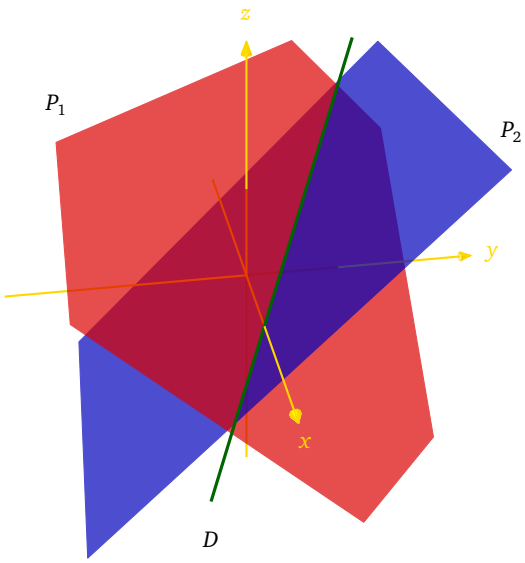
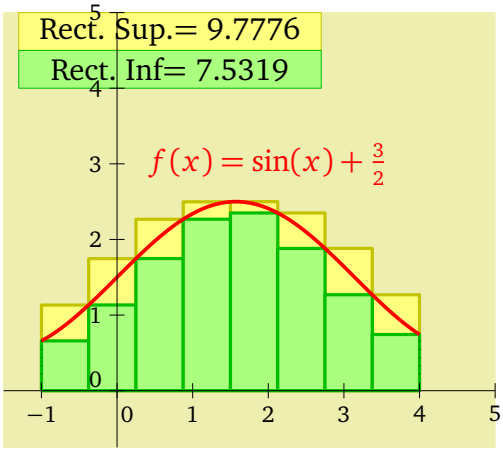
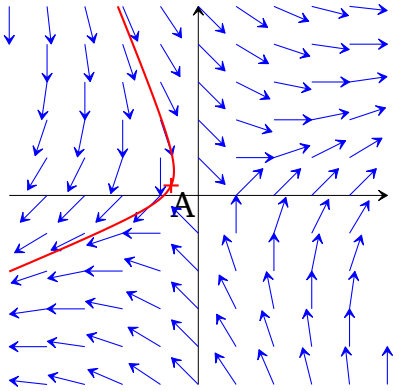
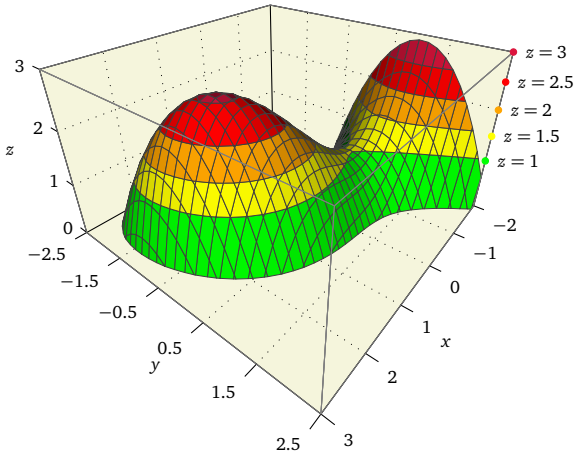
Aide de
TEXGRAPH 1.97









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

Patrick FRADIN

4 avril 2012

Table des matières

Table des figures

Chapitre I

Introduction à TeXgraph

1) Présentation

- TeXgraph est un programme permettant la création de graphiques mathématiques (comme les courbes, les surfaces, les constructions géométriques...), ainsi que leur exportation sous forme de fichiers textes aux formats : LaTeX (macros eepic), ou PsTricks, ou Pgf/Tikz (macros pgf), ou Eps, ou Psf (eps+Psfrag), ou pdf (conversion eps -> pdf) ou svg ... Il existe également des exports spécifiques à la 3D.
- Il a été écrit pour Windows et pour Linux.
- TeXgraph version 1.97 est distribué sous les termes de la licence GPL (General Public Licence).
  Cette version est une version écrite en Free Pascal avec Lazarus (0.9.31 Svn).
  Ce programme est libre, vous pouvez le redistribuer et/ou le modifier selon les termes de la Licence Publique Générale GNU publiée par la Free Software Foundation (version 2 ou bien toute autre version ultérieure choisie par vous).
  Ce programme est distribué car potentiellement utile, mais SANS AUCUNE GARANTIE, ni explicite ni implicite, y compris les garanties de commercialisation ou d'adaptation dans un but spécifique. Reportez-vous à la Licence Publique Générale GNU pour plus de détails.
  Vous devez avoir reçu une copie de la Licence Publique Générale GNU en même temps que ce programme ; si ce n'est pas le cas, écrivez à la Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, États-Unis.
- Vous rendrez service à l'auteur et à la communauté en signalant les bogues :
  par courrier électronique à l'adresse Email : texgraph@tuxfamily.org.
  ou à l'adresse postale :

  Patrick FRADIN
  La Jauvigère
  17, Impasse du Vieux Château
  16590 BRIE
- Le programme TeXgraph peut être téléchargé depuis : http://texgraph.tuxfamily.org/
  Des exemples peuvent également être consultés à cette même adresse. Ainsi que sur le site : http://melusine.eu.org/syracuse/
- Un forum de discussion sur le logiciel (contenant aussi une rubrique Exemples) se trouve à : http://texgraph.tuxfamily.org/forum/

2) Lancement de TeXgraph

Le programme nécessite une installation (voir le fichier *LisezMoi.txt*), l'exécutable s'appelle *TeXgraph*, et se lance par le script **startTeXgraph**.

Le dossier d'installation contient le dossier *TeXgraph* où se trouvent les exécutables ainsi que les dossiers *Exemples*, *doc* et *macros*.

TeXgraph gère trois types de fichiers : les fichiers sources (**.teg*), les fichiers modèles (**.mod*) et les fichiers de macros (**.mac*).

