain time (2 to 3 mn for about 20 000 facets)!

- The command *MakePoly* (p. 116) is the inverse operation.

- Example(s): executing ConvertToObj(Tetra(Origin, 2*vecI, 3*vecJ, vecK), S, F) returns the value 4+4*i, which means 4 vertices and 4 facets. The variable S contains as output the list: [0,0,3*i,0,2,0,0,1], and the variable F contains the list: [1,2,3,jump,1,3,4,jump,3,2,4,jump,1,4,2,jump].

2.5 ConvertToObjN

- **ConvertToObjN(<facet list>, <vertices>, <facets>)**

- Description: that function converts the <*facet list*> to the *obj* format, more precisely the two last arguments must be variables. The variable <*vertices*> gets in output the vertices list (without duplicates) where **each vertex is followed by its normal unit vector** (that vector is the average of the vectors normal to the facets sharing the vertex). The variable <*facets*> gets the facet list (delimited by the constant *jump*) not with the vertices coordinates, but its order of appearance in the vertices list. The function returns a complex $a+ib$ where a is the vertices number and b the faces number. That command is used in the *obj* and *geom* exports.

  Warning: for a great number of facets (thousand or more), that command takes a certain time !

- Example(s): executing the command:

  ConvertToObjN(Tetra(Origin,2*vecI,3*vecJ,vecK),S,F)

  returns the value 4+4*i, that means 4 vertices and 4 facets. The variable S contains in output the list:

  [0, 0, -0.57735026918962-0.57735026918962*i, -0.57735026918962,
  3*i, 0, -0.87287156094397 +0.43643578047198*i, -0.21821789023599,
  2, 0, 0.50709255283711 -0.84515425472851*i, -0.1690308509457,
  0, 1, -0.45584230583855 -0.56980288229819*i, 0.68376345875782],

  and the variable F contains in output the list: [1, 2, 3, jump, 1, 3, 4, jump, 3, 2, 4, jump, 1, 4, 2, jump].

2.6 Clip3DLine

- **Clip3DLine(<3Dpoint list>, <plane>, <closed(0/1)> [, behind])**

- Description: the function is clipping the point list with the given <*plane*>, represented in the form of a list [3Dpoint, 3Dvector] where the vector is normal to the plane and 3Dpoint is a point in the plane, the function returns the part of the list in the half space containing the normal vector (the front of the plane). The third argument precise if the list has to be closed or not. The last argument is optional, and must be a variable name, it will get in output the part of the list located behind the plane.

- Example(s): cut an helix:

```
\begin{texgraph}[name=Clip3DLine, file]
Graph image = [
 view(-5,5,-5,5),view3D(-4,4,-4,4,-4,4),
 size(7.5),plan:=[Origin,vecJ],
 C:=for t from -2*pi to 2*pi step 0.1 do
   [2*exp(i*t),t/3] od,
 L:=Clip3DLine(C,plan,0,L'),
 Ligne3D([M(0,-4,0), Origin],0),
 Color:=blue, Ligne3D(L',0),
 FillStyle:=full, FillColor:=gray,
 FillOpacity:=0.8, Color:=black,
 DrawPlan( [Origin, vecJ], vecI,5,5),
 FillStyle:=none,
 Yinf:=0, Axes3D(0,0,0),
 Color:=red, Ligne3D(L,0)
];
\end{texgraph}
```

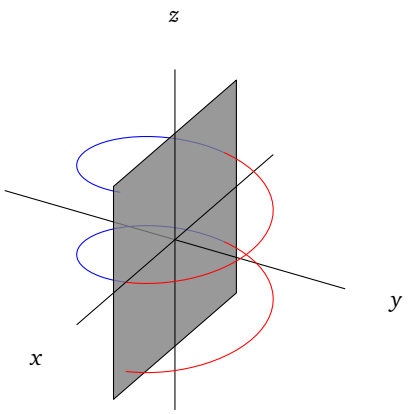


Figure 2: *Clip3DLine*

2.7 ClipFacet

- **ClipFacet(<facet list>, <plane>)**
- Description: a facet is represented under the form of a 3D points list ended by the constant *jump*. Those points should be coplanar. Example: *[Origin, M(0,1,0), M(0,0,3), jump]* is a facet. The facets are oriented by the vertices order of appearance.

  That function cuts all the facets in the list with the <*plane*>. That plane is represented under the form of a list [A,u] where A is a 3Dpoint and u a 3Dvector, ie: the plane passing through A normal to the vector u. Only the part of the facets in the half plane containing u is kept. The function returns the list of the cut facets.
- Example(s): the command [P:=Tetra(Origin, vecI, vecJ, vecK), ClipFacet(P, [M(0,0,0.5), -vecK])] defines a tetrahedron called P and returns the part of P located below the plane (under the form of facets).

```
\begin{texgraph}[name=ClipFacet, file]
Graph image = [
 view(-1,1,-1,1), view3D(-2,2,-2,2,-2,2),
 Marges(0,0,0,0), size(7.5),
 P:=Tetra(Origin, vecI, vecJ, vecK),
 P':=ClipFacet(P, [M(0,0,0.5), -vecK]),
 DrawFacet( P', [color:=pink])
];
\end{texgraph}
```

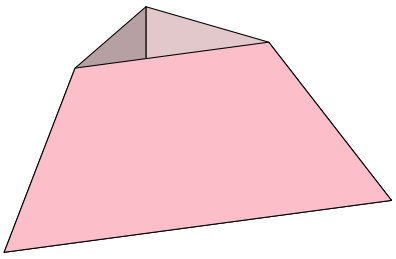


Figure 3: *ClipFacet*

2.8 DistCam

- **DistCam(<distance>)** ou **DistCam()**.
- Description: permits to change the camera position by modifying its <*distance*> to the origin. If the distance is too weak, the rendering may be not correct. If the argument is empty, the function returns the distance camera-screen, else is returns *Nil*. See also *ModelView* (p. 116) and *PosCam* (p. 117).

2.9 Fvisible

- **Fvisible(<facet>)**
- Description: that function returns 1 or 0 according to the fact that the <*facet*> is visible or not. A facet is visible if its normal vector is directed toward the observer (ie: the scalar product with the vector facet-observer, is positive). That function takes in count the current 3D transformation matrix and the projection type.

2.10 GetMatrix3D

- **GetMatrix3D()**
- Description: that function returns the current 3D matrix. (See also the commands *ComposeMatrix3D* (p. 112), *SetMatrix3D* (p. 119), and *IdMatrix3D* (p. 115))

2.11 GetSurface

- **GetSurface(<f(u,v)> [, uMin+i*uMax, vMin+i*vMax, uNbLg+i*vNbLg]).**
- Description: returns the facets list of the surface parametrized by <*f(u,v)*> where f is a function of two variables u and v, with values in the space. The second parameter represents the interval of the parameter u ($[-5,5]$ by default), the third parameter represents the interval of the parameter v ($[-5;5]$ by default), the fourth parameter represents, under complex form, the lines number for u and the number of lines for v (25 lines by default).
- Example(s): surface drawing:

```
\begin{texgraph}[name=GetSurface, file]
Graph image = [
 Marges(0,0,0,0), size(7.5),
 S:=GetSurface([u+i*v,sin(u)+cos(v)],
          pi*(-1+i),pi*(-1+i)),
 DrawFacet(S,[color:=Rgb(0.74,0.73,1)])
];
\end{texgraph}
```

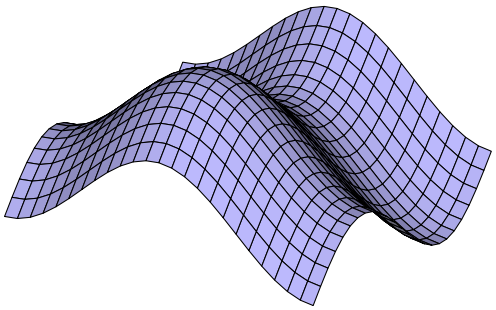


Figure 4: *GetSurface*

2.12 IdMatrix3D

- **IdMatrix3D()**
- Description: changes the current 3D matrix into the identity matrix. (See also the commands *ComposeMatrix3D* (p. 112), *SetMatrix3D* (p. 119), and *GetMatrix3D* (p. 115))

2.13 Inserer3D

- **Inserer3D(<list>, <3Dpoint> [, epsilon])**
- Description: the first argument must be a variable, the function adds the <*3Dpoint*> in the <*list*> without adding duplicates, and returns the position (integer) of that point in the variable <*list*> that is updated. The comparison test is performed to the nearest <*epsilon*> (0 by default).

2.14 MakePoly

- **MakePoly(<3Dpoints list>, <facets list (obj format)>**

- Description: that command takes as input a *<3Dpoints list>* that is representing vertices, and a *<facets list>* using the *obj* format, that is to say that the facets do not contain the vertices coordinates but the order of appearance in the vertices list. The command returns as output the facets list built with the vertices coordinates. That list can then be drawn by one of the macros *DrawPoly* (p. 145), *DrawFacet* (p. 144).

2.15 ModelView

- **ModelView(<ortho/central>)** or **ModelView()**.

- Description: modify the projection mode *ortho* for the orthographic projection and *central* for the central projection (see *Proj3D* (p. 117)). If the argument is empty, the function returns the current projection mode, else it returns *Nil*. See also *PosCam* (p. 117) and *DistCam* (p. 114).

2.16 Mtransform3D

- **Mtransform3D(<3Dpoints list>, <3dmatrix>)**

- Description: that function returns the *<3Dpoints list>* transformed by the *<3dmatrix>*. This marix represents the analytic expression of an affine map of the space, this is a three vectors list: the *vector3D1* is the translation vector, *vector3D2* is the first column vector of the matrix of the linear part in the canonical base, *vector3D3* is the second column vector of the matrix of the linear part, and *vector3D4* is the third column vector of the matrix of the linear part. For example, the matrix of the identity is written like the following: [M(0,0,0), M(1,0,0), M(0,1,0), M(0,0,1)] or [Origin, vecI, vecJ, vecK] (this is the default matrix). (See also the commands *GetMatrix3D* (p. 115), *ComposeMatrix3D* (p. 112), and *IdMatrix3D* (p. 115)).

```
\begin{texgraph}[name=Mtransform3D, file]
Graph image = [
 view(-5,5,-3,6), Marges(0,0,0,0), size(7.5),
 viewDir(115,70),
 P:=shift3d(Pyramide( [Origin,M(1,-1,0),M(3,1,0),
          M(3,3,0),M(1,4,0)],M(1,1,3)),2*vecJ),
 miroir:=[M(-4,0,0),M(4,0,0),M(4,0,5),M(-4,0,5),jump],
 P':=reverse3d( Mtransform3D( P,
          matrix3d(sym3d(M,[Origin,vecJ])))),
 FillStyle:=full, FillColor:=brown, Width:=8,
 DrawFacet( P', [color:=FillColor]),
 DrawFacet( miroir,
          [FillOpacity:=0.5, color:=lightgray]
      ),
 DrawPoly(P,4)
];
\end{texgraph}
```

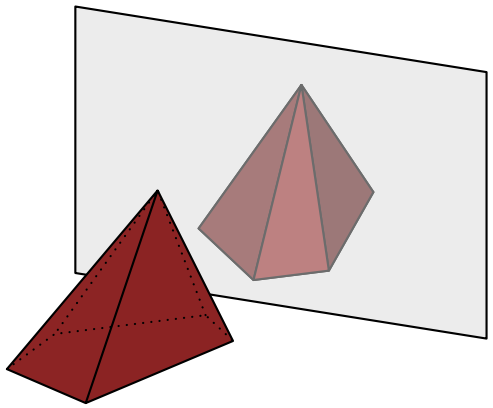


Figure 5: *The Mtransform3D() command*

2.17 Norm

- **Norm(<3Dvector>)**.

- Description: returns the *<vector>*'s norm.

2.18 Normal

- **Normal()**.

- Description: returns the unit vector normal to the projection plane and oriented towards the observer. That vector is $M(\sin(\phi)\cos(\theta), \sin(\phi)\sin(\theta), \cos(\phi))$.

2.19 PaintFacet

- **PaintFacet(<facets list>, <color+i*(non oriented 0/1)>, < (backculling 0/1)+i*contrast>)**.
- Description: that command returns the <*facets list*> after adding in the imaginary part of each constant *jump* that is located between the facets, a <*color*> (in fact this is the color+2). If the argument <*non oriented*> is 1, then the front and back of the facets are not distinguished. If the argument <*backculling*> is 1 then the non visible facets are eliminated. The argument <*contrast*> is a positive number or zero that permmits or not to accentuate the color contrast between the facets, with the value 0 the color will be solid.
- That command is used by the drawing macro *DrawFacet* (p. 144).

2.20 PaintVertex

- **PaintVertex(<facets list>, <color+i*(non oriented 0/1)>, < (backculling 0/1)+i*contrast>)**.
- Description: that command returns the <*facets list*> after adding in the imaginary part of the z-coordinate of each vertex, a <*color*> (in fact this is the color + 2). If the argument <*non oriented*> is 1, then the front and the back of the facets are not distinguished. If the argument <*backculling*> is 1 then the non visible facets are eliminated. The argument <*contrast*> is a positive number or zero that permits to accentuate or not the color contrast between facets, with the value 0 the color will be solid. The execution of that command can be quite long for a great number of facets.
- That command is used by the drawing macro *DrawFacet* (p. 144).

2.21 PosCam

- **PostCam(<3Dpoint>)** or **PostCam()**.
- Description: modifies the camera position. It always targets the origin and the projection plane is the plane passing through the origin and perpendicular to the origin-observer axis (this the screen plane). If the argument is empty, the command returns the current camera position. See also *ModelView* (p. 116) and *DistCam* (p. 114).

2.22 Prodvec

- **Prodvec(<vector3D1>, <vector3D2>)**.
- Description: returns the result of the vectorial product between the two vectors.

2.23 Prodscal

- **Prodscal(<vector3D1>, <vector3D2>)**.
- Description: returns the result of the scalar product between the two vectors.

2.24 Proj3D

- **Proj3D(< 3Dpoint list >)**.
- Description: that function *Proj3D* calculate and returns the list of the projections of the 3Dpoints on the plane passing through the origin and normal to the vector *Normal()* with coordinates $(sin(\varphi)cos(\theta), sin(\varphi)sin(\theta), cos(\varphi))$ [oriented towards the observer]. The 3Dpoints list may contain the constant *jump*, it will be copied in the result.

```
\begin{texgraph}[name=coord, file]
Graph image = [
view(-3.5,4.5,-3,4), view3D(-3.5,3.5,-3.5,3.5,-3.5,3.5), size(7.5),
Marges(0,0,0,0),
A:=M(3,3,3), Width:=8, Arrows:=1, Axes3D(0,0,0),
LabelDot3D(Origin, "$O$","NO",1),
Arc3D(px(A),Origin,pxy(A),1.5,1), Arc3D(pz(A),Origin,A,1.5,1),
Arrows:=0, LineStyle:=userdash,
Ligne3D([px(A),pxy(A),py(A), jump, A, pxy(A),Origin],0),
Arrows:=1, LineStyle:=solid, Ligne3D([Origin ,A],0),
LabelDot3D(A,"$\vec{n}$", "NE"),
Label(0.1228-1.0377*i,"$\theta$"),
Label(0.3509+1.3396*i,"$\varphi$")
];
\end{texgraph}
```

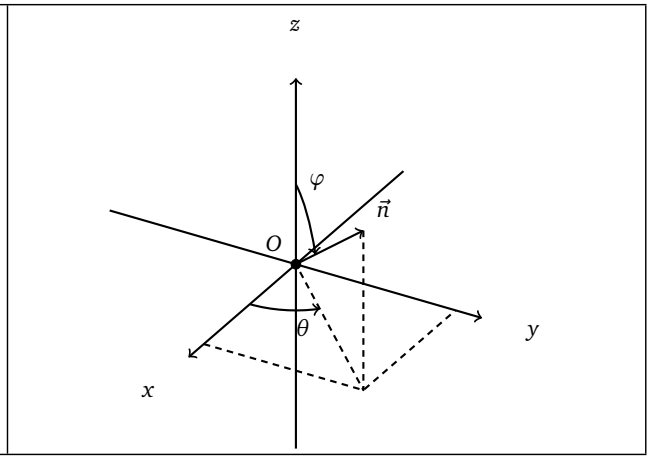


Figure 6: *Space Coordinates*

- There are two types of projection: orthographic and central. The mode is changed using the command *ModelView* (p. 116).
  - **orthographic projection**: orthogonal projection on the plane passing through the origin and normal to the vector *Normal()* (that plane corresponds to the screen plane). That means the observer is at the infinity. This projection has the advantage of being linear, and conserving the barycenters, we can then draw a BÉZIER curve in the space using the function *Bezier* (p. 86) of the plane: if A, B and C are three space points then a graphic element *Courbe/Bezier* can be created with the command Proj3D([A,C,B]) and we will see the drawing of the projection of the Bézier curve of ends A and B with C as a control point.
  - **central projection**: the observer is at certain point C of the space (other than the origin), the vector *Normal()* corresponds then to the vector $\vec{OC}$ normalized. The projection is always performed on the plane P passing through the origin and normal to the vector *Normal()*, the following manner: the projection of a point M is the intersection of the line (CM) with the plane P. If the distance is too short, the display is not always correct. The commands dedicated to this projection mode are *PosCam* (p. 117) and *DistCam* (p. 114).
- Example(s): representation of a plane curve in the space:

```
\begin{texgraph}[name=Proj3D, file]
Graph image = [
ModelView(central), view(-6,6,-6,6),
view3D(-5,5,-4,4,-4,4),
Marges(0,0,0,0), size(7.5),
L:= for z in Get(Cartesienne(sin(x)),0)
   do [z,0] od,
Arrows:=1, Axes3D(0,0,0),
Arrows:=0, Width:=8,
Color:=red, Ligne( Proj3D(L), 0)
];
\end{texgraph}
```

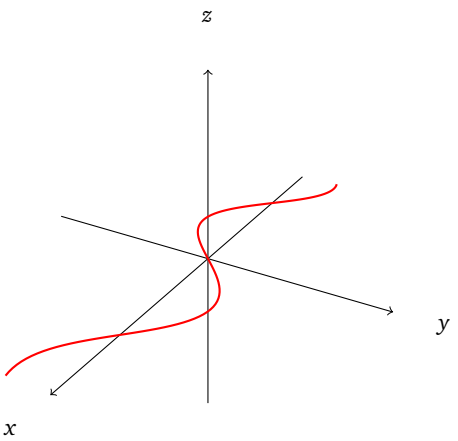


Figure 7: *Proj3D*

2.25 ReadObj

- **ReadObj(<"file">, <built facets>, <built lines> [, <vertices>, <facets obj>, <lines obj>])**

- Description: that command reads a <*"file"*> with the *obj* format (the extension is mandatory). The following arguments must be variables. The variable <*built facets*> gets the facets list ready to be drawn, same with the variable <*built lines*>. The optional arguments are also variables and get datas from the file under the *obj* format: list of the <*vertices*>, <*facets obj*> and <*lines obj*> with the appearance number of the vertices in the list.