- Les fichiers **.teg* : sont les fichiers que l'on obtient lorsque l'on sauvegarde un graphique. Ce sont donc les fichiers sources ordinaires.
- Les fichiers **.mod* : sont les fichiers modèles destinés à être chargés, on peut les considérer comme des fichiers sources prêts à l'emploi que l'on peut ensuite compléter à sa guise.
- Les fichiers **.mac* : sont les fichiers de macros destinées à être chargées. Ils peuvent aussi contenir des déclarations de variables. Contrairement aux deux précédents, **tout ce que contient un fichier **.mac* est considéré comme prédéfini** et ne sera donc pas sauvegardé avec le graphique (par contre le fichier source contiendra une commande ordonnant le chargement de ce fichier de macros).

Ces trois types de fichiers obéissent aux mêmes règles de syntaxe, celle-ci est décrite dans la section *src4latex* (p. 151).

Au lancement du programme, celui-ci charge plusieurs fichiers de macros : *TeXgraph.mac*, *couleurs.mac*, *scene3d.mac* et *interface.mac* (ce dernier est chargé uniquement avec la version GUI). Le contenu de ces fichiers est considéré comme prédéfini et restera en mémoire jusqu'à la fermeture du programme.

Il est également possible de charger un ou plusieurs autres fichiers de macros au lancement du programme en les ajoutant comme paramètres dans la ligne de commande. De la même façon, les contenus ainsi chargés au démarrage sont considérés comme prédéfinis et ne seront supprimés de la mémoire qu'en quittant le programme.

On charge un fichier de macros par le biais du menu avec l'option *Fichier/Charger des macros*. Les variables et macros ainsi chargées sont également considérées comme prédéfinies et ne feront pas partie des graphiques, **par contre elles seront supprimées de la mémoire au prochain changement de fichier**. Les variables et macros chargées avec l'option *Fichier/Importer un modèle* viennent s'ajouter au graphique en cours, elles seront enregistrées avec lui, et **elles seront supprimées au prochain changement de fichier**.

Pour un fonctionnement complet et correct de TeXgraph, votre système est supposé être équipé de :

1. Une distribution TEX correctement installée, avec en particulier les packages *tikz/pgf*, *pstricks*.
2. La suite ImageMagic pour toutes les conversions d'images (bouton *Snapshot* ou les gifs animés du modèle *Animation.mod*).
3. La suite swftools si vous utilisez le modèle *Animation.mod* avec une sortie Flash.
4. Le programme pstoedit qui est utilisé pour convertir des formules TEX compilées en chemins.
5. Le programme povray si vous utilisez le modèle *povray.mod*.

Si de plus vous utilisez les exports 3D *geom* et *jvx*, il vous faudra pour les visualiser :

1. Le programme geomview : pour les fichiers **.geom*.
2. Le programme javaview : pour les fichiers **.jvx*. Ce programme peut être exécuté en local, mais aussi sous forme d'applet dans une page web, comme on le peut le voir sur cette page.

Tous ces programmes sont libres disponibles gratuitement pour linux et windows.

3) Composition d'un graphique

Un graphique est la donnée de :

- *Paramètres* (p. 10) : comme les coordonnées de la fenêtre graphique, les échelles sur les axes, les marges ....
- *Variables globales* (p. 33) : celles-ci contiennent en général une liste de complexes, éventuellement la valeur *Nil*.
- *Macros* (p. 34) : celles-ci servent à simplifier la composition du graphique.
- *Eléments graphiques* (p. 13) : comme les axes, les courbes,...

4) Les paramètres

Ceux-ci correspondent à l'option *Paramètres* du menu, on y trouve les options :

- **Fenêtre** : permet de définir la zone rectangulaire du plan où s'éffectue le tracé, on précise les valeurs des "constantes" : Xmin, Xmax, Ymin, Ymax, puis l'échelle sur les deux axes : Xscale, Yscale en cm. Ces constantes peuvent être utilisées dans les commandes, mais pas modifiées directement, à moins d'utiliser la commande *Fenetre* (p. 41). Le repère est orthonormé lorsque Xscale=Yscale.
- **Marges** : permet de définir des marges autour du graphique en cas de débordement de labels par exemple. On précise les valeurs des "constantes" : margeG, margeD, margeH, margeB, en cm. Ces constantes peuvent être utilisées dans les commandes, mais pas modifiées directement, à moins d'utiliser la commande *Marges* (p. 45).
- **Exporter la bordure** : si cette option est cochée, il y aura un cadre autour du dessin lors de l'exportation, comme à l'écran. Ce cadre est un trait plein noir qui englobe également les marges. Cette option peut-être modifiée avec la commande *Border* (p. 36).
- **Exporter les couleurs** : si cette option est décochée le graphique sera exporté en nuances de gris.
- **Exporter les noms** : si cette option est cochée, il y aura en commentaire dans le fichier exporté le nom de chaque élément graphique juste avant leur tracé. Ceci permet de les retrouver facilement dans les exportations en LaTeX, pgf ou pstricks si on veut y effectuer des modifications.
- **Afficher les variables globales** : si cette option est cochée, les variables globales sont affichées à l'écran (mais leur affichage ne sera pas exporté) ce qui peut servir de points de repère.
- Il est également possible de masquer les colonnes de gauche et/ou de droite de l'interface graphique, ainsi que monter/cacher le point d'ancrage des labels.

5) Les couleurs

5.1 Couleurs prédéfinies

La liste des couleurs prédéfinies est sur cette page couleurs.html.

5.2 Commandes et macros liées aux couleurs

- **Lcolor(<couleur> [, niveau de gris])** : macro qui renvoie les trois composantes red, green, blue de la couleur sous forme d'une liste [r,g,b]. Le deuxième argument est facultatif et vaut 0 par défaut, lorsqu'il vaut 1 la couleur est convertie en niveau de gris avant le calcul des composantes.
- **Bcolor(<couleur>)** : macro qui renvoie la composante bleue de la couleur.
- **Gcolor(<couleur>)** : macro qui renvoie la composante verte de la couleur.
- **Rcolor(<couleur>)** : macro qui renvoie la composante rouge de la couleur.
- **CplColor(<couleur>)** : macro qui renvoie la couleur complémentaire.
- **Dark(<couleur>, <facteur>)** : macro qui fait un barycentre entre la couleur et le noir, le facteur est entre 0 et 1 et représente la proportion de noir (1=100%).
- **Light(<couleur>, <facteur>)** : macro qui fait un barycentre entre la couleur et le blanc, le facteur est entre 0 et 1 et représente la proportion de blanc (1=100%).
- **GrayScale(<0/1>)** : cette commande est décrite *ici* (p. 42). Elle permet d'activer ou désactiver la conversion des couleurs en nuance de gris.
- **HexaColor(<valeur héxa>)** : cette commande est décrite *ici* (p. 42). Exemple : Color :=HexaColor("F5F5DC").
- **MixColor(<color1>, <proportion1>, <color2>, <proportion2>, ..., <colorN>, <proportionN>)** : macro qui renvoie la couleur (rgb) obtenue après le mélange des différentes couleurs passées en arguments en suivants les proportions correspondantes.
- **Palette(<[Color1, Color2, ..., ColorN]>, <facteur dans [0 ;1]>)** : renvoie une couleur de la palette en fonction du facteur, 0 pour la première couleur et 1 pour la dernière.
- **Hsb(<hue (0..360)>, <saturation (0..1)>, <brightness (0..1)>)** : macro qui renvoie une couleur à partir de ses composantes hue, saturation , brightness. Exemple : Color :=Hsb(60,1,1).
- **HueColor(<couleur>)** : renvoie la composante hue de la couleur.
- **SatColor(<couleur>)** : renvoie la composante saturation de la couleur.
- **BrightColor(<couleur>)** : renvoie la composante brightness de la couleur.
- **ColorJump(<couleur>)** : macro qui renvoie la constante *jump* avec la <*couleur*> dans la partie imaginaire. La commande *Ligne* (p. 84) lit cette couleur et l'interprète comme la couleur de remplissage à utiliser pour peindre lorsque la variable *FillStyle* n'a pas a valeur *none*.
- **Rgb(<red (0..1)>, <green (0..1)>, <blue (0..1)>)** : cette commande est décrite *ici* (p. 50). Exemple : Color := Rgb(0.5, 1, 0.6).
- **RgbL(<[red, green, blue]>)** : cette macro a le même effet que *Rgb*, sauf que les trois composantes sont sous forme d'une liste.
- **Ryb(<red (0..1)>, <yellow (0..1)>, <blue (0..1)>)** : macro qui renvoie une couleur à partir de ses composantes rouge, jaune, bleu. Exemple : Color := Ryb(0.5, 0.8, 0.6).
- **Rgb2Hsb(<couleur>)** : macro qui convertit une couleur (rgb) en une couleur Hsb, c'est à dire une liste : [hue, saturation, brightness].
- **Rgb2Hexa(<couleur Rgb>)** : renvoie une chaîne représentant la couleur en hexadécimal, par exemple "FF0000" pour le rouge.
- **Rgb2Gray(<couleur Rgb>)** : renvoie la couleur en niveau de gris (au format rgb).

Exemple(s): on colorie chaque facette en fonction de la cote du centre de gravité, on ajoute pour cela cette couleur avec la macro *ColorJump* dans la constante *jump* de fin de facette. La macro *Hsb* permet de faire varier la couleur continûment. Pour dessiner la surface, on trie les facettes avec la commande *SortFacet* (p. 112), puis elles sont dessinées.

```
\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}
```

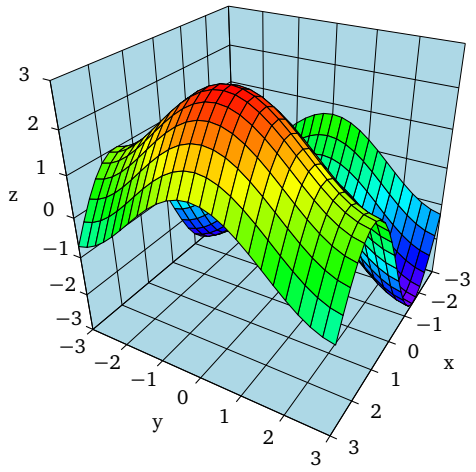


FIGURE 1: *Coloration de type chaleur*

Chapitre II

Eléments graphiques

Un graphique est une superposition d'éléments graphiques [1], ceux-ci peuvent être créés, modifiés et supprimés individuellement. Les éléments graphiques sont indépendants, sauf éventuellement ceux créés par l'utilisateur.

Chaque élément graphique est défini à partir d'un nom (un nom commence par une lettre et contient au plus 35 caractères parmi 0..9, a..z, A..Z, ' et _) et d'une *commande* (p. 25), de plus chaque élément graphique comporte des attributs comme : les couleurs, le style de ligne, l'épaisseur du tracé ...

Il y a un certain nombre d'éléments graphiques de base, ceux-ci peuvent être créés à partir du menu ou d'un raccourci. Ils peuvent également être créés à l'aide d'une commande graphique dans un élément *Utilisateur*.

Les éléments graphiques prédéfinis sont :

1) La grille

Pour tracer une grille de repérage (il peut y en avoir plusieurs).

- Raccourci : *Ctrl+G*
- La fenêtre demande les coordonnées de l'origine, le pas des graduations sur les axes Ox et Oy, celui-ci doit être positif, s'il est nul alors les graduations n'apparaîtront pas.
- Il n'y pas de labels dessinés avec la grille. Si on souhaite que ceux-ci apparaissent, il suffit de créer des axes.
- Commande graphique correspondante : *Grille* (p. 83) (utilisable dans un élément de type Utilisateur).

2) Les axes

Pour tracer des axes orthogonaux.