- Example(s): reading a file *triceratops.obj* loaded from the address:

  http://www.cs.technion.ac.il/~irit/data/Viewpoint/

  The image is get from a snapshot (snapshot button) with an *eps* export and a conversion into *png*.

```
\begin{texgraph}[name=triceratops, file]
Graph image = [
 view(-9,10,-7,7), Marges(0,0,0,0),
 size(7.5), viewDir(37,85),
 SetMatrix3D([Origin, vecJ, vecK, vecI]),
 background(full, lightblue),
 ReadObj("obj/triceratops.obj", stock1, stock2),
 draw("SmoothFacet", stock1, [color:=gray])
];
\end{texgraph}
```

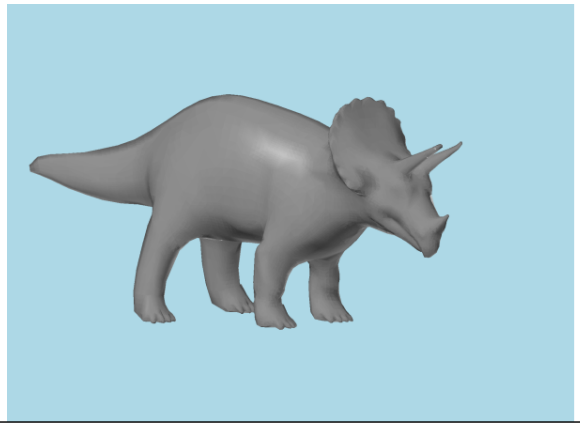


Figure 8: *ReadObj*

2.26 SetMatrix3D

- **SetMatrix3D(<[vector3D1, vector3D2, vector3D3, vector3D4]>)**

- Description: that function changes the current matrix into <*[vector3D1,vector3D2,vector3D3,vector3D4]*> (thus affecting the projection function *Proj3D* (p. 117)). That matrix represents the analytic expression of an affine map of the space, this is a three vectors list: *vector3D1* is the translation vector, *vector3D2* is the first column vector of the matrix of the linear part in the canonical base, *vector3D3* is the second column vector of the matrix of the linear part, and *vector3D4* is the third column vector of the matrix of the linear part. For example, the mtrix of the identity is : [M(0,0,0), M(1,0,0), M(0,1,0], M(0,0,1)] or [Origin, vecI, vecJ, vecK] (this is the default matrix). (See also the commands *GetMatrix3D* (p. 115), *ComposeMatrix3D* (p. 112), and *IdMatrix3D* (p. 115)).

- If f is an affine map of the space, then its linear part is Lf=f-f(Origin), the translation vector is f(Origin), and its matrix is : [f(Origin), Lf(vecI), Lf(vecJ), Lf(vecK)].

2.27 Sommets (vertices)

- **Sommets(< facets list>)**

- Description: that function returns the vertices list, without duplicates.

2.28 SortFacet

- **SortFacet(<facets list> [, (backculling 0/1)+i*contrast])**

- Description: a facet is a 3Dpoints list ended with the constant *jump*, those points should be coplanar. Example: *[Origin, M(0,1,0), M(0,0,3), jump]* is a facet. The facets are oriented by the order of appearence of the vertices. That function sorts the facets from the farest to the nearest to the observer (this is the height of the center of gravity on the axis oriented towards the observer that is taken in count), and returns the resulting sorted list (the input list is not changed).

  The optional argument is a complex in the form (0/1)+i*(0/1).

  If the real part is 1: the non visible facets are removed from the sort. A facet is visible if its normal unit vector (its direction is determined by the orientation of the facet) has the “same direction” as the unit vector oriented towards the obsever (the scalar product with the vector n() is positive).

  If the real part is 0: all the facets are sorted.

If the imaginary part is 1: to each facet a coefficient is given (the scalar product between the unit vecteur normal to the facet and *Normal()* that is used to nuance the fillcolor if *FillStyle=full*. That coefficient is stored in the imaginary part of the constant *jump* that is ending the facet. The graphical function *Ligne* (p. 90) reads that coefficient, that is between 0 and 1 for a visible facet, and multiply the rgb components of the fillcolor by that coefficient before painting.

If the imaginary part is 0: the fillcolor won't be nuanced.

By default, the optional argument is zero.

3) 3D related mathematical macros

3.1 aire3d

- **aire3d(<List of convex facets>)**.
- Description: returns the sum of the areas of the <*list of convex facets*>.

3.2 angle3d

- **angle3d(<vector3D1>, <vector3D2>)**.
- Description: returns the angular difference between the two space vectors.

3.3 bary3d

- **bary3d(<[point3D1, coef1, point3D2, coef2, ...]>)**.
- Description: returns the centroid of the weighted system <*[(point3D1, coef1), (point3D2, coef2), ...]*>.

3.4 det3d

- **det3d(<vector3D1>, <vector3D2>, <vector3D3>)**.
- Description: returns the determinant of the three space vectors.

3.5 interDD

- **interDD(<line>, <line> [, epsilon])**.
- Description: intersection line-line. The lines are in the form: [3Dpoint, direction vector]. If the lines are coplanar and not parallel, the macro returns a 3Dpoint. By default the tolerance <*epsilon*> is 1E-10.

3.6 interDP

- **interDP(<line>, <plane>)**.
- Description: intersection line-plane. The line is in the form: [3Dpoint, direction vector] and the plane : [3Dpoint, 3D normal vector], the macro returns a line and a 3D point.

3.7 interLP

- **interLP(<3D points list>, <plane> [, close(0/1)])**.
- Description: that macro returns the list of the intersection points between the polyline built with the <*3D points list*> and the <*plane*>. The plane is in the form [3Dpoint, 3Dnormal vector]. The optional parameter <*close*> indicate if the line has to be closed or not (0 by default).

3.8 interPP

- **interPP(<plane1>, <plane2>)**.
- Description: plane-plane intersection. Each plane is in the form: [3Dpoint, 3D normal vecteur] and the macro returns a line in the form of a list of the type [3Dpoint, direction vector].

3.9 IsAlign3D

- **IsAlign3D(<3D point list> [, epsilon])**.
- Description: returns 1 if the 3D points of the <*list*> are aligned, 0 if not. By default the tolerance <*epsilon*> is 1E-10. The <*list*> must be without the constant *jump*.

3.10 isobar3d

- **isobar3d(<3Dpoint list>)**.
- Description: returns the centroid of a space points list, the constant *jump* is ignored.

3.11 IsPlan

- **IsPlan(<3Dpoints list> [, epsilon])**.
- Description: returns 1 if the 3D points of the <*list*> are coplanars, 0 if not. By default the tolerance <*epsilon*> is 1E-10. The <*list*> must be without the constant *jump*.

3.12 KillDup3D

- **KillDup3D(<3D points list> [, epsilon])**.
- Description: returns the <*3D points list*> without duplicates, the comparisons are done to the nearest <*epsilon*> (<*epsilon*> is 0 by default).

3.13 length3d

- **length3d(<3Dpoint list> [, closed(0/1)])**.
- Description: returns the length of the <*3Dpoint list*> unsing the graphic units, the 3D coordinate system is orthonormal, the <*3D points list*> can represent a edges list or a facet. By default the parameter <*closed*> is 0.

3.14 Merge3d

- **Merge3d(<3Dpoints list>)**.
- Description: this macro allows to merge pieces of lists to get maximum length components and returns the resulting list. It is equivalent to the command *Merge* (p. 49) in the space.

3.15 n

- **n()**.
- Description: macro equivalent to the command *Normal()* (p. 116). Used in immediate development (`\n`) it is replaced with the command *Normal()*.

3.16 Nops3d

- **Nops3d(<3Dpoint list>)**.
- Description: returns the number of the 3Dpoints in the <*list*>, and adds the eventuals *jump*.
- Example(s): the command `Nops3d([Origin, jump, 1+i,1, M(1,2,3), jump])` returns the value 5.

3.17 normalize

- **normalize(<3Dpoint>)**.
- Description: returns the normalized vector.

3.18 permute3d

- **permute3d(<3Dpoint list>)**.
- Description: modify the <*3Dpoint list*> by placing the first 3D element (1 3Dpoint = 2 affixes) at the end, the <*3D point list*> must be a variable. If the first element of that list is the constant *jump* then it will moved to the end of the list (in that case only one affix is moved).

3.19 planEqn

- **planEqn(<[a,b,c,d]>)**.
- Description: returns the plane with equation $ax + by + cz = d$ under the form [3Dpoint, 3Dvector], ie: a point and a normal vector.

3.20 Pos3d

- **Pos3d(<3Dpoint>, <3D point list> [, epsilon])**.
- Description: returns the position list of the <*3Dpoint*> in the <*list*>, the comparison is done to the nearest <*epsilon*> (0 by default).
- Example(s): the command Pos3d(M(1,1,0), [Origin, jump, M(1,1,1), M(1,2,3)]) gives Nil, and Pos3d(M(1,1,1), [Origin, jump, M(1,1,1), M(1,2,3)]) gives the value 3.

3.21 purge3d

- **purge3d(<3Dpoint list> [, epsilon])**.
- Description: returns the <*3Dpoint list*> after removing the consecutive points that were equal, and removing the components of cardinality strictly less than 2. The test is performed to the nearest <*epsilon*> (1E-10 by default).

3.22 px, py, pz, pxy, pxz, pyz

- **px(<point3D>)**: projection on Ox.
- **py(<point3D>)**: projection on Oy.
- **pz(<point3D>)**: projection on Oz.
- **pxy(<point3D>)**: projection on xOy.
- **pxz(<point3D>)**: projection on xOz.
- **pyz(<point3D>)**: projection on yOz.

3.23 replace3d

- **replace3d(<3Dpoint list>, <position>, <replacement value>)**.
- Description: modifies the variable <*3Dpoint list*> by replacing the number <*position*> element with the <*value*>, the function returns *Nil*.
- Example(s): if S=[Origin, jump, M(1,1,1), M(1,2,3), jump], then after the command replace3d(S,3, [M(1,0,1),M(0,1,1)]), you will get S=[Origin, jump, M(1,0,1),M(0,1,1), M(1,2,3), jump], ie: S=[0,0,jump,1,1,i,1,1+2*i,3,jump].

3.24 reverse3d

- **reverse3d(<3Dpoint list>)**.
- Description: returns the <*3Dpoint list*> by reversing each component of that <*list*> (two components are delimited by a *jump*). But the <*list*> is not modified.
- Example(s): the command S:=reverse3d([Origin, M(1,1,0), jump, M(1,1,1), M(1,2,3), jump]) gives S=[M(1,1,0), Origin, jump, M(1,2,3), M(1,1,1), jump], ie: S=[1+i,0,0,0,jump,1+2*i,3,1+i,1,jump].

3.25 viewDir

- **viewDir(<3Dvector>)** or **viewDir(<theta>, <phi>)** or **viewDir(xOy/yOz/xOz)**
- Description: in the first version, the macro modifies the vector normal to the projection plane (see *n()* (p. 121)) so that it corresponds to the normalized <*3Dvector*>. In the second version, it modifies the view angles <*theta*> and <*phi*>, with the given values (in degrees). In the third version there are three possible arguments: *xOy* or *yOz* or *xOz*, thus defining the projection plane.

```
\begin{texgraph}[name=viewDir, file]
Mac
   dessin = [ BoxAxes3D(grid:=1, zlabelstyle:=right,
         zlabelsep:=0.15, xlabelsep:=0.25,
         ylabelsep:=0.25,
         xlegendsep:=0.35, ylegendsep:=0.35,
         FillColor:=lightcyan),
      Ligne3D(SortFacet(stock),1), RestoreWin()];
Cmd      [tMin:=-5,tMax:=0,DotScale:=1+i];
Graph objet1 = [view(-6,6,-6,6), Marges(0,0,0,0),
      size(7.5),
      view3D(-3,3,-3,3,-3,3),
      S:=GetSurface([u+i*v,2*sin(u)+cos(v)],
            -3+3*i,-3+3*i),
      stock:=for facette in S By jump do
         z:=Zde(isobar3d(facette)),
         facette,
         ColorJump(Hsb(270*(Zsup-z)/(Zsup-Zinf),1,1))
        od,
      FillStyle:=full, LabelSize:=tiny,
      ModelView(central), SaveTphi(), SaveWin(),
      view(-6,0,0,6), ChangeWinTo([-8-7*i,6+6*i]),
      dessin(), SaveWin(), ModelView(ortho),
      view(0,6,0,6), ChangeWinTo([-6-6*i,4+5*i]),
      viewDir(xOy), dessin(), SaveWin(),
      view(-6,0,-6,0), ChangeWinTo([-6-6*i,4+5*i]),
      viewDir(yOz), dessin(), SaveWin(),
      view(0,6,-6,0), ChangeWinTo([-6-6*i,4+5*i]),
      viewDir(xOz), dessin(), RestoreTphi() ];
\end{texgraph}
```

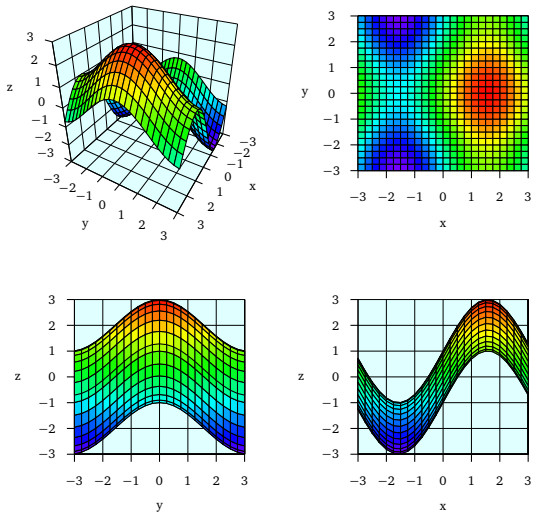


Figure 9: *Examples of views*

3.26 visible

- **visible(<3Dvector>)**.
- Description: returns 1 if the <*3Dvector*> is oriented towards the observer (positive scalar product).

3.27 Xde, Yde, Zde

- **Xde(<point3D>)**: returns the abscissa.
- **Yde(<point3D>)**: returns the ordinate.
- **Zde(<point3D>)**: returns the z-coordinate.

4) Geometric transformations of the space

4.1 antirot3d

- **antirot3d(<3Dpoint list>, <line>, <alpha>)**.
- Description: calculate the images of the list by the rotation with axis the <*line*> and direction angle the real <*alpha*>, composed with the reflexion with respect to the plane orthogonal to the <*line*>. The <*line*> is a list in the form: [3Dpoint, 3D direction vector], the direction vector is orienting the line and the orthogonal plane is the one passing through the same 3Dpoint.

4.2 defAff3d

- **defAff3d(<name>, <A>, <A'>, <linear part>)**
- Description: that function creates a macro called <*name*> representing the affine map that is transforming <*A*> into <*A'*>, whose linear part is the last argument. That linear part is in the form of a three 3Dvectors list: [Lf(vecI), Lf(vecJ), Lf(vecK)] where Lf is the linear part of the transformation.

4.3 dproj3d

- **dproj3d(<3Dpoint list>, <line>)**.
- Description: calculate the images of the list by the orthogonal projection onto the <*line*>. The <*line*> is a list in the form: [3Dpoint, 3D direction vector].

4.4 dproj3dO

- **dproj3dO(<3Dpoint list>, <line>, <3D normal vector>)**.
- Description: calculate the images of the list by the oblique projection onto the <*line*> perpendicular to the <*normal vector*>. The <*line*> is a list in the form: [3Dpoint, 3D direction vector].

4.5 dsym3d

- **dsym3d(<3Dpoint list>, <line>)**.
- Description: calculate the images of the list by the orthogonal symmetry with respect to the <*line*>. The <*line*> is a list in the form: [3Dpoint, 3D direction vector].

4.6 dsym3dO

- **dsym3dO(<3Dpoint list>, <line>, <3D normal vector>)**.
- Description: calculate the images of the list by the oblique symmetry with respect to the <*line*> perpendicular to the <*normal vector*>. The <*line*> is a list in the form: [3Dpoint, 3D direction vector].

4.7 ftransform3d

- **ftransform3d(<3Dpoint list>, <f(M)>)**
- Description: returns the list of the images of the points of the <*list*> by the function <*f(M)*>, that can be an expression function of M or a macro with argument M, representing a 3Dpoint.

4.8 hom3d

- **hom3d(<3Dpoint list>, <3Dpoint>, <lambda>)**.
- Description: calculate the images of the list by the homothety of center <*3Dpoint*> and ratio <*lambda*> (real).

4.9 inv3d

- **inv3d(<3Dpoint list>, <3Dpoint>, <R>)**.
- Description: calculate the images of the list by the inversion with respect to the sphere of center <*3Dpoint*> and radius <*R*>.

4.10 proj3d

- **proj3d(<3Dpoint list>, <plane>)**.
- Description: calculate the list of the orthogonal projection of the points from the <*3Dpoint list*> onto the <*plane*>. The <*plane*> is a list in the form: [3Dpoint, 3D normal vector].

4.11 proj3dO

- **proj3dO(<3Dpoint list>, <plane>, <vector>)**.
- Description: calculate the images of the list by the oblique projection onto the <*plane*> parallel to the <*vector*>. The <*plane*> is a list in the form: [3Dpoint, 3D normal vector].

4.12 rot3d

- **rot3d(<3Dpoint list>, <line>, <alpha>)**.
- Description: calculate the images of the list by the rotation with axis <*line*> and angle <*alpha*>. The <*line*> is a list in the form: [3Dpoint, 3D direction vector], The direction vector is orienting the line.

4.13 shift3d

- **shift3d(<3Dpoint list>, <3Dvector>)**.
- Description: calculate the list of the translated of the points from the <*3Dpoint list*> by the <*3Dvector*>.

4.14 sym3d

- **sym3d(<3Dpoint list>, <plane>)**.
- Description: calculate the list of the orthogonal symetric of the points of the <*3Dpoint list*> with respecto to the <*plane*>. The <*plane*> is a list in the form: [3Dpoint, 3D normal vector].

4.15 sym3dO

- **sym3dO(<3Dpoint list>, <plane>, <3Dvector>)**.
- Description: calculate and returns the list of the images of the <*3Dpoint list*> by the oblique symmetry with respect to the <*plane*> parallel to the <*3Dvector*>. The <*plane*> is a list in the form: [3Dpoint, 3D normal vector].

5) 3D transformation matrix

A 3D matrix is alist in the form **[vector3D1, vector3D2, vector3D3, vector3D4]**. That list represent the analytic expression of a space affine map. This is a three vectors list: **vector3D1** that is the translation vector, **vector3D2** is the first column vector of the matrix of the linear part in the canonical base, **vector3D3** is the second column vector of the matrix of the linear part, and **vector3D4** that is the third column vector of the matrix of the linear part.

If f is a space affine map, then its linear part is *Lf=f-f(Origin)*, the translation vector is *f(Origin)*, and its matrix is: *[f(Origin), Lf(vecI), Lf(vecJ), Lf(vecK)]*.

For example, the matrix of the identity is: *[M(0,0,0), M(1,0,0), M(0,1,0), M(0,0,1)]* or *[Origin, vecI, vecJ, vecK]* (this is the default matrix).

See also the commands *ComposeMatrix3D* (p. 112), *GetMatrix3D* (p. 115), *SetMatrix3D* (p. 119) and *IdMatrix3D* (p. 115).

5.1 invmatrix3d

- **invmatrix3d(<[f(0), Lf(vecI), Lf(vecJ), Lf(vecK)]>)**
- Description: returns the inverse of the matrix <*[f(0), Lf(vecI), Lf(vecJ), Lf(vecK)]*>, ie the matrix:

$$[f^{-1}(0), Lf^{-1}(vecI), Lf^{-1}(vecJ), Lf^{-1}(vecK)]$$

if it exists.