- Raccourci : *Ctrl+A*
- La fenêtre demande les coordonnées de l'origine, le pas des graduations sur les axes Ox et Oy, celui-ci doit être positif, s'il est nul alors les graduations n'apparaîtront pas.
- On peut régler deux paramètres (qui sont deux variables globales du programme) :
  - *xyticks* : c'est la longueur en cm des graduations sur les axes.
  - *xylabelsep* : c'est la distance en cm entre les labels et l'extrémité des graduations.
- Commande graphique correspondante : *Axes* (p. 79) (utilisable dans un élément de type Utilisateur).

3) Courbes

Pour tracer une courbe plane : cartésienne, polaire, ou paramétrée.

- Raccourcis : courbe paramétrée : *Ctrl+P*, courbe polaire : *Ctrl+O*, courbe cartésienne : *Ctrl+R*.
- On doit donner un nom.
- Puis :
  - Pour une courbe cartésienne $y = f(x)$, on donne l'expression de la fonction f(x).
  - Pour une courbe polaire $r = f(t)$, on donne l'expression de la fonction f(t).
  - Pour une courbe paramétrée $(x(t), y(t))$, on donne l'expression de la fonction $f(t) = x(t) + i * y(t)$.
- On peut régler deux paramètres de la courbe :

1. il y a un ordre d'affichage, on peut modifier celui-ci à la souris en faisant glisser les éléments.

  - Division(s) : c'est un entier positif ou nul qui indique combien de fois TeXgraph peut partager en deux (dichotomie) l'intervalle entre deux valeurs de t consécutives (5 par défaut). Cela augmente le nombre de points là où il y a de brusques variations.
  - Discontinuité : 0 ou 1, si cette valeur vaut 1 et si la distance entre deux points consécutifs est supérieure à un certain seuil, alors une discontinuité est insérée dans la liste de points.
- L'intervalle pour le paramètre t (variables globales *tMin* et *tMax*, s'applique aussi à la variable x des courbes cartésiennes) et le nombre de points (variables *NbPoints*) se règlent dans les Attributs.
- Commandes graphiques correspondantes *Cartesienne(f(x))*, *Polaire(r(t))* et pour les courbes paramétrées : *Courbe* (p. 81) (utilisables dans un élément de type Utilisateur).

4) Equation différentielle

Solution approchée (méthode de Runge-Kutta 4) d'une équation du type : $x'(t)+iy'(t)=f(t,x,y)$ avec une condition initiale $x(t_0)=x_0$ et $y(t_0)=y_0$:

- Raccourci : *Ctrl+E*
- On donne un nom.
- On donne une commande de la forme [f(t,x,y), t0, x0+i*y0],
- On choisit le mode de représentation : points de coordonnées (x,y) ou bien (t,x) ou bien (t,y).
- Commande graphique correspondante : *Equadif* (p. 82) (utilisable dans un élément de type Utilisateur).

5) Fonction implicite

Ensemble des points de coordonnées (x,y) tels que $f(x,y)=0$.

- Raccourci : *Ctrl+I*
- On donne un nom.
- On donne une commande de la forme f(x,y), ou bien [f(x,y),n,m], n représente le nombre de subdivisions de l'axe Ox et m le nombre de subdivisions de l'axe Oy (50 par défaut). Sur chaque pavé ainsi obtenu, on teste s'il y a un changement de signe, auquel cas on applique une dichotomie sur les bords du pavé.
- Commande graphique correspondante : *Implicit* (p. 83) (utilisable dans un élément de type Utilisateur).

6) Courbe de Bézier

Succession de courbes de BEZIER (avec éventuellement des segments de droite).

- Raccourci : *Ctrl+B*
- On donne un nom.
- Puis une commande du type [<liste de points>]. Il y a plusieurs possibilités pour la liste de points :
  1. une liste de trois points [A,C,B], il s'agit alors d'une courbe de Bezier d'origine <*A*> et d'extrémité <*B*> avec un point de contrôle C, c'est la courbe paramétrée par $(1-t)^2A+2t(1-t)C+t^2B$.
  2. une liste de 4 points ou plus : [A1,C1,C2,A2,C3,C4,A3...] : il s'agit alors d'une succession de courbes de Bézier à 2 points de contrôles, la première va de A1 à A2, elle est contrôlée par C1, C2 (paramétrée par $(1-t)^3tA1+3(1-t)^2tC1+3(1-t)t^2C2+t^3A2$), la deuxième va de A2 à A3 et est contrôlée par C3,C4 ...etc. Une exception toutefois, on peut remplacer les deux points de contrôle par la constante *jump*, dans ce cas on saute directement de A1 à A2 en traçant un segment de droite.
- Le nombre de points calculés (par courbe) est modifiable dans les Attributs (variable *NbPoints*).
- Commande graphique correspondante : *Bezier* (p. 80) (utilisable dans un élément de type Utilisateur).
- Exemple(s): [-2, -1+i, i, 1, jump, 1-i, jump, -2-i, jump, -2].

7) Spline cubique

Courbe du troisième degré passant par des points donnés avec ou sans contrainte aux extrémités.

- Raccourci : *Ctrl+S*
- On donne un nom.

- On donne une commande de la forme [v0,A1,A2,...,An,v1] où les complexes v0 et v1 sont les affixes des vecteurs tangents aux extrémités (s'ils sont nuls on considère qu'il n'y a pas de contrainte), et les complexes A1,...,An sont les affixes des points interpolés par la courbe.
- Le nombre de points calculés au total peut être modifié dans les Attributs, c'est la variable *NbPoints*.
- Commande graphique correspondante : *Spline* (p. 86) (utilisable dans un élément de type Utilisateur).

8) Droite

Droite du plan définie par deux points, un point et un vecteur directeur, ou une équation cartésienne.

- Raccourci : *Ctrl+D*
- On donne un nom.
- On donne une commande de la forme :
  - [A,B] pour une droite par les points d'affixes A et B.
  - [A,A+v] pour une droite passant par le point d'affixe A et dirigée par le vecteur d'affixe v.
  - [a,b,c] pour une droite d'équation cartésienne ax+by=c.
- Il est possible de déterminer l'intersection de deux droites avec l'opération Inter. Par exemple, si A,B,C,D sont les affixes de quatre points, alors l'exécution de [A,B] Inter [C,D] donnera l'affixe du point d'intersection de (AB) et (CD) si elles sont sécantes, *Nil* sinon.
- Commande graphique correspondante : *Droite* (p. 81) (utilisable dans un élément de type Utilisateur).

9) Point(s)

Pour tracer un point ou un nuage de points.

- Raccourci : *Alt+P*
- On donne un nom.
- On donne une commande de la forme : [A1,...,An] où A1,...,An sont les affixes des points.
- La valeur de *DotStyle* (style de point) se définit dans les Attributs ainsi que les valeurs des variables prédéfinies : *DotScale*, *DotAngle*, *DotSize*.
- Commande graphique correspondante : *Point* (p. 85) (utilisable dans un élément de type Utilisateur).

10) Ligne polygonale

Pour tracer une ligne polygonale (liste de points) ouverte ou fermée (polygone) ayant une ou plusieurs composantes connexes.

- Raccourci : *Ctrl+L*
- On donne un nom.
- On donne une commande de la forme : [A1,...,An] où A1,...,An sont les affixes des points constituant la ligne. S'il y a plusieurs composantes, on les sépare avec la constante *jump*, par exemple [A1,A2,A3,jump,A4,A5,A6].
- On peut régler deux paramètres de la ligne :
  - Un rayon (>0) pour avoir des angles arrondis (arcs de cercle de rayon souhaité).
  - Un booléen qui indique si la ligne est ouverte ou fermée.
- Le style de ligne se définit dans les Attributs ainsi que l'épaisseur, le mode de remplissage éventuel et les couleurs. Il est possible également d'ajouter une flèche aux extrémités.
- Un certain nombre d'éléments graphiques sont définis à partir d'une liste de points (comme les courbes, les équations différentielles...). Il est possible de récupérer cette liste de points avec la fonction *Get* (p. 41), par exemple si vous avez créé une spline appelée *S*1, vous pouvez récupérer tous les points de cette courbe et les mettre par exemple dans une variable *A* avec l'instruction : A := Get(S1).
- Il est possible de déterminer l'intersection de deux lignes polygonales avec l'opération InterL. Par exemple, l'exécution de Get(Courbe(t+i*t^2)) InterL Get(Droite(0,1+i)) renvoie :
  [0,0.999368819693+0.999368819693*i].
- Commande graphique correspondante : *Ligne* (p. 84) (utilisable dans un élément de type Utilisateur).

11) Path ou Chemin

Pour tracer un chemin (liste de points) ouvert ou fermé.

- Raccourci : *Ctrl+H*
- On donne un nom.
- On donne une commande de la forme d'une liste qui est une succession de points (affixes) et d'instructions indiquant à quoi correspondent ces points, ces instructions sont :
  - **line** : relie les points par une ligne polygonale,
  - **linearc** : relie les points par une ligne polygonale mais les angles sont arrondis par un arc de cercle, la valeur précédent la commande linearc est interprétée comme le rayon de ces arcs.
  - **arc** : dessine un arc de cercle, ce qui nécessite quatre arguments : 3 points et le rayon, plus éventuellement un cinquième argument : le sens (+/- 1), le sens par défaut est 1 (sens trigonométrique).
  - **ellipticArc** : dessine un arc d'ellipse, ce qui nécessite cinq arguments : 3 points, le rayonX, le rayonY, plus éventuellement un sixième argument : le sens (+/- 1), le sens par défaut est 1 (sens trigonométrique), plus éventuellement un septième argument : l'inclinaison en degrés du grand axe par rapport à l'horizontale.
  - **curve** : relie les points par une spline cubique naturelle.
  - **bezier** : relie le premier et le quatrième point par une courbe de Bézier (les deuxième et troisième points sont les points de contrôle).
  - **circle** : desine un cercle, ce qui nécessite deux arguments : un point et le centre, ou bien trois arguments qui sont trois points du cercle.
  - **ellipse** : dessine une ellipse, les arguments sont : un point, le centre, rayon rX, rayon rY, inclinaison du grand axe en degrés (facultatif).
  - **move** : indique un déplacement sans tracé.
  - **closepath** : ferme la composante en cours.

  Par convention, le premier argument du tronçon numéro n+1 est le dernier point du tronçon numéro n.
- Exemple(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])
- Commande graphique correspondante : *Path* (p. 84) (utilisable dans un élément de type Utilisateur).

12) Ellipse

Pour tracer une ellipse définie par son centre et ses deux rayons rx, ry et son inclinaison par rapport à l'horizontale.

- Raccourci : *Ctrl+C*
- On donne un nom.
- On donne une commande de la forme : [A, rx, ry] ou [A, rx, ry, inclinaison en degrés] : où A est l'affixe du centre, rx, ry les rayons. Par défaut l'inclinaison est par apport à l'horizontale est nulle.
- Si le repère n'est pas orthonormé l'ellipse sera déformée. Le repère est orthonormé lorsque les variables *Xscale* et *Yscale* sont égales : voir l'option Paramètres/Fenêtre du menu. Pour avoir une ellipse non déformée lorsque le repère n'est pas orthonormé, utiliser la macro *Rellipse()*.
- Une ellipse ne peut pas être fléchée (il faut utiliser pour cela un arc d'ellipse).
- Commande graphique correspondante : *Ellipse* (p. 81) (utilisable dans un élément de type Utilisateur).

13) Arc d'ellipse

Pour tracer un arc d'ellipse défini par trois points B, A, C (qui définissent un angle orienté), deux rayons rx, ry, et un sens.

- Raccourci : *Alt+Maj+A*
- On donne un nom.
- On donne une commande de la forme : [B, A, C, rx, ry] : où A est l'affixe du centre, le point de départ de l'arc est sur la demi-droite $[A, B)$, le dernier point sur la demi-droite $[A, C)$. Le booléen "sens trigo" permet de spécifier dans quel sens on tourne.
- Si le repère n'est pas orthonormé l'arc sera déformé. Le repère est orthonormé lorsque les variables *Xscale* et *Yscale* sont égales : voir l'option Paramètres/Fenêtre du menu. Pour avoir un arc non déformé lorsque le repère n'est pas orthonormé, utiliser la macro *RellipticArc()*.
- Un arc peut être fléché.
- Commande graphique correspondante : *EllipticArc* (p. 82) (utilisable dans un élément de type Utilisateur).