5.2 matrix3d

- **matrix3d(<affine function> [, variable]).**
- Description: returns the matrix of the <*affine function*>.By default, the <*variable*> is the letter *M* (represents a 3Dpoint). That matrix is in the form :[f(0), Lf(vecI), Lf(vecJ), L(vecK)], where f is the affine map and Lf its linear part, (vecI, vecJ, vecK) is its canonical base.
- Example(s): matrix3d(sym3d(M, [Origin,vecK])) returns [0,0,1,0,i,0,0,-1], representing the orthogonal symmetry with respect to the xOy plane.

5.3 mulmatrix3d

- **mulmatrix3d(<3Dmatrix of f>, <3Dmatrix of g>)**
- Description: returns the matrix of the composition: fog, with f and g two space affine maps defined by its marix, in the form [f(0), Lf(vecI), Lf(vecJ), Lf(vecK)] and Lf is the linear part.

6) Macros for the 3D window

6.1 drawWin3d

- **drawWin3d(<mode>)**
- Description: cette macro dessine la fenêtre 3D courante dans le <*mode*> voulu avec la macro *DrawPoly* (p. 145).

6.2 rectangle3d

- **rectangle3d(<3Dpoint list>)**
- Description: that macro determines the smallest cuboid containing the *3Dpoint list*, that macro returns the great diagonal of that box: *[M(Xinf, Yinf, Zinf), M(Xsup, Ysup, Zsup)]*

6.3 RestoreTphi

- **RestoreTphi()**
- Description: that maco restore th view angle values *theta* and *phi* from the stack (voir *SaveTphi* (p. 126)).

6.4 RestoreWin3d

- **RestoreWin3d()**
- Description: that macro restore the 3D window and the 3D matrix from the stack (see *SaveWin3d* (p. 126)).

6.5 SaveTphi

- **SaveTphi()**
- Description: this macro saves in a stack the values of the viewing angles : *theta* and *phi* (see also *RestoreTphi* (p. 126)).

6.6 SaveWin3d

- **SaveWin3d()**
- Description: this macro saves the current 3D window and the current 3D matrix in a stack (see also *RestoreWin3d* (p. 126)).

6.7 transformbox3d

- **transformbox3d(<[M(xinf, yinf, zinf), M(xsup, ysup, zsup)]> [, ortho])**

- Description: this macro computes the matrix transforming the box with great diagonal *<[M(xinf, yinf, zinf), M(xsup, ysup, zsup)]>* in the box with great diagonal $[M(-3,-3,-3), M(3,3,3)]$. If the optional parameter *<ortho>* is 1 (0 by default), then the coordinate system will be orthonormal, That matrix is **composed with the current 3D matrix**, the current 3D window is modified.

6.8 view3D

- **view3D(<xmin>, <xmax>, <ymin>, <ymax>, <zmin>, <zmax>)** or **view3D(<[M(xinf, yinf, zinf), M(xsup, ysup, zsup)]>)**

- Description: defines the 3D graphical window, ie the values of the variables: *Xinf*, *Xsup*, *Yinf*, *Ysup*, *Zinf* and *Zsup*.

7) Screen axes and 3D

The screen is the projection plane, passing through the origin of the space coordinate system. The unit vector normal to that plane and oriented towards the camera is the vector we get from the macro *n()* (p. 121).

7.1 ScreenX

- **ScreenX()**

- Description: that macro returns the space coordinates of the unit vector of the Ox axis of the screen.

7.2 ScreenY

- **ScreenY()**

- Description: that macro returns the space coordinates of the unit vector of the Oy axis of the screen.

7.3 ScreenPos

- **ScreenPos(<affix> [, distance])**

- Description: that macro returns the space coordinates of the point projected on the given *<affix>* at the given *<distance>* on the axis normal to the screen (or the axis oriented towrds the camera in central projection), that *<distance>* is optional and is 500 by default.

7.4 ScreenCenter

- **ScreenCenter()**

- Description: That macro returns the space coordinates of the center of the screen.

8) Clipping macros for 3D

8.1 Clip3D

- **Clip3D(<facets list>, <convex polyhedron> [, outside(0/1)])**

- Description: this macro returns the *<facets list>* clipped with the *<convex polyhedron>*. if the optional parameter *outside* is 0 (default value) this is the inside part of the polyhedron that is returned, if not, this is the outside part.

```
\begin{texgraph}[name=Clip3D, file]
Cmd  Fenetre(-5+5*i,5-5*i,0.5+0.5*i);
     Marges(0,0,0,0);
     viewDir(1,83);
Graph objet1 = [ size(7.5),
    background(full, lightgray),
    C1:=Cylindre(M(-4,0,0),8*vecI,2,25),
    C2:= Cylindre(M(0,0,-4),8*vecK,3,25),
    stock:= Clip3D(C2,C1,1),
    draw("SmoothFacet", stock,
                [color:=darkseagreen,
                 contrast:=0.5])
];
\end{texgraph}
```

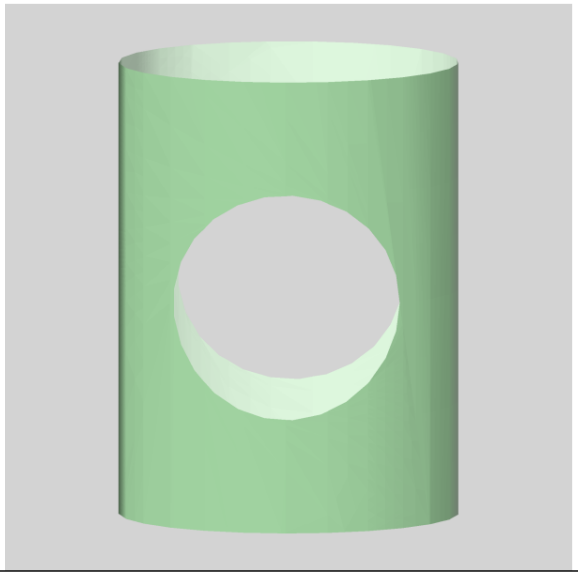


Figure 10: *Clip3D*

8.2 clipCurve

- **clipCurve(<3Dpoint list> [, 3D window])**
- Description: that macro returns the *<3Dpoint list>* clipped with the *<3D window>*, if that parameter is missing, then this is the current 3D window that is used. The *<3D window>* is given by its main diagonal: *[M(xinf, yinf, zinf), M(xsup, ysup, zsup)]*.

8.3 clipPoly

- **clipPoly(<facet list> [, 3D window])**
- Description: that macro returns the *<facet list>* clipped by the *<3D window>*, if that parameter is missing, this is the current 3D window that is taken in count. The *<3D window>* is given by its main diagonal: *[M(xinf, yinf, zinf), M(xsup, ysup, zsup)]*.

```
\begin{texgraph}[name=clipPoly, file]
Graph image = [
 ModelView(central), Marges(0,0,0,0), size(7.5),
 view3D(-3,3,-3,3,-3,3),
 S:=clipPoly( Sphere( M(1,0,1),3,30,15) ),
 HideWidth:=8, drawWin3d(0),
 DrawFacet(S, [color:=steelblue]),
 drawWin3d(1)
];
\end{texgraph}
```

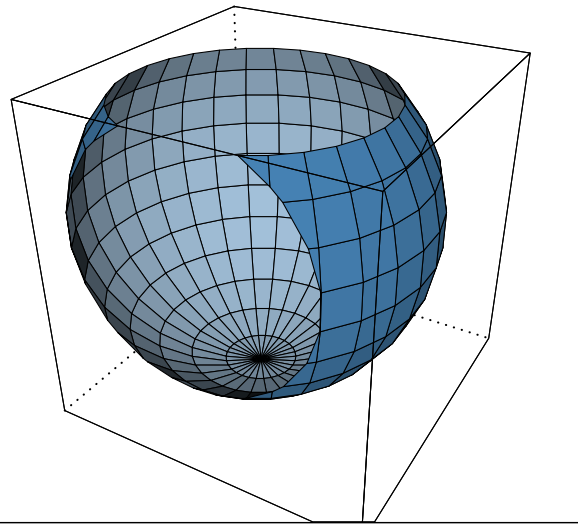


Figure 11: *clipPoly*

9) 3D objects construction macros

9.1 AretesNum (edges number)

- **AretesNum(<polyhedron>, <number list>)**
- Description: that macro returns the edges of the <*polyhedron*> whose numbers are in the <*number list*>. The edges are numbered in the order of appearance.

9.2 Chanfrein (chamfer)

- **Chanfrein(<convec polyhedron>, <thickness> [, blunting(0/1)])**
- Description: returns the chamfered <*convex polyhedron*>, for each edge, the solid is cut by a plane parallel to the bissector plane outside the two adjacent faces located at the distance equal to <*thickness*> towards the inside of the solid. The optional parameter <*blunting*> shows if the vertices have to be blunted or not (1 by default).

```
\begin{texgraph}[name=Chanfrein, file]
Graph image = [
 view(-3.5,4.5,-3,3), Marges(0,0,0,0), size(7.5),
 P:=Parallelep(Origin, 3*vecI,3*vecJ,3*vecK),
 P1:=Chanfrein(P,0.25,0), P2:=Chanfrein(P,0.25,1),
 DrawFacet( shift3d(P1,-2*vecJ), [color:=steelblue],
        shift3d(P2, 2*vecJ), [color:=steelblue]),
 Label(-2-1.5*i, "no blunting", 1.8-2.6*i, "blunting")
];
\end{texgraph}
```

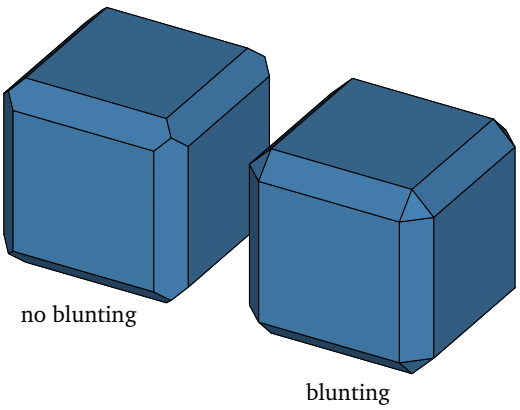


Figure 12: *Chanfrein (Chamfer)*

9.3 Cone

- **Cone(<3Dpoint>, <3Dvector>, <radius> [, nb faces, hollow])**
- Description: that macro returns a polyhedron representing the cone built from a <*3Dpoint*> that is the tip, a <*3Dvector*> showing the orientation axis and the cone height, a <*radius*> of the circular face and the <*nb faces*> number, that number is 35 by default. The <*hollow*> parameter is 0 or 1 (1 by default) shows if the cone has to be hollow or not, if not the circular face is added to the facets, this is the first in the list.

9.4 curve2Cone

- **curve2Cone(<f(t)>, <tmin>, <tmax>, <tip>, [, ratio, base])**
- Description: that macro returns in the form of facets, the cone starting with the <*tip*> and based on the left curve parametrized by $f(t) = [x(t) + i * y(t), z(t)]$ or $f(t) = M(x(t), y(t), z(t))$. The parameter <*ratio*> (zero by default) allow to build the other part of the cone using homothety, the last parameter , <*base*>, is a variable that is containing the output list of the points of the edge (or edges) of the cone.

```
\begin{texgraph}[name=curve2Cone, file]
Graph image = [
Marges(0,0,0,0), size(7.5), NbPoints:=100,
C:=curve2Cone( [cos(2*t)*exp(i*t)*5,-3],
          -pi,pi,[0,0],-1),
FillOpacity:=0.7,
DrawFlatFacet(C,[color:=darkseagreen])
];
\end{texgraph}
```

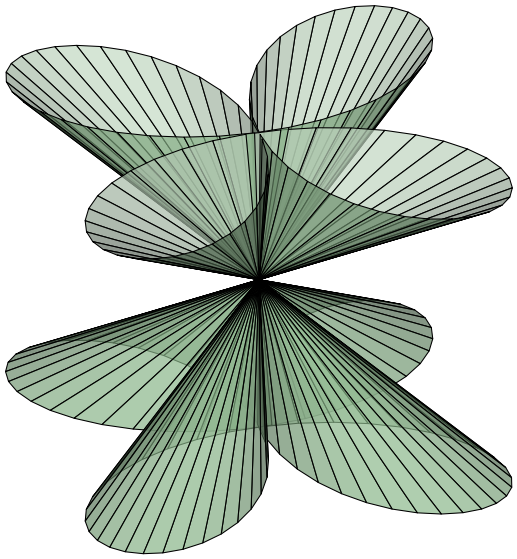


Figure 13: *curve2Cone*

9.5 curve2Cylinder

- **curve2Cylinder(<f(t)>, <tmin>, <tmax>, <3Dvector axis>, [, base])**
- Description: that macro returns in the form of facets, the cylinder based on the left curve parametrized by $f(t) = [x(t)+i*y(t),z(t)]$ or $f(t) = M(x(t),y(t),z(t))$. The parameter <*3Dvector axis*> shows how the base has to be translated to finish the cylinder. The last parameter <*base*>, is a variable that will contain the point list of the edge (or edges) of the cylinder.

```
\begin{texgraph}[name=curve2Cylinder, file]
Graph image = [
Marges(0,0,0,0), size(7.5),
C:=curve2Cylinder([4*cos(t)+2*i*sin(t),-3],
          -pi,pi,[i,6]),
DrawFlatFacet(C,[color:=violet])
];
\end{texgraph}
```

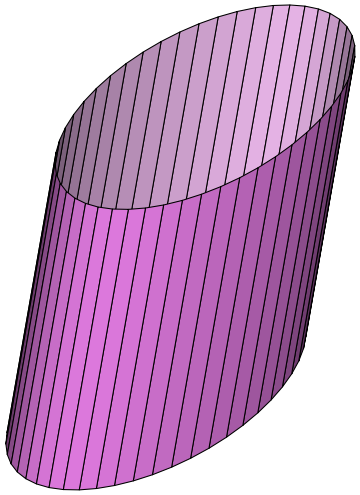


Figure 14: *Example with curve2Cylinder*

9.6 curveTube

- **curveTube(<f(t)>, <radius>, <tmin>, <tmax> [, nb points, nb faces, hollow (0/1)])**
- Description: that maco returns in the form of facets, a tube centered on the left curve parametrized by $f(t) = [x(t)+i*y(t),z(t)]$ or $f(t) = M(x(t),y(t),z(t))$, with the given <*radius*>. The parameter <*nb points*> is by default equal to the global variable *NbPoints*. The parameter <*nb faces*> is 4 by default, and the parameter <*hollow*> is 1 by default.

```
\begin{texgraph}[name=curveTube, file]
Graph image = [
Marges(0,0,0,0), size(7.5),
S:=curveTube([i*t,2.5*sin(t)],0.75,-pi,pi,50,12),
DrawFlatFacet(S,[color:=darkseagreen])
];
\end{texgraph}
```

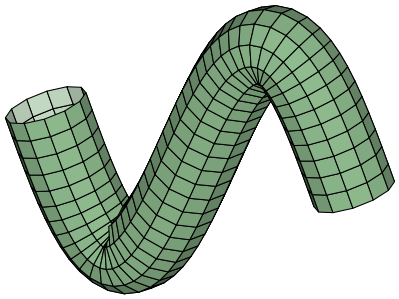


Figure 15: *curveTube*

9.7 Cvx3d

- **Cvx3d(<3Dpoint list>)**
- Description: returns the convex envelope of the <*list*> in the form of a facet list. The <*list*> must not contain the constant *jump*.

9.8 Cylindre

- **Cylindre(<3Dpoint>, <3Dvector>, <radius> [, nb faces, hollow])**
- Description: that macro returns a polyhedron representing the cylinder built from a <*3Dpoint*> that is the center of one of the two circular faces, a <*3Dvector*> of the axis showing the orientation and the height of the cylinder, a <*radius*> and the number :<*nb faces*>, that last is 35 by default. The parameter <*hollow*> is 0 or 1 (1 by default) shows if the cylinder is hollow or not, if not, the two circular faces are added to the facet list, those are the two first in that list.

9.9 FacesNum

- **FacesNum(<polyhedon>, <number list>)**
- Description: that macro returns the <*polyhedron*> faces whose numbers are in the <*number list*>. The faces are numbered in the order of appearance.

9.10 getdroite (3D straight line)

- **getdroite(<[3Dpoint,3Dvector]> [, scale])**
- Description: that macro returns a 3D segment corresponding to the line <*[3Dpoint,3Dvector]*> clipped by the current 3D window. The line is defined in the form of one of its points and a direction vector. The optional parameter <*scale*>, is 1 by default, is used to scale up or down the size of that segment (with respect to its middle).

9.11 getplan

- **getplan(<[3Dpoint,3Dvector]> [, scale])**
- Description: that macro returns a facet corresponding to the plane <*[3Dpoint,3Dvector]*> clipped by the current 3D window. The plane is defined in the form of one of its points and a normal vector. The optional parameter <*scale*>, is 1 by default, is used to scale up or down the size of that facet.

9.12 getplanEqn

- **getplanEqn(<[a,b,c,d]> [, scale])**
- Description: that macro returns a facet correponding to the plane <*[a,b,c,d]*> clipped by the current 3D window. The plane is defined in the form of a cartesian equation $ax + by + cz = d$. The optional parameter <*scale*>, is 1 by default, is used to scale up or down the size of that facet.

9.13 grille3d (3D grid)

- **grille3d(<x or y or z>, <value> [, <step>])**
- Description: that macro returns the plane <*x or y or z*> = <*value*> in the form of a grid (segments list). By default the <*step*> is 1, but it can be in the form `step1+i*step2` if you want different steps on the two sides of the grid; if `step2` is zero, we consider it is equal to `step1`.

```
\begin{texgraph}[name=grille3d, file]
Graph image = [
view(-8,8,-8,8), Marges(0,0,0,0), size(7.5),
ModelView(central), DistCam(30),
Color:=darkgray,
Ligne3D([grille3d(x,-5,1+2*i), grille3d(y,-5,1+2*i),
      grille3d(z,-5)],0),
Color:=black,
S:=curveTube([3*exp(i*t), t/3], 0.5,
        -3*pi, 3*pi, 100, 12),
Color:=black, Width:=1,
DrawFlatFacet(S,
      [color:=steelblue, contrast:=0.5])
];
\end{texgraph}
```

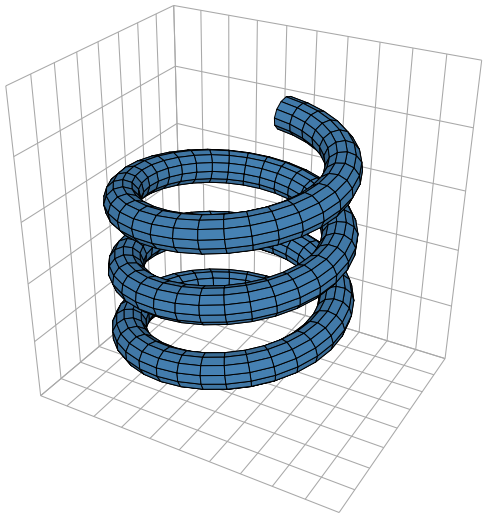


Figure 16: *grille3d (3D grid)*

9.14 HollowFacet

- **HollowFacet(<polyhedron> [, thickness+i*(mode 0/1), inside])**
- Description: that macro hollows each facet of the <*polyhedron*> leaving a <*thickness*> at the edge (0.25 by default), if <*mode*> is zero (default value) the cut is done parallel to the edge, if the <*mode*> is 1, the cut is done by a polyhedron whose vertices are taken on each edge of a given facet (see example below) . The removed pieces are put in the variable called <*inside*> if this one is present.

```
\begin{texgraph}[name=HollowFacetbis, file]
Graph image = [
view(-5,5,-2.5,2.5), Marges(0,0,0,0), size(7.5),
F:=for z in carre(3-3*i,3+3*i,1) do z,0 od,
theta:=-pi/2,phi:=0, ep:=2,SaveWin(),
view(-5,0,-2.5,2.5),ChangeWinTo([-5-5*i,5+5*i]),
DrawFacet(HollowFacet(F,ep), [color:=lightblue]),
Arrows:=2,Ligne3D([-3+3.25*i,0,(-3+ep/2)+3.25*i,0],0),
Arrows:=0,Label(-3+ep/4+4*i,"ep/2"),
RestoreWin(),SaveWin(),
view(0,5,-2.5,2.5), ChangeWinTo([-5-5*i,5+5*i]),
DrawFacet(HollowFacet(F,ep+i), [color:=lightblue]),
Arrows:=2, Ligne3D([3+3.25*i,0,(3-ep)+3.25*i,0],0),
Arrows:=0,Label(3-ep/2+4*i,"ep"),
RestoreWin(),
Label(-2.5-2*i,"mode=0"), Label(2.5-2*i,"mode=1")
];
\end{texgraph}
```

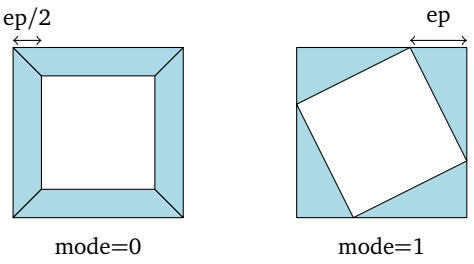


Figure 17: *(HollowFacet) mode values*

```
\begin{texgraph}[name=HollowFacet, file]
Graph image = [
ModelView(central), Marges(0,0,0,0), size(7.5),
background(full,darkblue),
P:=Parallelep(M(-2.5,-2.5,-2.5), 5*vecI,
        5*vecJ, 5*vecK), nb:=15,
Y:=[1,1,0], R:=[1,0,0],
Build3D(
  for k from 1 to nb do
   P:=HollowFacet(P, 0.25+i, aux),
   C:=((nb-k+1)*R+(k-1)*Y)/nb,
   bdFacet(P, [color:=RgbL(C)]),
   P:=aux, sep3D
  od),
Display3D()
];
\end{texgraph}
```

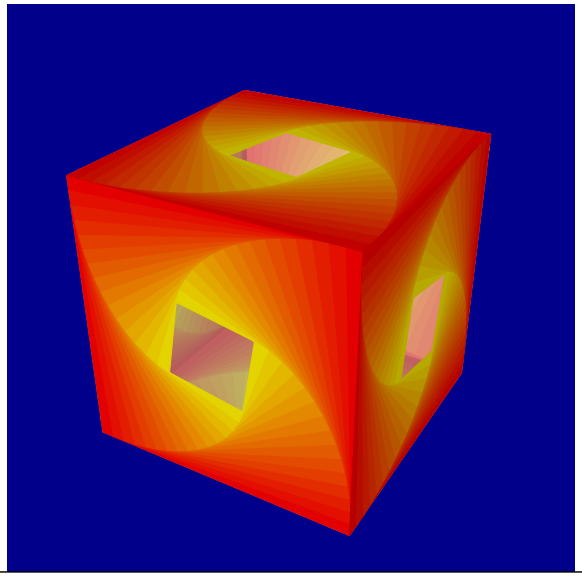


Figure 18: *HollowFacet: example*

9.15 Intersection

- **Intersection(<plane>, <polyhedron>) [, facet])**
- Description: the plane must be in the form: $[S,u]$ (plane passing through the point S and normal to the vector u). The macro determine the intersection of the <*polyhedron*> with that <*plane*> and returns the result in the form of an *edges list* (that can be drawn using the macro *DrawAretes* (p. 112)). It is possible to get the intersection in the form of a <*facet*> by putting a variable as third parameter.

9.16 line2Cone

- **line2Cone(<3D line>, <tip>, [, closed(0/1), ratio, base])**
- Description: that macro returns in the form of facets, the cone starting from the <*tip*> and based on the <*3D line*>, it must not contain the constant *jump*. The parameter <*ratio*> (zero by default) is used to build the other part of the cone with an homothety, the last parameter, <*base*>, is a variable that will contain the output list of the points of the edge (or edges) of the cone. The argument <*claosed*> shows if the line has to be closed or not (0 by default).

9.17 line2Cylinder

- **line2Cylinder(<3D line>, <3D vector axis>, [, closed(0/1), base])**
- Description: this macro returns in the form of facets, the cylinder based on the <*3D line*> that must not contain the constant *jump*. The parameter <*3D vector axis*> shows how the base has to be translated to finish the cylinder. The last parameter, <*base*>, is a variable that will contain the list of the points of the edge (or the edges). The argument <*closed*> shows if the line has to be closed or not (0 by default).

9.18 lineTube

- **lineTube(<3D point list>, <radius>, <nb faces> [, closed, hollow])**
- Description: that macro returns in the form of facets, a tube centered on the <*3D point list*>, with the given <*radius*> and <*nb faces*>. The parameter <*closed*> is 0 or 1 and shows if the line has to be closed or not (0 by default). The parameter <*hollow*> is 0 or 1 and shows if the tube is hollow or has to be closed at the end (1 by default), That parameter is not taken in count if the line is closed.

```
\begin{texgraph}[name=lineTube, file]
Graph image = [
Marges(0,0,0,0), size(7.5),
L:=for z in polyreg(0, 3.5, 6)
  do [z,0] od,
S:=lineTube(L, 1, 12, 1),
DrawFlatFacet(S,[color:=steelblue,
          backculling:=1])
];
\end{texgraph}
```

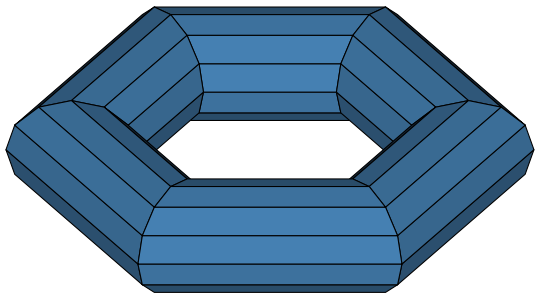


Figure 19: *lineTube*

9.19 Parallelep

- **Parallelep(<vertex>, <vector3D1>, <vector3D2>, <vector3D3>)**
- Description: that macro builds and returns the facets list of a parallelepiped from one <*vertex*> and three vectors, supposed to be in the standard orientation.

9.20 pqGoneReg3D

- **pqGoneReg3D(<axis, <tip>, <[p,q]>)**
- Description: that macro build an return the list of the points of a regulat <*p/q*>-gon of the space, from its <*axis*> and a <*tip*>. The axis is a straight line of the space ie: a list in the form [3Dpoint, 3Dvector], and the tip is a 3Dpoint.

9.21 Prisme

- **Prisme(<basis>, <3Dvector>)**
- Description: that macro returns the list of the facets of a prism from a <*basis*> and a <*3Dvector*> representing the translation vector of the basis to the opposite face. The basis is a list of 3D coplanar points, that list has to be in the standard orientation, given the fact that the plane is oriented by the translation vector.

9.22 Pyramide

- **Pyramide(<basis>, <tip>)**
- Description: that macro builds and returns the list of the facets of a pyramid built from its <*basis*> and <*tip*>. The basis is a list of coplanar 3D points, that list has to be in the standard orientation, knowing that the the plane is oriented by the tip.

9.23 rotCurve

- **rotCurve(<f(t)>, <Axis>, <tmin>, <tmax> [, angleMin, angleMax , tNbpoints, angleNbpoints])**
- Description: that macro returns in the form of facets, the surface obtained by turning around the <*Axis*>, the left curve parametrized by $f(t) = [x(t) + i * y(t), z(t)]$ or $f(t) = M(x(t), y(t), z(t))$. The argument <*Axis*> is a straight line of the space determined by a list [3D point,3D direction vecteur]. By default <*angleMin*>=$-\pi$, <*angleMax*>=π, <*tNbpoints*>=25, and <*angleNbpoints*>=25.

```
\begin{texgraph}[name=rotCurve, file]
Graph image = [
Marges(0,0.15,0,0), size(7.5),
C:=rotCurve(
  2*[1.5*i+cos(3*t)*cos(t)*i,-cos(3*t)*sin(t)],
  [0,0,0,1], -pi/2, pi/2, 0, pi, 50),
DrawFlatFacet(C,[color:=coral]),
tMin:=-pi/2,tMax:=pi/2, Color:=blue,
Width:=8,
Courbe3D(M(0,3+2*cos(3*t)*cos(t),
        2*cos(3*t)*sin(t)))
];
\end{texgraph}
```

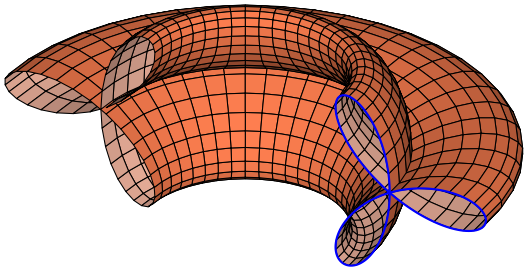


Figure 20: *rotCurve*

9.24 rotLine

- **rotLine(<3D line>, <Axis>, [, closed(0/1), angleMin, angleMax, angleNbpoints])**

- Description: that macro returns in the form of fecets, the surface obtained by turning around the <*Axis*>, the <*3D line*>, That must not contain the constant *jump*. The argument <*Axis*> is a straight line of the space determined by a list [3D point, 3D direction vector]. By default <*angleMin*>=$-\pi$, <*angleMax*>=π, and <*angleNbpoints*>=25. The argument <*closed*> shows if the <*3D line*> has to be closed or not (0 by default).

```
\begin{texgraph}[name=rotLine, file]
Graph image = [
view(-2,3.5,-3,3.5),Marges(0,0,0,0),
size(7.5), viewDir(55,60),
L:=[M(0,3,0), M(-0.5,2.5,0),
   M(-1,3,0), M(-1,2,0), M(0,2,0)],
P:=rotLine(L,[Origin, vecI],1,0,2*pi,35),
DrawFacet(P,[color:=coral])
];
\end{texgraph}
```

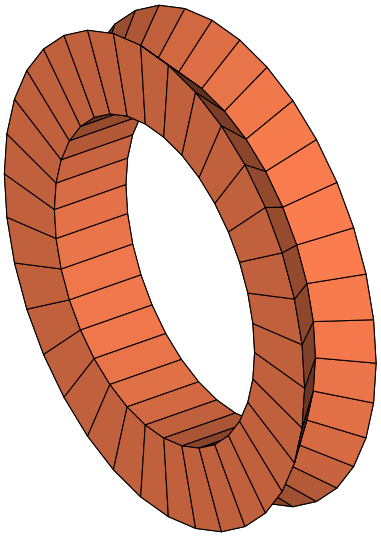


Figure 21: *rotLine*

9.25 Section

- **Section(<plane>, <polyhedron>)**

- Description: that macro cuts a <*polyhedron*> with a <*plane*>. The plane must be in the form: $[S,u]$, this is the plane passing through the point S and normal to the vector u. The macro determines the section of the polyhedron by that plane, and the part of the polyhedron that is in the half-space containing the vector u, is kept and returned by the macro in the form of a polyhedron (facets list).

- Example(s): section of a cube:

```
\begin{texgraph}[name=Section, file]
Graph image = [
view(-2,3,-2.5,3), Marges(0,0,0,0), size(7.5),
cube:=Parallelep(Origin,3*vecI,3*vecJ,3*vecK),
plan:=[M(3,0,0),-vecI-vecK/2],
S:=Section(plan, cube), FillStyle:=full,
FillColor:=firebrick, HideWidth:=8,
DrawPoly(S,4)
];
\end{texgraph}
```

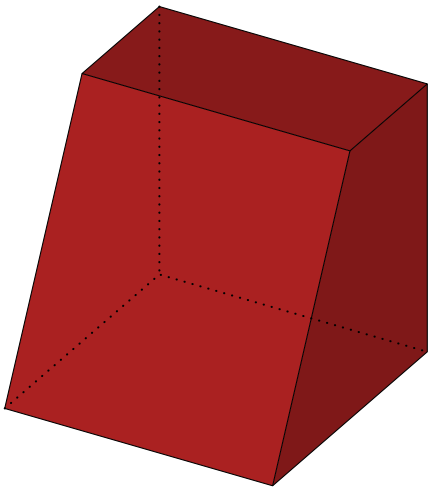


Figure 22: *Section*

9.26 Sphere

- **Sphere(, <radius> [, nb meridians, nb parallels))**
- Description: that macro returns a polyhedron representing the sphere built from its <*center*> and <*radius*>. the two other optional parameters show the faces number. By default the <*nb meridian*> is 40 and the <*nb parallels*> is 25.

9.27 Tetra

- **Tetra(<vertex>, <vector3D1>, <vector3D2>, <vector3D3>)**
- Description: that macro build and returns the list of the facets of a tetrahedron from a <*vertex*> and three vectors , supposed being positively oriented.

9.28 trianguler (triangulation)

- **trianguler(<list of convex facets>)**
- Description: that macro returns the <*list of convex facets*> after triangulation.

10) Line drawing macros for 3D

10.1 Arc3D

- **Arc3D(<B>, <A>, <C>, <radius>, <orientation>)**.
- Description: draw an arc with center <*A*>, radius <*radius*>, delimited by the line (*AB*) and the line (*AC*) while staying in the plane (*ABC*), positively oriented if <*orientation*> is strictly positive.

10.2 Axes3D

- **Axes3D(<Ox>, <Oy>, <Oz>, <stepx>, <stepy>, <stepz>)**.
- Description: draw the axes of the space coordinate system, the origin coordinate and the step of the ticks on the axes are given (0= no ticks).

10.3 AxeX3D

- **AxeX3D(<option1>, <option2>, ...).**
- Description: draw the Ox axis of the space coordinate system, that axis is oriented by the vector *vecI* passing through a point that is by default the origin. Options are:
  - axeOrigin := ⟨ point3D ⟩: defines a point of the axis. By default, that point is the origin: M(0,0,0).
  - xlimits := ⟨ [xinf,xsup] ⟩: defines the axis range. By default, this is the interval [Xinf, Xsup].
  - xgradlimits := ⟨ [x1,x2 ⟩: define the ticks range. By default this is the same as *xlimits*.
  - xstep := ⟨ nombre ⟩: define the ticks step: 1 by default. If the value is zero, then threre will be no ticks nor labels.
  - tickdir := ⟨ 3D vector ⟩: shows the ticks direction, by default that vector is *-vecK*.
  - tickpos := ⟨ 0..1 ⟩: shows the ticks position with respect to the axis, by default the value is 0.5 this means the axis passes through the middle of the ticks.
  - labels := ⟨ 0/1 ⟩: shows if the ticks labels will be displayed or not (1 by default).
  - originlabel := ⟨ 0/1 ⟩: shows if the label of the origin is displayed or not (0 by default).
  - nbdeci := ⟨ integer ⟩: the displayed decimal places number (2 by default). If the predefined variable *usecomma* is 1, the decimal point is replaced by a comma. If the variable *dollar* is 1, the graduations are framed with the character $.
  - xlabelstyle := ⟨ left/right/... ⟩: define the label style, the default value is equal to *LabelStyle*. The style does not apply to the legend.
  - xlabelsep := ⟨ distance en cm ⟩: define the distance between the end of the graduations and the labels (0.25 by default).
  - newxlegend(<"texte">): macro defining the legend for the Ox axis, by default the text is `"$x$"`. If the string is empty, there will be no legend.
  - xlegendsep := ⟨ distance en cm ⟩ define the distance between the end of the ticks and the legend or the end of the axis depending on the position. That distance is 0.5 by default and is added to *xlabelsep* if the legend is not at one end.
  - legendpos := ⟨ 0..1 ⟩: define the legend position, if there is one. With the value 0, the legend is “below” the end of the axis, with the value 1 the legend is “above” the end the axis, else it is along the axis. By default the value is 0.5 (middle of the axis).

10.4 AxeY3D

- **AxeY3D(<option1>, <option2>, ...).**
- Description: draw the Oy axis of the space coordinate system, that axis is oriented by the vector *vecJ* passing through a point that is by default the origin. Options are:
  - axeOrigin := ⟨ point3D ⟩: defines a point of the axis. By default, that point is the origin: M(0,0,0).
  - ylimits := ⟨ [yinf,ysup] ⟩: defines the axis range. By default, this is the interval [Yinf, Ysup].
  - ygradlimits := ⟨ [y1,y2 ⟩: define the ticks range. By default this is the same as *ylimits*.
  - xstep := ⟨ nombre ⟩: define the ticks step: 1 by default. If the value is zero, then threre will be no ticks nor labels.
  - tickdir := ⟨ 3D vector ⟩: shows the ticks direction, by default that vector is *-vecK*.
  - tickpos := ⟨ 0..1 ⟩: shows the ticks position with respect to the axis, by default the value is 0.5 this means the axis passes through the middle of the ticks.
  - labels := ⟨ 0/1 ⟩: shows if the ticks labels will be displayed or not (1 by default).
  - originlabel := ⟨ 0/1 ⟩: shows if the label of the origin is displayed or not (0 by default).

- nbdeci := ⟨ integer ⟩: the displayed decimal places number (2 by default). If the predefined variable *usecomma* is 1, the decimal point is replaced by a comma. If the variable *dollar* is 1, the graduations are framed with the character $.
- ylabelstyle := ⟨ left/right/... ⟩: define the label style, the default value is equal to *LabelStyle*. The style does not apply to the legend.
- ylabelsep := ⟨ distance en cm ⟩: define the distance between the end of the graduations and the labels (0.25 by default).
- newylegend(<"texte">): macro defining the legend for the Oy axis, by default the text is `"$y$"`. If the string is empty, there will be no legend.
- ylegendsep := ⟨ distance en cm ⟩ define the distance between the end of the ticks and the legend or the end of the axis depending on the position. That distance is 0.5 by default and is added to *ylabelsep* if the legend is not at one end.
- legendpos := ⟨ 0..1 ⟩: define the legend position, if there is one. With the value 0, the legend is "below" the end of the axis, with the value 1 the legend is "above" the end the axis, else it is along the axis. By default the value is 0.5 (middle of the axis).