14) Label

Pour afficher du texte dans le graphique.

- Raccourci : *Alt+L*
- On donne un nom.
- On choisit un affixe (point de référence du label).
- On entre le texte (sans le délimiter par le caractère ")
- On définit le style de label (variable *LabelStyle*) dans les Attributs, ainsi que la taille (variable *LabelSize*) et l'orientation (*LabelAngle*).
- Les labels peuvent contenir des formules mathématiques et des macros de TEX, elles seront compilées par TEX dans les exportations sauf : en eps, pdf et svg (voir la rubrique *Exportations* (p. 18)).
- Commande graphique correspondante : *Label* (p. 84) (utilisable dans un élément de type Utilisateur).

15) Utilisateur

Cet élément permet à l'utilisateur de créer son propre élément graphique, celui-ci sera considéré comme une seule entité.

- Raccourci : *Ctrl+U*
- On donne un nom.
- On entre une commande. Celle-ci peut utiliser des commandes graphiques (droites, courbes...) ou des macros graphiques (ce sont des macros qui ont un effet graphique) comme celles qui sont dans le fichier TeXgraph.mac.
- Exemples :
  - Voici la commande d'un élément graphique Utilisateur :

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

    celle-ci trace la courbe de la fonction sinus avec les paramètres courants, on règle la variable globale Arrows à 2 (nombre de flèches), puis on trace un morceau de la tangente[2] à la courbe de sinus en $\pi/3$, de longueur 2 (unités graphiques).
  - Autre exemple :

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

    on trace une famille de courbes cartésiennes : $t \mapsto t^m$ pour m variant dans la liste $[-1, -0.25, 0.5, 2]$, pour chaque valeur de m on change également la couleur du tracé.
- Commande correspondante : *NewGraph* (p. 47).

2. tangente est une macro graphique du fichier TeXgraph.mac.

Chapitre III

Exportation des graphiques

Les graphiques créés avec TeXgraph peuvent être sauvegardés sous forme de fichiers sources (*.teg) et/ou exportés sous formes de fichiers destinés à être inclus dans un document (La)TeX. Il faut faire simplement attention à ce que (La)TeX soit en mesure de trouver ces fichiers au moment de la compilation, soit on les met dans le même répertoire que le document, soit on spécifie leur chemin d'accès dans le document. Il y a plusieurs formats d'exportations :

1) Format tex

- Ces fichiers sont exportés avec l'extension *.tex*, ils utilisent les macros des packages : xcolor (couleurs), epic et eepic (tracés de lignes) et éventuellement rotating (rotation de labels, celle-ci ne sera visible que dans la version postscript du document). Ces packages sont assez pauvres en capacités graphiques : pas de remplissage solides, pas de transparence, ..., ce qui fait que cet export est plutôt réservé aux graphiques ultra-basiques. Pour les graphiques plus élaborés, on préférera les formats pgf/tkz ou pstricks ou eps ou pdf.
- Exemple (minimal) :

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

- Compilations possibles :
  - latex
  - LaTex + dvips
  - latex + dvips + ps2pdf
  - latex + dvipdfmx (ou dvipdfm)

2) Format pst

- Ces fichiers sont exportés avec l'extension *.pst*, ils utilisent les macros du paquet pstricks (version 1.27 minimum).
- Exemple (minimal) :

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

- Compilations possibles :
  - LaTex + dvips
  - latex + dvips + ps2pdf

3) Format pgf

- Ces fichiers sont exportés avec l'extension *.pgf*, ils utilisent les macros du paquet pgf (version 2.0 minimum).
- Exemple (minimal) :

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

- Compilations possibles :
  - pdflatex
  - LaTex + dvips
  - latex + dvips + ps2pdf
  - latex + dvipdfmx (ou dvipdfm), à condition de rajouter :

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

    avant la déclaration du paquet pgf.

4) Format tkz

- Ces fichiers sont exportés avec l'extension *.tkz*, ils utilisent les macros du paquet pgf (version 2 minimum) mais dans un environnement *tikzpicture*, permettant ainsi l'ajout de macros propres à tikz.
- Exemple (minimal) :

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

- Compilations possibles :
  - pdflatex
  - LaTex + dvips
  - latex + dvips + ps2pdf
  - latex + dvipdfm (ou dvipdfmx), à condition de rajouter :

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

    avant la déclaration du paquet tikz.

5) Format eps

- Ces fichiers sont exportés avec l'extension *.eps*, ils utilisent le langage postscript. Dans ce format les labels ne seront pas compilés par TeX, donc s'ils contiennent des formules mathématiques ou des macros de TeX, celles-ci seront affichées mais non interprétées.
- Exemple (minimal) :

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

- Compilations possibles :
  - LaTex + dvips
  - latex + dvips + ps2pdf
  - latex + dvipdfmx (ou dvipdfm), à condition que votre installation soit configurée pour que dvipdfmx puisse convertir à la volée (avec epstopdf) l'image eps en image pdf.

6) Format psf (eps+psfrag)

- Ces fichiers sont exportés avec l'extension *.psf*. Il y a en réalité deux fichiers générés, un fichier eps et un fichier psf. Le premier contient la version postscript du graphique sans les labels, et le second contient les labels que le paquet psfrag replacera dans le graphique après leur compilation par (La)TeX. Dans ce format les formules mathématiques ou les macros de TeX seront compilées. Le fichier psf contient dans sa dernière ligne, l'instruction :

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

- Exemple (minimal) :

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

- Compilations possibles :
  - LaTex + dvips
  - latex + dvips + ps2pdf

Remarque : dans ce format, certains labels utilisent des macros de pstricks.

7) Format pdf

- Ces fichiers sont exportés avec l'extension *.pdf*. Il y a en réalité deux fichiers générés : TeXgraph crée un fichier eps puis appelle un convertisseur eps vers pdf, celui-ci est (par défaut) le programme *epstopdf*. Il est possible de modifier ce dernier en éditant le fichier de macros TeXgraph.mac et en modifiant la macro appelée *pdfprog*. Dans ce format les labels ne seront pas compilés par TeX, donc s'ils contiennent des formules mathématiques ou des macros de TeX, celles-ci seront affichées mais non interprétées.
- Exemple (minimal) :

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

- Compilations possibles :
  - pdflatex

8) Formats compilés

8.1 Format epsc

Lors de cet export, le programme demande un nom pour le fichier créé au format eps, appelons-le *Toto.eps*. Le graphique et alors exporté au format pstricks dans un fichier appelé *file.pst* qui se trouve dans le répertoire temporaire de TeXgraph, puis on lance le script `./CompileEps.sh` sous linux et `CompileEps.bat` sous windows, avec comme argument le nom du fichier `Toto`.

Contenu du script sous linux (similaire sous windows) :

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

Ce script lance la compilation du fichier *CompileEps.tex* suivant :

```
\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}
```

et sa conversion en image eps grâce au programme *dvips*. Bien entendu, ce fichier peut être modifié, il faut pour cela effacer la copie qui se trouve dans le dossier temporaire (`$HOME/.TeXgraph` sous linux et `c:\tmp` sous windows), et modifier l'original qui se trouve dans le dossier TeXgraph.

8.2 Format pdfc

Lors de cet export, le programme demande un nom pour le fichier créé au format pdf, appelons-le *Toto.pdf*. Le graphique est alors exporté au format pgf dans un fichier appelé *file.pgf* qui se trouve dans le répertoire temporaire de TeXgraph, puis on lance le script `CompilePdf.sh` sous linux et `CompilePdf.bat` sous windows, avec comme arguments la valeur 1 suivie du nom du fichier `Toto`.

Contenu du script sous linux (similaire sous windows) :

```
#!/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
```

Ce script crée le fichier *CompilePdf.tex*, que l'on peut lire dans le script et qui est créé dans le dossier temporaire, lance sa compilation par pdflatex et donne finalement l'image attendue. La valeur 1 signifie qu'il n'y a qu'une seule image à créer (c'est le même script qui est utilisé pour créer des animations).

9) Format svg

C'est un format vectoriel à destination du web, le fichier exporté est un fichier texte xml que l'on peut ensuite inclure dans une page html comme ceci par exemple :

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

Attention ! Tous les lecteurs d'html ne sont pas forcément capables d'afficher du svg en natif. Pour être tranquille, prenez plutôt firefox !

10) Récapitulatif

Export	paquet(s)	Compilation(s)	code	Labels TEX interprétés
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	aucun	format non reconnu	xml	

11) Exporter dans le presse-papier

Il y a un bouton dans la barre d'outils permettant de copier le graphique en cours dans le presse-papier. Le graphique est copié en tant que texte comme dans un fichier, il est possible de copier le graphique aux formats :

- **tex**, **pgf**, **tkz**, **pst** : on peut ensuite coller directement le graphique dans un document (La)TeX sans avoir à charger de fichier avec la macro *input*.
- **teg** : c'est le format source pour TeXgraph.
- **src4latex** : c'est le format source pour TeXgraph mais dans un environnement afin d'être inclus directement dans un document LATEX. Ce format est décrit dans *cette section* (p. 151).
- **texsrc** : c'est le source écrit en couleurs dans le langage TEX, cela permet d'afficher des exemples colorisés dans des documents LATEX comme celui-ci.

12) L'apercu

Cliquer sur ce bouton (en forme d'oeil) provoque l'exécution de la macro *Apercu* du fichier *interface.mac*, la commande qui définit cette macro est :

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

Le graphique en cours est donc exporté au format pgf dans le fichier *file.pgf*, dans le répertoire temporaire de TeXgraph, puis on lance la compilation du fichier *apercu.tex* avec pdflatex, et enfin on ouvre le fichier créé : *apercu.pdf* dans le lecteur pdf (celui-ci est défini dans le fichier de configuration, option Paramètres/Fichier de configuration). Le contenu du fichier *apercu.tex* est :

```
\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}
```

Bien entendu, ce fichier peut être modifié, il faut pour cela effacer la copie qui se trouve dans le dossier temporaire (`$HOME/.TeXgraph` sous linux et `c:\tmp` sous windows), et modifier l'original qui se trouve dans le dossier TeXgraph.

13) Export personnalisé

Il est possible par le biais de la commande *MyExport* (ou la commande *draw* qui est un alias) de créer de nouveaux éléments graphiques avec un export personnalisé différent de l'export prévu par défaut dans TeXgraph.

- **MyExport(<"nom">, <paramètre 1>, ..., <paramètre n>)**
- Description: cette commande s'utilise comme une commande graphique. L'utilisateur choisit un *<"nom">* et doit créer deux macros :
  - la première dont le nom doit être la concaténation du mot *Draw* et du *<"nom">*, cette macro fait le dessin,
  - la deuxième dont le nom doit être la concaténation du mot *Export* et du *<"nom">*, cette macro fait l'export en écrivant dans le fichier d'exportation avec la commande *WriteFile* (p. 55).

  Lors de l'évaluation graphique, la commande *MyExport* appelle la macro de dessin "*Draw*"+"*nom*" en lui passant les différents paramètres : *<paramètre 1>*, ..., *<paramètre n>*.
  Lors d'un export, la commande *MyExport* appelle la macro d'exportation "*Export*"+"*nom*" en lui passant les différents paramètres : *<paramètre 1>*, ..., *<paramètre n>*. Si cette macro renvoie la valeur 0 alors TeXgraph procède à l'export « classique ».
- Exemple(s): exportation des courbes cartésiennes en pstricks en utilisant la macro `\psplot`. Choisissons le nom `pstcartesian`, on écrit alors la macro de dessin *Drawpstcartesian(f(x), [options])* avec les options :
  - clip := ⟨ 0/1 ⟩ : pour faire un clip ou non avec la fenêtre définie par l'option *clipwin* (0 par défaut),
  - clipwin := ⟨ [xmin+i*ymin, xmax+i*ymax] ⟩ : définit la fenêtre de clipping, fenêtre graphique par défaut,
  - x := ⟨ [xmin, xmax] ⟩ : intervalle de tracé de la fonction, [tMin, tMax] par défaut.