10.5 AxeZ3D

- **AxeZ3D(<option1>, <option2>, ...).**
- Description: draw the Oz axis of the space coordinate system, that axis is oriented by the vector *vecK* passing through a point that is by default the origin. Options are:
  - axeOrigin := ⟨ point3D ⟩: defines a point of the axis. By default, that point is the origin: M(0,0,0).
  - zlimits := ⟨ [zinf,zsup] ⟩: defines the axis range. By default, this is the interval [Zinf, Zsup].
  - zgradlimits := ⟨ [z1,z2 ⟩: define the ticks range. By default this is the same as *zlimits*.
  - zstep := ⟨ number ⟩: define the ticks step: 1 by default. If the value is zero, then threre will be no ticks nor labels.
  - tickdir := ⟨ 3D vector ⟩: shows the ticks direction, by default that vector is *-vecJ*.
  - tickpos := ⟨ 0..1 ⟩: shows the ticks position with respect to the axis, by default the value is 0.5 this means the axis passes through the middle of the ticks.
  - labels := ⟨ 0/1 ⟩: shows if the ticks labels will be displayed or not (1 by default).
  - originlabel := ⟨ 0/1 ⟩: shows if the label of the origin is displayed or not (0 by default).
  - nbdeci := ⟨ integer ⟩: the displayed decimal places number (2 by default). If the predefined variable *usecomma* is 1, the decimal point is replaced by a comma. If the variable *dollar* is 1, the graduations are framed with the character $.
  - zlabelstyle := ⟨ left/right/... ⟩: define the label style, the default value is equal to *LabelStyle*. The style does not apply to the legend.
  - zlabelsep := ⟨ distance en cm ⟩: define the distance between the end of the graduations and the labels (0.25 by default).
  - newzlegend(<"texte">): macro defining the legend for the Oz axis, by default the text is `"$z$"`. If the string is empty, there will be no legend.
  - zlegendsep := ⟨ distance in cm ⟩ define the distance between the end of the ticks and the legend or the end of the axis depending on the position. That distance is 0.5 by default and is added to *zlabelsep* if the legend is not at one end.
  - legendpos := ⟨ 0..1 ⟩: define the legend position, if there is one. With the value 0, the legend is "below" the end of the axis, with the value 1 the legend is "above" the end the axis, else it is along the axis. By default the value is 0.5 (middle of the axis).

```
\begin{texgraph}[name=AxeZ3D, file]
Graph image = [
view(-6.5,6.5,-3,5.5),Marges(0,0,0,0),size(7.5),
view3D(-3,3,-3,3,-3,3),ModelView(central),
Width:=8,Color:=blue, FillStyle:=full,
FillColor:=lightcyan,Cercle3D(Origin,3,vecK),
Arrows:=1,LabelSize:=scriptsize,
Width:=4,Color:=black,
AxeX3D(axeOrigin:=M(-3,-3,0),tickdir:=-vecJ,
    xlabelstyle:=right,tickpos:=0,xlimits:=[-3,3.5],
    legendpos:=1,xlabelsep:=0.15),
AxeY3D(axeOrigin:=M(-3,-3,0), tickdir:=-vecI,
     tickpos:=0, ylimits:=[-3,3.5]),
AxeZ3D(axeOrigin:=M(-3,-3,0),zlimits:=[0,3.5],
     tickdir:=M(1,-1,0),zgradlimits:=[1,3]),
LineStyle:=dashed, Arrows:=0,
Ligne3D([M(0,-3,0),Origin,M(-3,0,0)],0),
LabelDot3D(Origin, "$O$","E",1)
];
\end{texgraph}
```

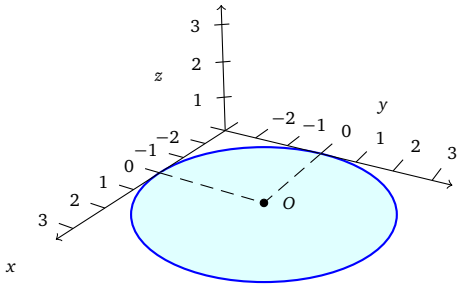


Figure 23: *Axes examples*

10.6 BoxAxes3D

- **BoxAxes3D(<option1>, <option2>, ...).**
- Description: draw the three axes Ox, Oy and Oz of the space coordinate system on three of the edges of the box corresponding to the current 3D window. Options are:
  - labels := ⟨ 0/1 ⟩: shows if the graduation labels have to be displayed or not (1 by default).
  - nbdeci := ⟨ entier ⟩: number of displayed decimal places (2 by default). If the predefined variable *usecomma* is 1, the decimal point is replaced by a comma. If the variable *dollar* is 1, the graduations are framed by the character $.
  - drawbox := ⟨ 0/1 ⟩: shows if all the edges of the box have to be drawn (0 by default).
  - grid := ⟨ 0/1 ⟩: shows if a grid has to be drawn (0 by default). If that option is 1, then the three grids at the back of the box are drawn. If the variable *FillStyle* is *full* then they are painted using the color *FillColor*.
  - gridcolor := ⟨ color ⟩: grid color if it is drawn (black by default).
  - gridwidth := ⟨ thickness ⟩: thickness of the grid drawings (2 by default).
  - xaxe := ⟨ 0/1 ⟩: shows if the Ox axis is displayed (1 by default).
  - xlimits := ⟨ [xinf,xsup] ⟩: define the axis range, by default this is the interval [Xinf, Xsup].
  - xgradlimits := ⟨ [x1,x2] ⟩: define the ticks range, by default the same as *xlimits*.
  - xstep := ⟨ nombre ⟩: define the ticks step: 1 by default. If the value is zero, then there will be no ticks nor labels.
  - xlabelstyle := ⟨ left/right/... ⟩: define the label style for the Ox axis, the default value is equal to *LabelStyle*. The style does not apply to the legend.
  - xlabelsep := ⟨ distance in cm ⟩: define the distance between the end of the ticks and te labels (0.25 by default).
  - newxlegend(<"text">): macro defining the legend for the Ox axis, by default the text is `"$x$"`. If the string is empty, then there will be no legend.
  - xlegendsep := ⟨ distance en cm ⟩: define the distance between the end of the ticks and the legend. That distance is 0.5 by default and is added to *xlabelsep*.
  - yaxe := ⟨ 0/1 ⟩: shows if the Oy axis has to be displayed (1 by default).
  - ylimits := ⟨ [yinf,ysup] ⟩: define the axis range, by default this is the interval [Yinf, Ysup].
  - ygradlimits := ⟨ [y1,y2] ⟩: define the ticks range, by default this is the same as *ylimits*.
  - ystep := ⟨ nombre ⟩: define the ticks step: 1 by default. If the value is zero, then there will be no ticks nor labels.

- ylabelstyle := ⟨ left/right/... ⟩: define the label style for the Oy axis, the default value is equal to *LabelStyle*. The style does not apply to the legend.
- ylabelsep := ⟨ distance in cm ⟩: define thedistance between the ticks end and the labels (0.25 by default).
- newylegend(<"text">): macro that is defining the legend for the Oy axis, by default the text is `"$y$"`. if the string is empty, then there will be no legend.
- ylegendsep := ⟨ distance in cm ⟩: define the distance between the end of the ticks and the legend. That distance is 0.5 by default and is added to *ylabelsep*.
- zaxe := ⟨ 0/1 ⟩: shows if the Oz axis is displayed or not (1 by default).
- zlimits := ⟨ [zinf,zsup] ⟩: define the axis range, by default this is the interval [Zinf, Zsup].
- zgradlimits := ⟨ [z1,z2] ⟩: define the ticks range, by default this is the same as *zlimits*.
- zstep := ⟨ number ⟩: define the ticks step: 1 by default. If the value is zero, then there will be no ticks nor labels.
- zlabelstyle := ⟨ left/right/... ⟩: define the label style for the Oz axis, the default value is equal to *LabelStyle*. The style does not apply to the legend.
- zlabelsep := ⟨ distance in cm ⟩: define the distance between the end of the ticks and the labels (0.25 by default).
- newzlegend(<"text">): macro defining the legend for the Oz axis, by default the text is `"$z$"`. If the string is empty, then there is no legend.
- zlegendsep := ⟨ distance in cm ⟩: define the distance between the ticks end and the legend. That distance is 0.5 by default and is added to *zlabelsep*.

- Example(s): See *here* (p. 12).

10.7 Cercle3D (circle)

- **Cercle3D(<3Dpoint>, <radius>, <3D normal vector>).**
- Description: draw a circle in the space, with center <*3Dpoint*>, the <*3D normal vector*> is normal to the circle plane and not zero.

10.8 Courbe3D

- **Courbe3D(<f(t)> [, divisions, discontinuities])**
- Description: draw a left curve parametrized by <*f(t)*> with $f(t) = [x(t) + iy(t), z(t]$ or $f((t) = M(x(t), y(t), z(t))$. The <*divisions*> number can be given by 2 between 2 consecutive points, and <*discontinuités*> (0 or 1) can be taken in count as in the function *Courbe* (p. 87).

10.9 Dcone

- **Dcone(<3Dpoint>, <3Dvector>, <radius>, <mode>)**
- Description: draw a cone from ist tip <*3Dpoint*>, a <*3Dvector*> of the axis showing the direction and height of the cone, and the <*radius*> of the circular face. The <*mode*> value can be:
  - 0: wire view, with hidden parts,
  - 1: outline visible only, with the style: FillStyle:=full to fill in the outline.
  - 2: outline visible (with the style: FillStyle:=full to fill in the outline), with superposition of the hidden parts.

  The drawing of the hidden part uses the variables *HideStyle*, *HideColor*, *HideWith*.

10.10 Dcylindre

- **Dcylindre(<3Dpoint>, <3Dvector>, <radius>, <mode>)**
- Description: draw a cylinder from a *<3Dpoint>* that is the center of one of its circular faces, a *<3Dvector>* of the axis showing the direction and height of the cylinder, and a radius r. The *<mode>* can be:
  - 0: wire view, with hidden parts,
  - 1: outline visible only, with the style: FillStyle:=full to fill in the outline.
  - 2: outline visible (with the style: FillStyle:=full to fill in the outline), with superposition of the hidden parts.

  The drawing of the hidden part uses the variables *HideStyle*, *HideColor*, *HideWith*.

10.11 DpqGoneReg3D

- **DpqGoneReg3D(<axis>, <vertex>, <[p,q]>)**
- Description: that macro draw a regular *<p/q>*-gon of the space, from its *<axis>* and a *<vertex>*. The axis is a straight line of the space, ie a list in the form: [3Dpoint, 3D vector], and the vertex is a 3Dpoint.

10.12 DrawAretes

- **DrawAretes(<edges list> , mode (0/1))**
- Description: draw an *<edges list>*. An edge is a list of two 3Dpoints ended by the constant *jump*, the imaginary part of it, is 0 for a hidden edge and 1 for a visible edge (see the command *Aretes* (p. 112)). the *<mode>* can be:
  - 0: all the edges are drawn,
  - 1: only the visible edges are drawn.

  drawing hidden edges uses the variables : *HideStyle*, *HideColor*, *HideWith*.

10.13 DrawDdroite

- **DrawDdroite(<line> [, length L])**
- Description: draw a half-line [A,A+u) of the space. It is in the form *[A=3Dpoint, u=3D direction vector]*. If there is no other argument, then the half-line is entirely drawn. If the parameter *<L>* is present, then this is the segment [A A+L*u/norm(u)] that is drawn.

10.14 DrawDroite

- **DrawDroite(<line> [, length L1, length L2])**
- Description: draw a line of the space, written in the form: *[3Dpoint, 3D direction vector]*. If there is no other argument, then the line is entirely drawn. If there are two other parameters: *<L1>* and *<L2>*, then if *A* is the point and *u* the direction vector, this is the segment joining A-L1*u/norm(u) to A+L2*u/norm(u) that is drawn.

10.15 DrawPlan

- **DrawPlan(<plane>, <3Dvector>, <length1>, <length2> [, type])**
- Description: permits to represent a plane of the space, the parameter *<plane>* is in the form *[3Dpoint, 3D normal vector]*, let be *A* the point and *u* the 3D normal vector, The following parameter is a vector of the plane (call it *v*), the macro computes the vectorial product $w = u \wedge v$ and determine the following parallelogram:

```
\begin{texgraph}[name=drawplan1, file]
Graph image = [
view(-5,5,-5,5),Marges(0,0,0,0), size(8), DotStyle:=cross,
A:=0, M4:=-4+3*i, M1:=-2-3*i, M2:=4-3*i, M3:=2+3*i,
LabelDot(A,"$A$","NE",1), Arrows:=1, Ligne([A,A+2*i],0),
Width:=4, Arrows:=0, angleD(A+i,A,A-1, 0.25),
LabelStyle:=scriptsize, LabelDot(A+i,"$\vec{u}$","O"),
Width:=8, Color:=red, Arrows:=1, Ligne([M1,M2],0),
Width:=4,Color:=black, LabelStyle:=left,
LabelDot((M1+M2)/2,"$\dfrac{L_1\cdot\vec{v}}{\|\vec{v}\|}$","S"),
Width:=8,Color:=red, Ligne([M2,M3],0),
Width:=4,Color:=black,LabelStyle:=top,
LabelDot((M2+M3)/2,"$\dfrac{L2\cdot\vec{w}}{\|\vec{w}\|}$","E"),
Arrows:=0,
LabelDot(M4,"$M4$","NO",1),
LabelDot(M1,"$M1$","SO",1),
LabelDot(M2,"$M2$","SE",1),
LabelDot(M3,"$M3$","NE",1),
Ligne([M3,M4,M1],0)
];
\end{texgraph}
```

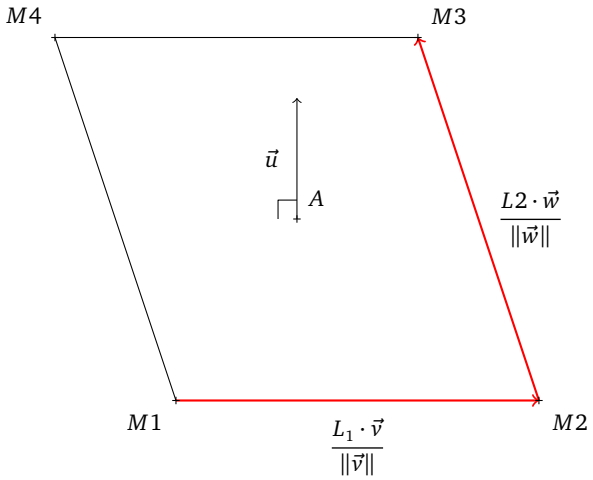


Figure 24: *The drawplan macro*

where L1 is the parameter <*length1*> and L2 the parameter <*length2*>. If the last parameter <*type*> is not present, then this is the parallelogram that is drawn. The possible type values are : -1, -2, -3, -4, 1, 2, 3, 4. that is giving (the point *A*, the vector *u* and the right angle have been added):

```
\begin{texgraph}[name=drawplan2, file]
Cmd                    [Fenetre(-6+5.5*i,6-5.5*i,0.625+0.625*i),
Marges(0,0,0,0), Border(0)];
     [OriginalCoord(1),IdMatrix()];
          [theta:=0.0872, phi:=1.1345, IdMatrix3D(), Mod-
elView(ortho)];
Var
  A = [-4.5*i,4];
  B = [-4.5*i,-1];
  C = [0,-5];
Mac
  plan = [ a:=%1, type:=%2, Arrows:=0,
    LabelDot(Proj3D(%1),"$A$","E",1,0.2),
    Width:=8,
    DrawPlan( [a,vecK], vecJ, 2, 2, type),
    angleD( Proj3D(a+vecK), Proj3D(a), Proj3D(a-vecJ), 0.15),
    Arrows:=1,
    Ligne( Proj3D( [a, a+vecK]),0),
    LabelDot( Proj3D([a+vecK]), "$\vec{u}$", "N",0)
    ];
Graph objet1 = [
    Width:=8, Marges(0,0,0,0), size(7.5),
      plan(A,1), plan( A+3*vecJ,2), plan( A+6*vecJ,3),plan(
A+9*vecJ,4),
      plan(B,-1), plan( B+3*vecJ,-2), plan( B+6*vecJ,-3),plan(
B+9*vecJ,-4),
    plan(C),
    Arrows:=0,LabelSize:=footnotesize,
    Label(-4.5+2.7564*i,"type=$1$"),
    Label(-1.2529+2.7564*i,"type=$2$"),
    Label(1.5+2.7564*i,"type=$3$"),
    Label(4.4824+2.7564*i,"type=$4$"),
    Label(-4.7471-2.0032*i,"type=$-1$"),
    Label(-1.5-2.0032*i,"type=$-2$"),
    Label(1.5-2.0032*i,"type=$-3$"),
    Label(4.2529-2.0032*i,"type=$-4$"),
    Label(-0.2471-5.2532*i,"no type")
    ];
\end{texgraph}
```

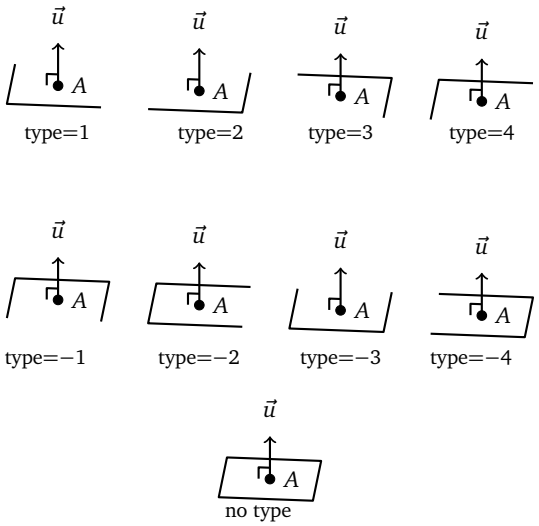


Figure 25: *Planes types*

10.16 Dsphere

labelDsphere

- **Dsphere(<3Dpoint>, <radius>, <mode>)**
- Description: draw a sphere from its center <*3Dpoint*> and <*radius*>. The <*mode*> can be:
  - 0: wire view, with hidden parts,
  - 1: outiline visible only. Using the style: FillStyle:=full will fill in the sphere.
  - 2: outline visible (using the style: FillStyle:=full is also possible), and the hidden part are superposed on top of it.

  Drawing the hidden part is using the variables *HideStyle*, *HideColor*, *HideWith*.

10.17 LabelDot3D

- **LabelDot3D(<point3D>, <"text">, <orientation> [, DrawDot, distance])**.
- Description: that macro displays a text near the point <*point3D*>. The three following parameter apply to the projection of the point on the screen plane. The orientation can be "N" for North, "NE" for north-east, "NO" for Northwest, "O" for West...etc, or a list in the form [length, direction] where direction is a complex. In that second case, the optional parameter <*distance*> is ignored. The point is also displayed if <*DrawDot*> is 1 (0 by default) and the <*distance*> (in cm) between the point and the text (0.25cm by default) can be redefined.

10.18 Ligne3D

- **Ligne3D(<3Dpoint list>, <closed>)**
- Description: draw a polyline in the space, the <*3Dpoint list*> can contain the constant *jump*. The parameter <*closed*> is 0 or 1 and shows if the curve has to be clased or not (1=closed).

10.19 markseg3d

- **markseg3d(<point3D1>, <point3D2>, <n>, <spacing>, <length> [, angle])**.
- Description: mark the segment defined by <*point3D1*> and <*point3D2*> with <*n*> small lines, the <*spacing*> is using the graphic unit, and the <*length*> is in cm. The optional parameter <*angle*> permits to define (in degrees) the angle of the marks with respect to the segment (45 degrees by default).

10.20 Point3D

- **Point3D(<3Dpoint list>)**
- Description: identical to the command *Point* (p. 92), but with space points..

11) Facet's drawing macros for the 3D

Those macro are in charge of displaying objects with facets based on a sort depending on the distance of the facet's centroid to the observer. That method does not always give good results, mainly in case of "big" facets.

11.1 Dparallelep

- **Dparallelep(<vertex>, <vector3D1>, <vector3D2>, <vector3D3> [, mode, contrast])**
- Description: that macro draws a parallelepiped starting from a <*vertex*> and three vectors, supposed to be positively oriented. That macro uses *DrawPoly* (p. 145) to draw in the given <*mode*> with the given <*contrast*>.

11.2 Dprisme

- **Dprisme(<basis>, <3Dvector> [, mode, contrast])**
- Description: that macro draw a prism starting from one <*basis*> and a <*3Dvector*> representing the translation vector from one basis to the opposite one. The basis is a list of coplanar 3Dpoints, that list has to be in the positive orientation, given that the plane is oriented by the translation vector. That macro uses *DrawPoly* (p. 145) to draw in the given <*mode*> with the given <*contrast*>.

11.3 Dpyramide

- **Dpyramide(<basis>, <tip> [, mode, contrast])**
- Description: that macro draws a pyramid from its <*basis*> and <*tip*>. The basis is a list of coplanar 3Dpoints, that list has to be positively oriented given that the plane is oriented by the tip. That macro uses *DrawPoly* (p. 145) to draw in the given <*mode*> with the given <*contrast*>.

11.4 DrawFacet

- **DrawFacet(facets1, [options1], facets2, [options2], ...)**
- Description: that macro sorts all the facets and displays them with respect to their options with the possibility to smooth (GOURAUD algorithm) or not, but **the eventual intersections are not handled**. Possible options are:
  - backculling := ⟨ 0/1 ⟩. Shows if the non visible facets have to be eliminated or not (0 by default).
  - color := ⟨ color ⟩. setting the color (white by default).
  - contrast := ⟨ positive number ⟩. The ordinary contrast is 1 (default value), a contrast set to zero means the color is solid. That number is used to vary the contrast between the facets of the same list.
  - smooth := ⟨ 0/1 ⟩. Shows if the GOURAUD algorithm (facets smoothing) has to be used or not in the *pstricks* or *eps* exports (0 by default).
- The default options are not reinitialized between <*facets1*> and <*facets2*> (idem for the following), then, by default, options of <*facets2*> and <*facets1*> are the same. If the options are identical, <*facets1*> can be replaced by <*[facets1,facets2]*>, or by an empty list ([]) for <*options2*>.
- If there is not smoothing at all, the macro *DrawFlatFacet* (p. 145) is a bit more accurate. If there are many smoothings (or only smoothings) to do on a great number of facets, the screen rendering can take time and the command *draw("SmoothFacet",...)* (p. 146) is then preferable, because the smoothing is only done at the export not at the execution.

```
\begin{texgraph}[name=DrawFacet, file]
Graph image = [
Load("PolyedresII.mac"),
Marges(0,0,0,0),size(7.5),
background(full,beige),
DrawFacet(Sphere(M(0,0,3.5),1.5,30,15),
          [color:=steelblue,
           backculling:=1,smooth:=1],
       Cube([0,0,vecK],M(1,1,0)),
          [color:=orange,smooth:=0],
       Cube([0,0,vecK],M(2,2,-4)),
          [color:=gold]
          )
];
\end{texgraph}
```

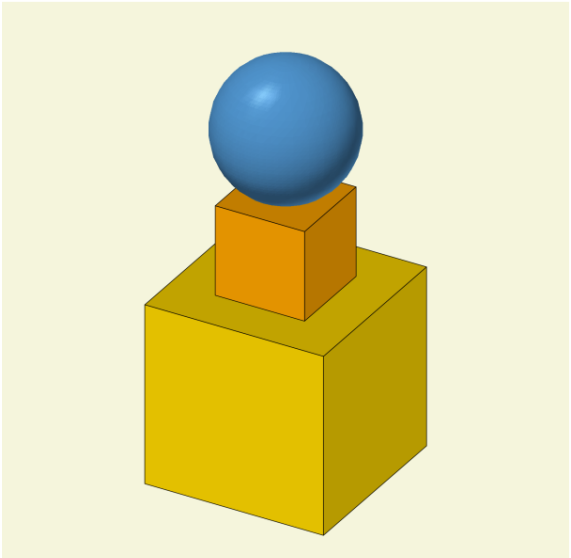


Figure 26: *DrawFacet*

11.5 DrawFlatFacet

- **DrawFlatFacet(facets1, [options1], facets2, [options2], ...)**
- Description: that macro sorts the set of all the facets and displays them according to their options, but **the eventual intersections are not handled and there is no GOURAUD smoothing**. Possible options are:
  - backculling := ⟨ 0/1 ⟩. Shows if the non visible facets have to be eliminated or not (0 par défaut).
  - color := ⟨ color ⟩. setting the color (white by default).
  - contrast := ⟨ positive number ⟩. The ordinary constrast is 1 (default value), a contrast set to zero means that the color is solid.

```
\begin{texgraph}[name=DrawFlatFacet, file]
Graph image = [
Marges(0,0,0,0), size(7.5),
theta:=75*deg, phi:=60*deg,
S:=Cylindre(M(0,0,-4), 8*vecK, 2, 25, 0),
C:= curveTube([3*exp(i*t),t/3],
      0.5, -2*pi, 2*pi, 75, 12,0),
DrawFlatFacet(S,[color:=steelblue,
            backculling:=1],
          C, [color:=crimson])
];
\end{texgraph}
```

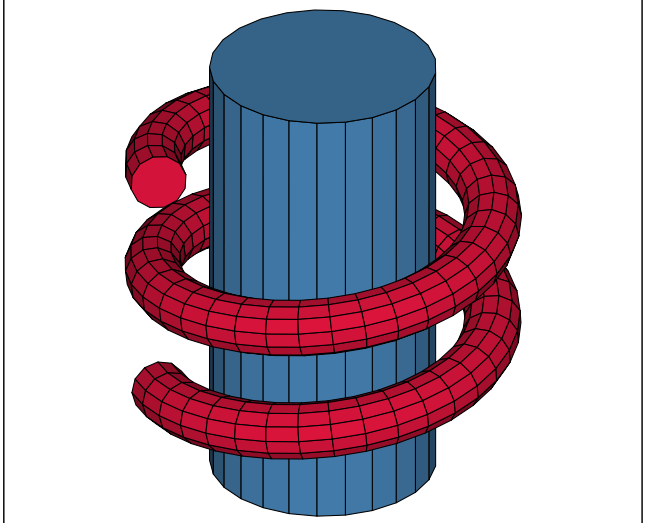


Figure 27: *DrawFlatFacet*

11.6 DrawPoly

- **DrawPoly(<convex polyhedron> [, mode, contrast]])**
- Description: permits to draw a <*convex polyhedron*> in the given <*mode*> (0 by default). Possible mode values are:
  - mode 0: the drawing is done edge by edge, hidden edges included (those are drawn using the style HideStyle), no filling,
  - mode 1: the drawing is done by visible faces, with filling or not according to the *FillStyle* attribute, all the facets are then with the same color (*FillColor*),
  - mode 2: the drawing is done like in the mode 1 (visibles faces), then the hidden edges are added,
  - mode=3: as in the mode 1 but the fill color is nuanced depending on the exposure of the facets and the <*contrast*> value,
  - mode=4: the drawing is done by visible faces but the fill color of the facets is nuanced depending on the facets exposure and the <*contrast*> value, then the hidden edges are added.
- The parameter <*contrast*> is a positive number (1 by default), it permits to modify or not the contrast of the color of the facets, the value 0 will give a solid color like the modes 1 and 2.
- The advantage of that macro is that the the edges are managed, that is not the case with the macro *DrawFacet* (p. 144).

11.7 DrawSmoothFacet

- **draw("SmoothFacet", facets1, [options1], facets2, [options2], ...)**
- Description: that macro sort the set of all the facets and displays them according to their options but **the eventual intersections are not handled**.with exports in *pstrick* or *eps*, and then *epsc* and *pdf* and (but not *pdfc*), the GOURAUD algorithm is used to fill the facets (after triangulation) that is giving a smoothing effect, the smoothin is not visible on screen. **With that macro the edges are not drawn**. Options are:
  - backculling := ⟨ 0/1 ⟩. Shows if the hidden facets have to be eliminated or not (0 par défaut).
  - color := ⟨ color ⟩. Setting the color (white by default).
  - contrast := ⟨ positive number ⟩. The ordinary contrast is 1 (default value), a contrast set to zero means the color is solid.
  - That macro uses a personalized export and then can be used in the form **draw("SmoothFacet",facets1, [options1], facets2, [options2], ...)**, with that form, the export will automatically launch the execution of the macro macro **ExportSmoothFacet()** that is defined in the file *scene3d.mac*. While in the form **DrawSmoothFacet(facets1, [options1], facets2, [options2], ...)** the export will be classic export, that is what can be seen on screen (facets without smoothing).

```
\begin{texgraph}[name=DrawSmoothFacet, file]
Graph image = [
Marges(0,0,0,0),size(7.5),
background(full,beige),
draw("SmoothFacet", Sphere(M(-3,0,0),3,25,15),
                [color:=steelblue,
                 backculling:=1],
                Sphere(M(3,0,0),3,25,15),
                [color:=orange])
];
\end{texgraph}
```

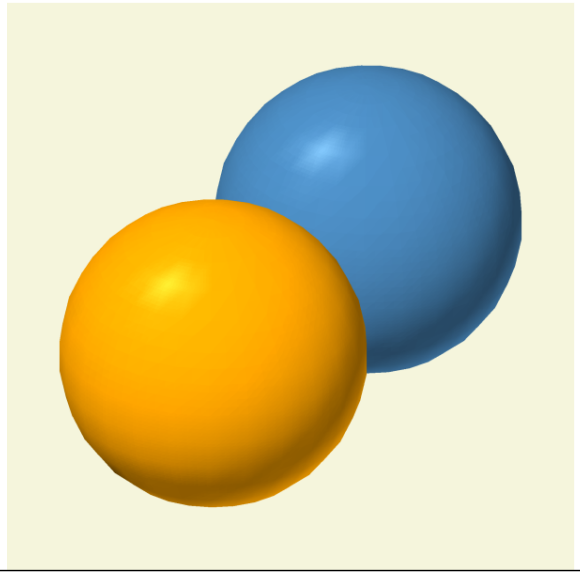


Figure 28: *Example with DrawSmoothFacet*

Warning: the above example illustrate the macro *DrawSmoothFacet* that permits to smooth the facets with the GOURAUD algorithm. But that algorithm is really known only by ghostscript. That is why pdf rendering takes sometimes long time (very long time) and not accurate for big images. In that case, it is preferable to get a high resolution jpeg (or an *eps* export if the document has to stay using ps format).

11.8 Dsurface

- **Dsurface(<f(u,v)> [, uMin+i*uMax, vMin+i*vMax, uNbLg+i*vNbLg, (smooth 0/1)+i*contrast])**
- Description: that macro draw a surface parametrized by <*f(u,v)*> where f is a function of two real variables u and v, with values in the space. The second parameter represent the interval of the variable u ($[-5,5]$ by default), The third parameter represents the interval of the variable v ($[-5;5]$ by default), the fourth parameter represents, in the form of a complex, the lines number for u and the lines number for v (25 lines by default). This is the macro *DrawFacet* (p. 144) that is doing the rendering with the color corresponding to the variable *FillColor* with the given <*contrast*> (1 by default) and a smoothing if <*smooth*> is 1 (0 by default).

11.9 Dtetraedre

- **Dtetraedre(<vertex>, <vector3D1>, <vector3D2>, <vector3D3> [, mode, contrast])**
- Description: that macro draws a tetrahedron starting from a <*vertex*> and three vectors, supposed to be positively oriented. That macro uses *DrawPoly* (p. 145) to draw in the gven <*mode*> with the given <*contrast*>.

Chapter XI

Build3D command: representation of a 3D scene

It is now possible to mix several 3D objects in a same 3D scene (the intersections are now handled) That scene is built using the BSP-trees algorithm in the form of a tree by the command **Build3D**, and the command **Display3D** is used to display the scene on screen.

Warning: This technique gives vectorial images that can rapidly become very heavy with scenes that are a bit complex (ie: with a great number of facets).

1) The two basic commands

1.1 Build3D

This command define the list of 3D elements that compose the scene. That command does not draw anything; as it can be seen in the example file *display3d.teg*, the different scene are built in macros, and one graphic element is enough, it contains the instruction *Display3D()* (p. 148). That command calculate the scene (more precisely: it builds a display tree), and display the scene. If the viewing angle is changed, only the command Display3D() is updated, and not the command Build3D().

The general syntax for Build3D is the following:

- **Build3D(<object1>, <object2>,...)**
- Description: that function removes the existig scene and build a new one with the given objects as arguments, it returns *Nil*. Each object can be a list of several 3D objects, then delimited with the constant: *sep3D*. You will find further the *building macros for Build3D* (p. 148), but here are presented the "atomic" objects. There are four kind of those objects, internally coded in the following manner:
  - **facets**: in that case, the object has to be in the form:
    [<±1+i*shade>, <color±i*opacity>, <facet list>]
    The value <*-1*> means the GOURAUD smoothing has to be used in the exports that can handle it. With the value <*1*> there is no smoothing. The <*shade*> is optional and is 0 by default. The <*opacity*> is optional and is 1 by default, else it must be a number between 0 and 1. If the opacity is multiplied by -i, it means, by convention, the front and the back of the face are not distinguished, while with +i the two sides do not have exactly the same color. The facets color is nuanced according to exposure. The parameter <*nuance*> can modify this. Its value must be greater or equal to -1:
    - nuance=-1: no shading. All the facets have the same color,
    - nuance=0: this is the default value,
    - the higher the value is increased, the more the contrast increases.
  - **lines**: in that case the object has to be in the form:
    [<2>, <color+i*opacity>, <thickness+i*LineStyle>, <3Dpoint list>]
  - **points**: in that case the object has to be in the form:
    [<3>, <color+i*opacity>, <width+i*linestyle, <3Dpoint list>]
  - **labels**: in that case, the object has to be in the form:
    [<3+i>, <color+i*number>, <labelsize+i*labelstyle>, <[pos,dir]>]

– **compiled labels**:
[<3-i>, <color+i*number>, <labelsize+i*labelstyle>, <[pos,dir]>]

- Some macros from the file *scene3d.mac* (loaded at startup) simplify the definition of elements in a 3D scene and can be used as arguments with the command *Build3D*. All these macros contain a list of options in their last argument. An option is declared like the following: *<name> := <value>*.

- Example(s): a cut sphere is drawn, a plane, a cylinder, then the axes with the hidden lines.

```
\begin{texgraph}[name=Build3D, file]
Graph image = [
 view(-5.5,5.5,-5.5,5.5),Marges(0,0,0,0),
 size(7.5),background(full,beige),
 z:=-2,
 Build3D(
  bdPlan([M(0,0,z), vecK],
   [color:=gold,border:=0,bordercolor:=black]),
  bdCylinder(M(-2,3,2), 7*M(2/3,-1,-2/3), 1,
   [color:=slategray,smooth:=1]),
  bdSphere(Origin, 3,
   [color:=darkseagreen, clip:=-1,
    clipwin:=[M(2,1,1),M(-1,-1,-1)],
    smooth:=1, backculling:=0]),
  bdCercle(M(0,0,z),sqrt(5),vecK,
   [color:=blue, width:=12]),
  bdAxes([0,0],
   [hidden:=1, arrows:=1,color:=firebrick])
  ),
 Display3D()
];
\end{texgraph}
```

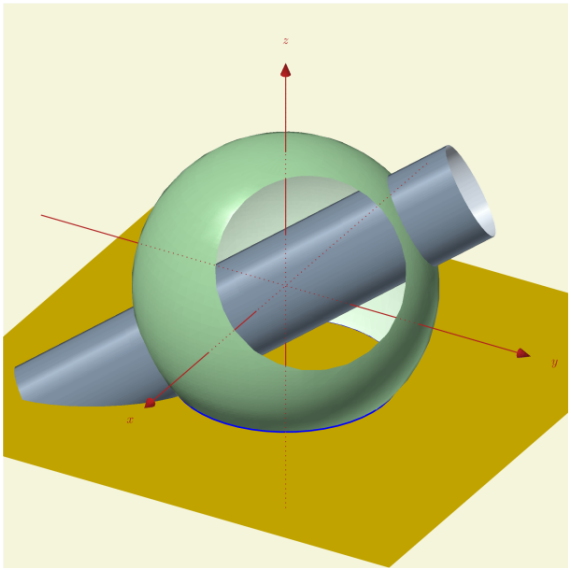


Figure 1: *Build3D*

1.2 Display3D

- **Display3D()**

- Description: that function draw on screen the scene created with *Build3D* (p. 147). That function is used without any argument.

2) Macros for Build3D()

2.1 globlal options

- hiddenLines := ⟨ 0/1 ⟩: that option is taken in count by the macro *bdLine* (p. 153). This is the default value of the option hidden. If its value is 1, the line is drawn a second time but on top of the scene, in the same color, with the style *HideStyle* and thickness *HideWidth* (or 0.8pt if its value is *Nil*). That superposition is not seen on visible parts of the line, but only on hidden parts.

- TeXifyLabels := ⟨ 0/1 ⟩: that option is taken in count by the macro *bdLabel* (p. 152). This is the default value of the option TeXify, it shows if the label is a math formula that has to be compiled by TEX, TeXgraph launch a pdflaTeX compilation in the background then call the tool *pstoedit* (`http://www.pstoedit.net/`) that translate the pdf file into flattened postscript that TeXgraph can then parse to get the formula in the form of paths. This assumes a TEX distribution is intalled with the program pstoedit. The compiled file is called *tex2FlatPs.tex*, and can be found in the directory `$HOME/.TeXgraph` of the linux user, and in `c:\tmp` under windows. There is also one copy in the installation directory of TeXgraph, by default that file is using the *fourier* font with 12pt, if the variable *dollar* is 1, the formula is inserted between two delimiters: `\[..\]`, if not, it is let unchanged, and it is composed with the size `\large`. By default, that option is 0.

2.2 bdArc

- **bdArc(<B>, <A>, <C>, <R>, <orientation>, [options])**
- Description: defines an arc in the space with radius <*R*>, starting from (AB) to (AC). The plane (BAC) is oriented by the basis $(\vec{AB},\vec{AC})$ and the <*orientation*> is 1 if it is positively oriented, -1 if not. bdArc options:
  - labelarc(<"text">). This is a macro for creating a label on the arc.
  - normal := ⟨ non zero 3Dvector ⟩. Vector that will be considered as the vector normal to the plane if the angle is flat (*Nil* by default).
  - radscale := ⟨ number ⟩. Number that, multiplyed by the arc radius will given the distance of the label to the arc center (1.25 by default).
- This is the macro bdCurve that is called to draw the arc, therefore *bdCurve* (p. 150)'s options can be used.

2.3 bdAngleD

- **bdAngleD(<B>, <A>, <C>, <length>, [options])**
- Description: create the "right angle" of the space defined by the two lines (AB) and (AC), with the 3Dpoints :*A*, *B* and *C*.
- This macro calls bdLine, then *bdLine* (p. 153)'s options can be used.
- Example(s):

```
\begin{texgraph}[name=bdAngleD, file]
Graph image = [
Marges(0,0,0,0), view(-3,3,-3,3),
view3D(-3,3,-3,3,-3,3), size(7.5),
background(full, gray),
B:=M(0,2,0), A:=M(0,0,0),C:=M(0,0,1.5),
Build3D(
 bdAngleD(B,A,C,1, [color:=firebrick,tube:=1]),
 bdDot([A,B,C], [dotstyle:=cube,
                 dotscale:=0.85,
                 color:=forestgreen]),
 bdArc(B,A,C,2,1,[color:=blue, width:=12,
                 arrows:=1,labelarc("$\pi/2$")]),
 bdAxes([0,0], [color:=gold,arrows:=1]),
 bdLabel(B,"$B$",[labelpos:=[0.5,-i]]),
 bdLabel(C,"$C$",[labelpos:=[0.5,-1]]),
    ),
Display3D()
];
\end{texgraph}
```

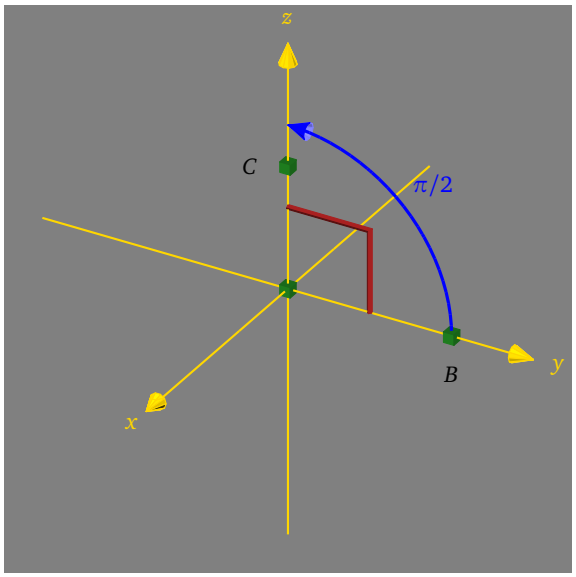


Figure 2: *bdAngleD*

2.4 bdAxes

- **bdAxes(<3Dpoint>, [options])**
- Description: defines the axes, <*3Dpoint*> is the intersection point of the three axes. bdAxes options:
  - labels := ⟨ 0/1 ⟩. Shows if the letters x, y and z are present at the end of the three axes (1 by default).
  - newxlegend(<"text">), newylegend(<"text">), newzlegend(<"text">): macros defining the legend on the axes, these are by default: x, y and z.
- That macro calls bdLine, that is why the options of *bdLine* (p. 153) can be used.

2.5 bdCercle

- **bdCercle(<3Dpoint>, <radius R>, <3D normal vector>, [options])**
- Description: defines the circle in the space with center <*3Dpoint*> and <*radius R*>, the circle plane is orthogonal to the <*3D normal vector*>.
- That macro calls bdCurve, that is why the options of *bdCurve* (p. 150) can be used.
- Example(s): *Villarceau* (p. 155) circles.

2.6 bdCone

- **bdCone(<3Dpoint>, <3Dvector>, <radius>, [options])**
- Description: defines the cone built from its tip <*3Dpoint*>, a <*3Dvector*> of the axis showing the orientation and height of the cone, and the <*radius*> of the circular face. bdCone options are those of *bdFacet* (p. 151), plus:
  - hollow := ⟨ 0/1 ⟩. Shows if the cone is hollow or not (1 by default).
  - nbfacet := ⟨ facets number ⟩. Defines the facets number (35 by default).
  - border := ⟨ 0/1 ⟩. Shows if the border has tobe drawn or not (0 by default).
  - bordercolor := ⟨ color ⟩. Shows the border color (same as *color* by default).

2.7 bdCurve

- **bdCurve(<f(t)>, [options])**
- Description: defines a curve in the space, parametrized by $f(t) = [x(t) + i * y(t), z(t)]$ or $f(t) = M(x(t), y(t), z(t))$, where $x(t)$, $y(t)$ and $z(t)$ are functions of one variable t. bdCurve options:
  - t := ⟨ [tmin, tmax] ⟩. Interval for the paraamеter t, [-5,5] by default.
  - nbdot := ⟨ positive integer ⟩. Defines the number of points, 25 by default.
- That macro calls bdLine, options of *bdLine* (p. 153) can then be used.

2.8 bdCylinder

- **bdCylinder(<3Dpoint>, <3Dvector>, <radius>, [options])**
- Description: defines the cylinder built from a <*3Dpoint*> that is the center of one of the two circular faces, a <*3Dvecteur*> of the axis showing the direction and height of the cylinder, and the <*radius*>. The options of bdCylinder are those of *bdFacet* (p. 151), plus:
  - hollow := ⟨ 0/1 ⟩. Shows if the cylinder is hollow or not (1 by default).
  - nbfacet := ⟨ facets number ⟩. Defines the facets number (35 by default).
  - border := ⟨ 0/1 ⟩. Shows if the border has to be drawn or not (0 by default).
  - bordercolor := ⟨ color ⟩. Shows the border color (by default identical to *color*).

2.9 bdDot

- **bdDot(<3Dpoint list>, [options])**
- Description: defines a list of points in the space. bdDot options:
  - color := ⟨ color ⟩. Defines the points color (black by default).
  - dir := ⟨ 3Dvector or [vector3D1,vector3D2] ⟩. If dotstyle=line , the "dir" option must contain a direction vector of the line to be drawn (in the space). If dotstyle=cross , the "dir" option must contain a list of two direction vector for the lines to be drawn (in the space). By default "dir" is *Nil*.
  - dotscale := ⟨ positive number ⟩. Defines a scale factor (1 by default).
  - dotstyle := ⟨ disc/cube/line/cross ⟩. Defines the points style (disc by default).
- If dostsyle=cube the macro bdFacet is called, in that case, the options of *bdFacet* (p. 151) can be used, if dotstyle=line or cross the macro bdLine is called, in that case, options of *bdLine* (p. 153) can be used.

2.10 bdDroite

- **bdDroite(<[3Dpoint, 3Dvector]>, [options])**
- Description: defines a straight line, represented with the list *<[3D point, 3D direction vector]>*. bdDroite options:
  - scale := ⟨ strictly positive number ⟩. The line is clipped by the current 3D window thus giving a segment, that can be scaled.
- That macro calls bdLine, in that case, options of *bdLine* (p. 153) can be used.

2.11 bdFacet

- **bdFacet(<facets list>, [options])**
- Description: defines a facets list. bdFacet options are:
  - backculling := ⟨ 0/1 ⟩. Shows if the non visible facets have to be eliminated or not (0 by default). A facet is not visible if its normal vector is not in the direction of the observer.
  - clip := ⟨ 0/1 ⟩. Shows if the facets have to be clipped by the window defined with the option clipwin if clip is 1, or by the plane defined with the option clipwin if clip is -1 (clip=0 by default).
  - clipwin := ⟨ [M(xinf,yinf,zinf), M(xsup,yup,zsup)] ⟩. Defines the 3D window for an eventual clipping if clip=1, the window is then given by its great diagonal: [M(xinf,yinf,zinf), M(xsup,yup,zsup)] (this is the current window by default). But if clip=-1 the option clipwin is interpreted as a plane: [3Dpoint, 3D normal vector].
  - triangular := ⟨ 0/1 ⟩. Shows if the facets are triangulated or not (0 by default).
  - addsep := ⟨ "x" or "y" or "z" ⟩. That options, if it is not equal to *Nil* (default value), determine the englobing box of each facet and add in the list one of the faces of that box (the face perpendicular to the Ox axis with x minimal if the option value is "x"), that new facet will remain invisible and will be used as separation wall, therefore the "real" facet won't be cut by a facet that is entirely behind the wall. That option is useless for convex objets.
  - color := ⟨ couleur ⟩. Defines the color (white by default).
  - contrast := ⟨ positive number ⟩. The ordinary contrast is 1 (default value), a contrast set to zero means the color is solid.
  - smooth := ⟨ 0/1 ⟩. Shows if the GOURAUD algorithm (facets smoothing) has to be used or not at the pstricks or eps exports(0 by default). Warning, pdf viewers are slow to display those type of images !
  - opacity := ⟨ number between 0 and 1 ⟩. Opacity value (1 by default), permits to introduce transparency if opacity is strictly less than 1.
  - matrix := ⟨ 3D matrix ⟩. Define a transformation matrix that will be applied on the facets (this is the identity by default). The transformation is done before an eventual clipping.

- twoside := ⟨ 0/1 ⟩. Show if the front-back of the facets have to be distinguished. If yes, the two sides won't be in the same color (1 by default).
- above := ⟨ positive number or zero ⟩. Places the facets above the scene, translated with the vector `above*500*\n` (0 by default).
- border := ⟨ 0/1 ⟩. Shows if the edges of the facets have to be drawn or not (0 by default).
- bordercolor := ⟨ color ⟩. Edges color if border=1 (black by default).
- hidden := ⟨ 0/1 ⟩. Shows that the hidden edges have to be drawn if border=1, if yes, then the variables *HideStyle* and *HideWidth* are used. By default, that option has the value of the general option *hiddenLines*.

2.12 bdLabel

- **bdLabel(<3Dpoint>, <"text">, [options])**
- Description: defines a label in the space, the *<3Dpoint>* is the anchor. The label is drawn on the projection plane and not really in the space, but its anchor is handled in the scene to determine the display order. Here are the bdLabel options:
  - TeXify := ⟨ 0/1 ⟩: shows if the label has to be compiled by TEX, see general option *TeXifyLabels* (p. 148).
  - scale := ⟨ number>0 ⟩. If the option *TeXify* is 1, the label size can be modified with this option.
  - color := ⟨ color ⟩. Sets the label color (black by default).
  - dotcolor := ⟨ color ⟩. Defines the anchor color if it has to be displayed (equal to the color option by default).
  - labelpos := ⟨ [distance cm, direction affix] ⟩. Shows the lable position with respect to the anchor **on the projection plane** (*Nil* by default, in that case the distance is considered as zero).
  - labelsize := ⟨ small/... ⟩. Defines the label size like LabelSize (equal to *LabelSize* by default) if the option *TeXify* is 0.
  - labelstyle := ⟨ label type ⟩. Defines the style label like LabelStyle (equal to *LabelStyle* by default).
  - showdot := ⟨ 0/1 ⟩. Shows if the anchor has to be drawn or not (0 by default).
- if *showdot* is 1, options of *bdDot* (p. 151) can be used because that macro will be called.
- Example(s):

```
\begin{texgraph}[name=texify, file]
Graph image = [
Marges(0,0,0,0),view(-3,3,-3,3),
view3D(-3,3,-3,3,-3,3), size(7.5),
B:=M(0,2,0), A:=M(0,0,0), C:=M(0,0,1.5),
Build3D(
 bdAxes([0,0], [color:=gold,arrows:=1]),
 bdPlan([0,0,1+i,2], [color:=darkseagreen,
                      scale:=0.75]),
 bdSurf(M(u,-v,sqrt(u^4+v^4)-2),
        [color:=steelblue, u:=[-2,2],
         v:=u, smooth:=1,clip:=1,
         clipwin:=[M(-3,-3,-3),M(3,3,2)]]),
 bdLabel([0.25*(1+i),2.25],"z=\sqrt{x^4+y^4}-2",
        [TeXify:=1, scale:=0.75])
),
Display3D()
];
\end{texgraph}
```

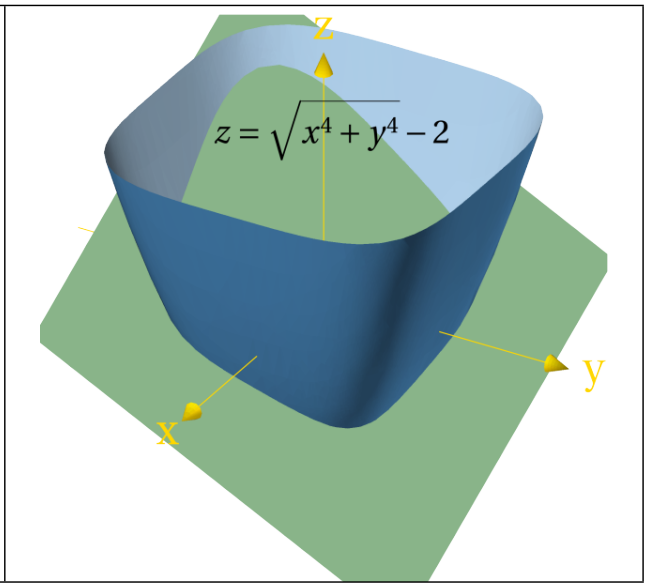


Figure 3: *Option usage of TeXify*

2.13 bdLine

- **bdLine(<3Dpoint list>, [options])**
- Description: defines a polyline in the space. bdLine options:
  - arrows := ⟨ 0/1/2 ⟩. Shows if there are arrow(s) or not (none, one or two, none by default). That option assumes that the line does not contain the constant *jump*.
  - arrowscale := ⟨ nombre positif ⟩. Scale factor for the arrows (1 by default).
  - clip := ⟨ -1/0/1 ⟩. Shows if the line has to be clipped with the window defined by the option *clipwin* if *clip* is 1, or with the plane defined by the option *clipwin* if *clip* is -1 (clip=0 by default).
  - clipwin := ⟨ [M(xinf,yinf,zinf), M(xsup,yup,zsup)] ⟩. Defines the 3D window for an eventual clipping if *clip* vaut 1, the window is then given by its great diagonal: [M(xinf,yinf,zinf), M(xsup,yup,zsup)] (this is the current window by default). But if *clip* is -1, the option *clipwin* is interpreted as a plane: [3Dpoint, normal vector].
  - close := ⟨ 0/1 ⟩. Shows if the line will be closed or not, (0 by default).
  - color := ⟨ color ⟩. Color setting (black by default).
  - hollow := ⟨ 0/1 ⟩. If the option *tube* is 1, the line is replaced by a tube with facets. That tube can be hollow (value 1) or not (value 0). 0 is the default value.
  - linestyle := ⟨ line style ⟩. Defines the line style (solid by default).
  - nbfacet := ⟨ facets number ⟩. Defines the facets number if *tube* is 1 (4 facets by default).
  - opacity := ⟨ number between 0 and 1 ⟩. Opacity value (1 by default), introduces the transparency if the opacity is strictly less than 1.
  - radius := ⟨ tube radius ⟩. The tube radius if *tube* is 1 (0.01 by default).
  - radiusscale := ⟨ number>0 ⟩. Scale factor for the tube radius if *tube* is 1 (1 by default).
  - tube := ⟨ 0/1 ⟩. Shows if a tube (with facets)has to be built or not around the line (0 by default).
  - width := ⟨ line thickness ⟩ (8 by default).
  - matrix := ⟨ 3D matrix ⟩. Defines a transformation matrix applied to the points of the line (identity by default). The transformation is done before the eventual clipping.
  - above := ⟨ positive number or zero ⟩. Places the line above the scene, translated with the vector `above*500*\n` (0 by default).
  - hidden := ⟨ 0/1 ⟩. Shows that the hidden lines have to be drawn if border=1. If yes, then the variables *HideStyle* and *HideWidth* are used. By default, that option has the value of the general option *hiddenLines*.
- If the option *tube* is 1, the macro bdFacet is called, the options of *bdFacet* (p. 151) can then be used.

2.14 bdPlan

- **bdPlan(<plane>, [options])**
- Description: defines a plane. That <*plane*> is represented by a list in the form: [3D point, 3D normal vector]. bdPlan options:
  - scale := ⟨ strictly positive number ⟩. The plane is intersected with the current 3D window, thus giving a facet that can be scaled.
- That macro calls bdFacet, in that case the options of *bdFacet* (p. 151) can be used. By default, the option *twoside* is 0 (the two sides of the facet are not distinguished).

```
\begin{texgraph}[name=intersection, file]
Graph image = [
Marges(0,0,0,0), ModelView(central), DistCam(20),
view(-6,6,-6,6), size(7.5),
theta:=-10*deg, phi:=60*deg,
P1:=planEqn([1,1,1,2]),P2:=[Origin, vecK-vecJ],
D:= interPP(P1,P2),
a:=Copy(getdroite(D),1,2),
b:=Copy(getplan(P1,0.75),11,2),
c:=Copy(getplan(P2,0.75),3,2),
Build3D(
 bdPlan(P1, [color:=red, opacity:=0.7,
          scale:=0.75 ]),
 bdPlan(P2, [color:=blue,opacity:=0.7,
         scale:=0.75]),
 bdDroite(D, [color:=darkgreen,
          width:=12]),
 bdAxes([0,0],[color:=gold,
          width:=8, arrows:=1]),
 bdLabel(a,"$D$",[labelpos:=[0.5,-i]]),
 bdLabel(b,"$P_1$",[labelpos:=[0.5,i]]),
 bdLabel(c,"$P_2$",[labelpos:=[0.5,i]])
      ),
Display3D()
];
\end{texgraph}
```

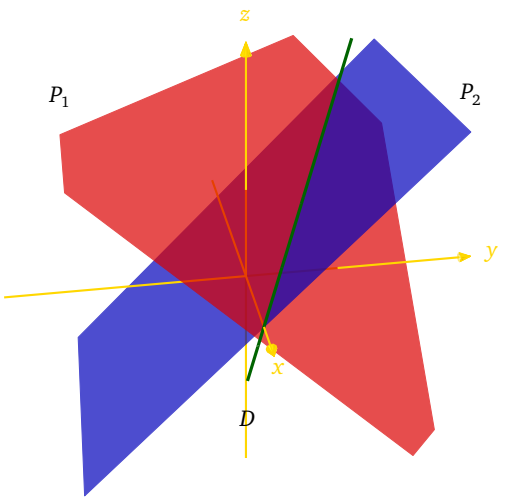


Figure 4: *Intersection of 2 planes*

2.15 bdPlanEqn

- **bdPlanEqn(<[a,b,c,d]>, [options])**
- Description: defines the plane with the equation $ax + by + cz = d$, it is represented by the list: *<[a,b,c,d]>*. Here are the bdPlanEqn options:
  - scale := ⟨ strictly positive number ⟩. The plane is intersected by the current 3D window, thus giving a facet that can be scaled..
- That macro calls bdFacet, in that case the options of *bdFacet* (p. 151) can be used. The option *twoside* is 0 by default (the two sides of the facet are not distinguished).

2.16 bdPrism

- **bdPrism(<3Dpoint list>, <3Dvector>, [options])**
- Description: defines the prism built from a *<3Dpoint list>* representing the basis (supposed to be in a plane), and a translation *<3Dvector>* to calulate the other basis. bdPrism options are those of *bdFacet* (p. 151), plus:
  - hollow := ⟨ 0/1 ⟩. Shows if the prism is hollow or not (1 by default).

  If the option border is 1, the macro *bdLine* (p. 153) is called, its options can then be used.

2.17 bdPyramid

- **bdPyramid(<3Dpoint list>, <3Dpoint>, [options])**
- Description: defines the pyramid built from a *<3Dpoint list>* representing the base (supposed to be in a plane), and a *<point3D>* representing the tip. bdPyramid options are those of *bdFacet* (p. 151), plus:
  - hollow := ⟨ 0/1 ⟩. Shows if the pyramid is hollow or not (1 by default).

  If the option border is 1, the macro *bdLine* (p. 153) is called, its options can then be used.

2.18 bdSphere

- **bdSphere(<3Dpoint>, <radius R>, [options])**
- Description: defines a sphere with center <*point3D*>, and <*radius R*>. The options are those of *bdFacet* (p. 151) plus:
  - grid := ⟨ [nb meridians, nb parallels] ⟩. Number of meridians and parallels to define the facets. By default: [40,25].
  - border := ⟨ 0/1 ⟩. Shows if the border has to be drawn or not (0 by default).
  - bordercolor := ⟨ color ⟩. Shows the broder color (by default identical to *color*).

2.19 bdSurf

- **bdSurf(<f(u,v)>, [options])**
- Description: defines a surface parametrized by $f(u,v) = [x(u,v) + i * y(u,v), z(u,v)] = M(x(u,v), y(u,v), z(u,v))$, with x, y and z that are functions of the variables u and v. bdSurf options:
  - u := ⟨ [umin, umax] ⟩. interval for the variable u, [-5,5] by default.
  - v := ⟨ [vmin, vmax] ⟩. interval for the variable v, [-5,5] by default.
  - grid := ⟨ [unbdot, vnbdot] ⟩. Defines the grid, ie the number of points for u and v. By default the value is [25,25].
- That macro calls bdFacet, therefore options of *bdFacet* (p. 151) can be used.

2.20 bdTorus

- **bdTorus(<3Dpoint>, <radius R>, <radius r>, <3D normal vector>, [options])**
- Description: defines a torus with center <*3Dpoint*>, great <*radius R*>, small <*radius r*>, the <*3D normal vector*> sets the "plane of the torus". bdTorus options are those of *bdFacet* (p. 151), plus:
  - grid := ⟨ [nb meridians, nb parallels] ⟩. Parallels and meridians numbers to define the facets. By default: [40,25].

```
\begin{texgraph}[name=villarceau, file]
Graph image = [
view(-6,6,-5,5),Marges(0,0,0,0),size(7.5),
$R:=3, $r:=1,
N:=rot3d(vecK,[Origin,vecI],arcsin(r/R)),
view3D(-5,5,-5,5,-5,5),
background(full,lightgray),
Build3D(
 bdPlan([Origin, -N],
   [color:=seagreen, opacity:=0.8]),
 bdTorus( Origin, R, r, vecK,
   [color:=steelblue, smooth:=1]),
 view3D(-5.5,5.5,-5.5,5.5,-5,5),
 bdAxes( Origin,
   [arrows:=1, newxlegend("x"),newylegend("y"),
    newzlegend("z")]),
 bdCercle(M(r,0,0),R,N,[color:=red, tube:=1]),
 bdCercle(M(-r,0,0),R,N,[color:=red, tube:=1])
      ),
Display3D()
];
\end{texgraph}
```

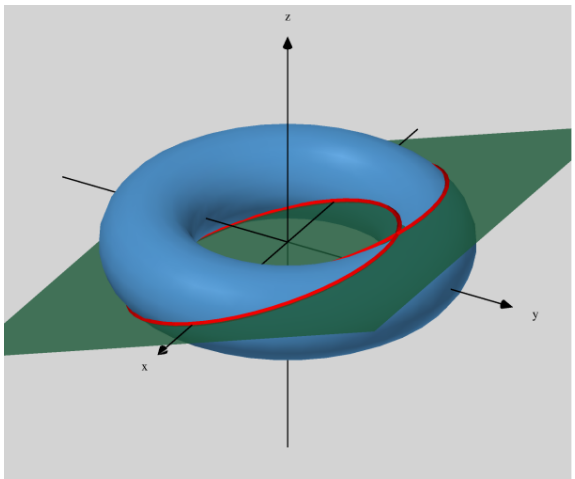


Figure 5: *villarceau circles*

3) obj, geom and jvx exports

3.1 Scene built using Build3D

Three new exports appeared at the bottom of the *File (Fichier)* menu, those only apply on the scene built with the command *Build3D()*. Those exports are:

1. **obj** format: *obj* files can be read by most of the great 3D software, like *Blender* (`http://www.blender.org/`) for example.

2. **geom** format: the geom are dedicated to the program *geomview* (`http://www.geomview.org/`) that allows to move the figure in the space using the mouse.

3. **jvx** format: the *jvx* files are dedicated to the *javaview* applet (`http://www.javaview.de/`) that allows to move the figure using the mouse, and many other options to manipulate the scene (like hidding some elements, exports...) using a control panel. The display can be done on a web page, or locally in a java window.

Those three exports can also be activated with the commands:
Export(obj, <file name>) or Export(geom, <file name>) or Export(jvx, <file name>).
<file name> is the full file name with its extension.

3.2 Building a Scene without Build3D

It is also possible to export a scene in the formats *obj*, *geom* and *jvx* without the Build3D command:

- **SceneToObj(<file name>, <element1>, <element2>, ...)**

- **SceneToGeom(<file name>, <element1>, <element2>, ...)**

- **SceneToJvx(<file name>, <element1>, <element2>, ...)**

- Description: the argument *<file name>* is the full file name without its extension, it will be automatically added. The following arguments are the elements composing the scene, **these are the same arguments that would have been used with the Build3D command**, that is why the macro created for *Build3D* (p. 148) can be used(bdAxes, bdArc, ...).

3.3 Isolated element export

There are two other macros for the exports:

- **WriteObj(<file name>, <vertices list>, <facets list> [, lines list])**,

- **WriteOff(<file name>, <vertices list>, <facets list> [, lines list])**,

- Description: the argument *<file name>* is the full file name without extension, it will be automatically added. The following argument is the 3D points list that are representing the facet's vertices and/or the lines that are following. The third argument is the facets list where **each vertex is replaced by its position number in the vertices list** (same with the last argument). This is the natural file format for the *obj* files. The command *ConvertToObj* (p. 113) can be used to to that conversion.

The format *off* is a format of the *geomview* software.

Chapter XII

TeXgraph code in a LaTeX file

1) Installation

Under windows, you will have to copy the file *texgraph.sty* in your TEX tree and update the base. Under linux, installing the package *texgraph.sty* is automatically done with the execution of the script *install.sh*.

WARNING: compiling a LATEX document with that package, must be done with the option –shell-escape (or –enable-write18 according to your distribution).

2) The *texgraph* environment

Once declared with: `\usepackage{texgraph}`, the following environment can be used:

```
\begin{texgraph}[<options>]
 <TeXgraph code>
\end{texgraph}
```

At the compilation stage, the code is copied in a file named *<name>.teg* (TeXgraph source file) as a User graphical element (by default), then the program *TeXgraphCmd* is called, it loads the file *<nom>.teg*, exports the results in the asked format, and finally, the LATEX compiler gets the hand and the resulting file is loaded with `\input` or `\includegraphics` according to the requested export.

To be fully precise, this is a script that is called: *CmdTeXgraph*.

Possible options are:

- name = < name >: give a name to the image (without extension), this is by default the current file name followed by the order of appearance in the environment (file1, file2, ...). That parameter must be given at first place if not omitted
- export = < none/pst/pgf/tkz/eps/psf/pdf/epsc/pdfc/teg/texsrc >: That parameter can take the following values: *none* (no file is exported), *pst* (pstricks, default option), *pgf*, *tkz* (pgf code in a tikzpicture environment thus allowing to add tikz intructions), *eps*, *psf* (eps+psfrag), *pdf*, *epsc* (compiled eps), *pdfc* (compiled pdf), *teg* (texgraph source file) or *texsrc* (coloured texgraph source file for TEX). The export type is automatically determined and also the inclusion mode (input or includegraphics or nothing).
- call = < true/false >: that boolean is by default *true*. It shows if TeXgraph is really called, if not, the TeXgraph code is ignored, thus avoiding useless calls in case of multiple compilations, the image file is though included, according to the parameter *auto*. If *call* value is *true*, a *<fichier>.teg* file is created, compiled using *TeXgraphCmd* that is then exporting an image file and a log file.
- auto = < true/false >: that boolean is *true* by default, it shows if the image file has to be automatically included using macros “input” or “includegraphics”. If not, the image file is not loaded. If this option is not omitted, that option has to be put after the export option.
- commandchars = < true/false >: that boolean is *false* by default. If its value is *true*, the environment can contain TEX commands but the \ before each command has to be replaced by # , eg: `#command{...}`. If that command is containing macros that must to be not developed, they will be preceeded by `\noexpand`.
- src = < true/false >: that boolean is *false* by default, If the value is *true*, TeXgraph will export, added to the graphic, the source file coloured for TEX (file with the extension *src*), and this is the source file that is included, replacing the environment, like in all the examples that can be seen in this document. All the colors are predefined in the file *texgraph.sty* and can be changed by the user in his document. Here are the definitions:

```
\newcommand*{\TegSrcFontSize}{small}%font size
\definecolor{TegIdentifier}{rgb}{0.5451,0.2706,0.0745}%
\definecolor{TegComment}{rgb}{0.502,0.502,0.502}%
\definecolor{TegNumeric}{rgb}{0.0000,0.5020,0.5020}%
\definecolor{TegConstant}{rgb}{0.5020,0.5020,0.0000}%
\definecolor{TegString}{rgb}{0,0,1}%
\definecolor{TegSymbol}{rgb}{1,0,0}%
\definecolor{TegKeyWord}{rgb}{0,0,0}%
\definecolor{TegVarGlob}{rgb}{0.0000,0.0000,0.5020}%
\definecolor{TegMacUser}{rgb}{0.5020,0.0000,0.5020}%
\definecolor{TegVarPredef}{rgb}{0.0000,0.3922,0.0000}%
\definecolor{TegMacPredef}{rgb}{0.5020,0.0000,0.0000}%
\definecolor{TegParam}{rgb}{1.0000,0.0000,1.0000}%
\definecolor{TegGraphElem}{rgb}{0.4392,0.5020,0.5647}%
```

- file = < true/false >: that boolean is *false* by default, it shows if the inside of the environment is a full TeXgraph source file (*file=true*), or only a User graphical element (*file=false*).
- preload = < {"<file1>";"<file2>";...} >: load one or several packages before creating the graphic, eg: *preload={"papiers.mod";"draw2d.mod"}*.
- cmdi = < command >: import the graphic within the command, eg: `cmdi={\raisebox{-2cm}}`
- cmdii = < command >: applies a second command over the first one(*cmdi*).

The package provides three global options:

- nocall: that options redefines the default value of the option *call* to the value *false*, then texgraph environments will call the program *TeXgraphCmd* only if the option *call* (or *call=true*) is mentioned.
- src: changes the default value of the option *src* and set it to the value *true* for all the texgraph environments.
- export = < pst/pgf/tkz/eps/psf/pdf/epsc/pdfc >: that option redefines the default export.
- server: that option is allowing TeXgraph to be launched in server mode, and close the program at the end of the compilation. Then the program is executed only once for the whole document.

Exemples: `\usepackage[nocall]{texgraph}` or `\usepackage[export=pgf,server]{texgraph}`.

WARNING

- Commands related to the graphic interface (the mouse, the menu, buttons, items, the timer, ...) are ignored.
- When a line is starting with a comment between braces an error will occur if the option *commandchars* is activated. Though, beginning a line with a comment is done like the following: `//blablabla` (The whole line is then a comment).

3) Examples

With the option *file=false* (default value), the TeXgraph code is included in a user graphical element before being sent to the program *TeXgraphCmd*:

```
\begin{texgraph}[name=surf1,export=pgf]
view(-7,7,-7,7), Marges(0,0,0,0),
size(7.5), FillStyle:=full,
FillColor:=lightblue,
Dsurface( M(u,v,cos(u)+sin(v)),
          -5+5*i,
          -5+5*i, 25+25*i)
\end{texgraph}
```

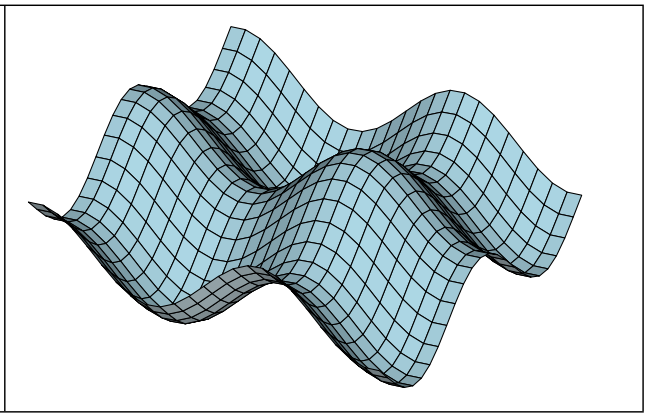


Figure 1: One example with file=false

In that first example, the file that is really sent to the program is:

```
TeXgraph#
Graph image = [
   view(-7,7,-7,7), Marges(0,0,0,0),
   size(7.5), FillStyle:=full,
   FillColor:=lightblue,
   Dsurface( M(u,v,cos(u)+sin(v)),
             -5+5*i,
             -5+5*i, 25+25*i)
   ];
```

With the option *file=true*, the TeXgraph code is considered as a source file for the program *TeXgraphCmd*:

```
\begin{texgraph}[name=polyedre,export=pgf,file]
Cmd Marges(0,0,0,0); size(7.5);
Include "PolyedresII.mac";
Var A = M(3,0,0);
Mac f = M(%1,%2,sin(%1)+cos(%2));
Graph objet1 = [
  background(full,lightgray),
  ColorL:=slategray, StyleL:=12,
  CubeAdc(Origin, A,C,T1,T2,Ar),
  Build3D(C,T1,T2,Ar,
     bdSurf(f(u,v),
       [color:=steelblue,
        u:=[-pi,pi],v:=u,
        contrast:=0.125]),
     bdAxes(Origin,
       [color:=forestgreen,arrows:=1])
     ),
  Display3D()];
\end{texgraph}
```

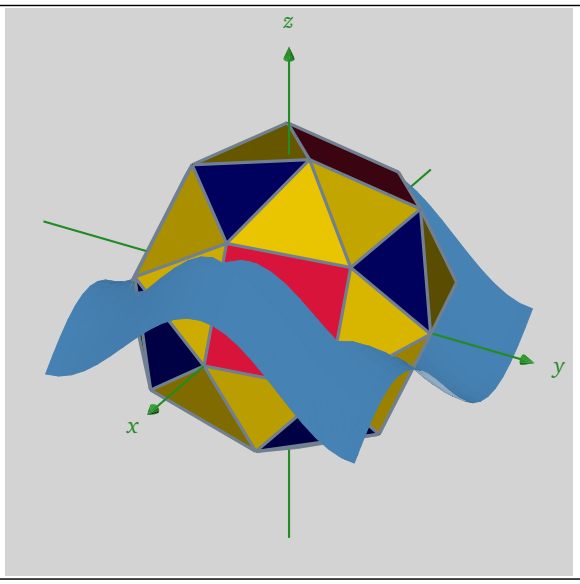


Figure 2: One example with file=true

4) Source file syntax

In that second example, the file that is really sent to the program is:

```
TeXgraph#
Cmd Marges(0,0,0,0); size(7.5);
Include "PolyedresII.mac";
Var A = M(3,0,0);
Mac f = M(%1,%2,sin(%1)+cos(%2));
Graph objet1 = [
  background(full,lightgray),
  ColorL:=slategray, StyleL:=12,
 CubeAdc(Origin, A,C,T1,T2,Ar),
 Build3D(C,T1,T2,Ar,
     bdSurf(f(u,v),
       [color:=steelblue,
        u:=[-pi,pi],v:=u,
        contrast:=0.125]),
     bdAxes(Origin,
       [color:=forestgreen,arrows:=1])
     ),
  Display3D()];
```

- The first line (`TeXgraph#`) is automatically added. It tells the following is a source file (according to the 1.95 versions and newer, the sources of the old versions still remain compatible).
- The **Cmd** part contain commands, each command is ended by a semicolon, the commands are interpreted as and when the file is read.

- The part entitled **Include** shows the files to be loaded, each file name is a string followed by a semicolon, the files are loaded as and when it is read.
- The **Var** part contains the declarations of the global variables. The syntax to be used is:

  <name> = <expression> ;

  The <*expression*> is evaluated before affectation to the global variable <*name*>. Declarations are executed as and when the file is read.
- The **Mac** part contains the declarations of the macros. The syntax to be used is:

  <name> = <expression> ;

  The <*expression*> is analysed and if there is no errors a macro called <*name*> is created with the <*expression*>. The declarations are executed as and when the file is read.
- The **Graph** part contains the user graphic elements declarations. The syntax to be used is:

  <name> = <expression> ;

  The <*expression*> is analysed and if there is no errors, a graphical element called <*name*> is created with that <*expression*>. The graphical elements are created as and when the file is read.

Some rules:

1. The first line is mandatory.
2. There can be several parts *Cmd*, *Include*, *Var*, *Mac* and *Graph*.
3. The differents part do not have to follow a particular order. The only thing is to remember that a global variable (or a macro) only exists after its declaration.

5) The *tegprog* environment and the *tegrun* macro

The *texgraph.sty* package also provides the environment:

```
\begin{tegprog}[<options>]{name}
 <code TeXgraph>
\end{tegprog}
```

That environment saves the program <*name*>.teg. That program is created to be executed by the command *tegrun*, the parameters that will be used will be in the global variable *param* (list) of the program. That program also has a macro *Return(string)*, that is writing the string in an output file. That file will be automatically included by the command *tegrun*.

Possible options are:

- file: that option shows that the inside of the environment is a whole TeXgraph source file, if not, this is only a command.
- commandchars: with that option, the environment may contain calls to TEX commands if the backslash \ is replaced by # before the commands name, ex: `#command{...}`. If this command contains macros that must not be developped, they will have to be preceeded by `\noexpand`.
- preload = < {"<file1>";"<file2>";...} >: loads one or several packages before creating the graphic, eg: *preload={"papiers.mod";"dr*

Once saved, a program can be executed in a TEX document with the command:
`\tegrun{name}{param1 param2 ...}`, it is saving the parameters in the file <*name*>.prm, ask TeXgraph to execute the program <*name*>.teg, and include the result file <*name*>.res. Here is an example:

```
\begin{tegprog}{PrintPgcd}
 a:=param[1], b:=param[2],
 if a<b then Echange(a,b) fi,
 Return("\begin{tabular}{|c|c|c|}\par\hline{}a&b&r\tabularnewline\hline"),
 r:=b,
 while r>0 do
    r:=mod(a,b),
    Return(Concat(a,"&",b,"&",r,"\tabularnewline\hline")),
    a:=b, b:=r
 od,
 Return("\end{tabular}")
\end{tegprog}
\newcommand{\PrintPgcd}[2]{\tegrun{PrintPgcd}{#1 #2}}%
```

In that example, the program *PrintPgcd.teg* is created, it calculates the gcd of two integers *a* and *b* by giving all the steps of the Euclid algorithm in the form of a tabular. The parameters list is in the variable *param*[1]. The macro *Return* write in the output file whose name is *PrintPgcd.res*.

Then a two arguments macro called *PrintPgcd* is created. That macro calls the command `\tegrun{PrintPgcd}{#1 #2}`, that command write the two arguments in the parameter file *PrintPgcd.prm*, and ask TeXgraph to execute the program *PrintPgcd.teg*, and finally includes the file *PrintPgcd.res*.

The execution of `\PrintPgcd{456}{166}` gives

a	b	r
456	166	124
166	124	42
124	42	40
42	40	2
40	2	0

.

6) The *tegcode* environment and the *directTeg* macro

If the package *texgraph.sty* is called with the *server* option, it provides the environment:

```
\begin{tegcode}
 <TeXgraph file>
\end{tegcode}
```

The syntax is the same as a source file without its first line: `TeXgraph#`, that will be automatically added. The file may contain TEX macros if the backslash \ is replaced by # before the commands names, eg: `#command{...}`. Once declared, the file is read by Texgraph and **will remain in memory until the end of the document**. The variables and macros defined in that file will then be available for other calls to TeXgraph. Those macros can use the instruction *Return(string)*, only if it is then used by the macro `\directTeg`.

The macro `\directTeg{command}` launches the <*command*> via TeXgraph, that <*command*> may use the macro *Return(string)*, this one is writing the string in an output file, and that file will be automatically included by the macro `\directTeg`. Here is an example:

```
\begin{tegcode}
Mac Gcd = [//Gcd(integers list)
        $L:=%1, $N:=Nops(L),
        if N<2 then "error !"
        else
            $r:=pgcd(L[1],L[2]),
            if r=1 then 1
            elif N=2 then r
            else Gcd( [r,L[3,0]] )
            fi
        fi
        ];
\end{tegcode}
\newcommand*{\Gcd}[1]{\directTeg{Return(Gcd([#1]))}}%
```

Execution of `\Gcd{12,68,36}` gives 4. Execution of `\Gcd{12}` gives error !.

[1] The program initializes that variable by reading the file *PrintPgcd.prm* that is containing the parameters list.

Index