  Ces options doivent être des *variables globales*.

```
{Drawpstcartesian(f(x),[options])}
[SaveAttr(), clip:=0, clipwin:=[Xmin+i*Ymin, Xmax+i*Ymax], x:=[tMin,tMax],
$aux:=%2, {Evaluation des options}
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() ]
```

On écrit ensuite la macro d'exportation : *Exportpstcartesian(f(x), [options])*

```
{Exportpstcartesian(expression,[options])}
if ExportMode=pst then {on teste le mode d'exportation}
 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 signifie export normal}
fi
```

La macro *cvfunction* renvoie la fonction au format pstricks sous forme d'une chaîne :

```
{cvfunction( chaine ): conversion vers la syntaxe de pstricks}
[$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]
```

Si on crée ensuite un élément graphique avec la commande : MyExport("pstcartesian", x^2, [x :=[-2,2], clip=1]), alors l'export pstricks donnera le fichier :

```
\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 : cet exemple est incomplet car il ne traite pas le problème de l'exportation des attributs : couleurs, épaisseur, style de ligne, ...

Chapitre IV

Le langage de TeXgraph

1) Les commandes de TeXgraph

Les commandes sont en réalité des fonctions au sens mathématique du terme. Celles-ci renvoient un résultat qui peut être une **liste de nombres complexes et/ou de chaînes de caractères** ou bien *Nil* .

Certaines commandes permettent un minimum de programmation : affectation d'une valeur à une variable, et structures de contrôles (alternative, boucles).

1.1 Syntaxe générale

- La syntaxe générale d'une commande de TeXgraph est : [argument1, ..., argumentN], lorsqu'il y a un seul argument, les crochets ne sont pas obligatoires (ceux-ci représentent la fonction *Liste* (p. 44)). Chaque argument est une expression mathématique.
- L'exécution de la commande consiste à *évaluer chaque argument* et à renvoyer la *liste des résultats* qui sont différents de *Nil*.
- Exemple(s):
  - [2,1+i,sqrt(-2),"toto",1/2] renvoie la liste : [2,1+i,"toto",0.5].
  - Seq(k^2,k,1,5) renvoie la liste : [1,4,9,16,25].
  - Droite(0,1+i) renvoie la valeur *Nil*, mais la fonction *Droite* (p. 81) a un effet graphique si on l'utilise dans un élément graphique *Utilisateur*.
  - Supposons que l'on ait défini 3 variables globales : A, B et C, alors la commande : [C, C+i*(B-A)] renvoie la valeur de C suivie de la valeur $C + i(B - A)$ cette expression peut être la commande pour définir la perpendiculaire à (AB) passant par C.
  - Supposons que l'on veuille construire un triangle (ABC) avec ses trois médianes comme un seul élément graphique, alors
    - on choisit *Eléments Graphiques/Créer/Utilisateur*,
    - on choisit un nom pour l'objet,
    - on saisit la commande :

      [Ligne([A,B,C],1), Droite(A,(B+C)/2), Droite(B,(A+C)/2), Droite(C,(A+B)/2)]

      Les fonctions *Ligne* (p. 84) et *Droite* (p. 81) renvoient la valeur *Nil* mais elles ont un effet graphique dans le contexte *Utilisateur*,
    - il ne reste plus qu'à créer les trois variables A, B, C (si ce n'est déjà fait). Bien sûr on peut aussi créer séparément la ligne polygonale et les trois droites.
- Les calculs sur les réels strictement positifs se font en principe dans l'intervalle $[10^{-324}, 10^{308}]$.
- TeXgraph est sensible à la casse, c'est à dire qu'il fait la distinction entre majuscules et minuscules.
- Chaque objet de TeXgraph est identifié à l'aide d'un *identificateur* (ou nom), celui-ci doit respecter les règles suivantes :
  - Commencer par une lettre.
  - Contenir au plus 35 caractères.
  - Chaque caractère doit être : une lettre, ou un chiffre, une quote (apostrophe) ou un soulignet.

1.2 Structures de contrôles

Afin de simplifier la saisie, les structures suivantes ont été introduites :

- l'alternative : *if then else fi*,
- la boucle conditionnelle : *while do od*,
- la boucle répétitive : *repeat until od*,
- et la boucle itérative : *for do od*.

Signalons aussi que :

- la commande *Set* (p. 51) (affectation) peut être remplacée par :=, par exemple, on peut écrire x :=2 à la place de Set(x,2). La commande *Set* (p. 51) fait une évaluation alphanumérique de son premier argument, ce qui signifie par exemple que si k est une variable contenant la valeur 2, alors la commande Set(["x",k], 5) sera comprise comme : Set(x2,5). Ceci est valable avec le symbole de l'affection : ["x",k] := 5.
- si x est une variable contenant une liste de complexes, la commande Copy(x, n, 1) qui renvoie la valeur du n-ième élément de la liste x, peut être remplacée par : x[n], plus généralement la syntaxe est : x[départ, nombre] avec la convention que si nombre=0 alors on va jusqu'à la fin de la liste, et si départ=−1 alors on part de la fin de liste et on remonte.
  NB : l'instruction x[n] :=1 ne changera pas le n-ième élément de la liste x, car x[n] est une valeur numérique ! C'est la macro *replace* qui permet de modifier les éléments d'une liste : replace(x, n, 1) remplacera le n-ième élément de la liste x (qui doit être une variable) par la valeur 1, la valeur de remplacement peut être également une liste.

L'alternative

C'est l'équivalent de la commande *Si* (p. 52). C'est une fonction qui renvoie la valeur *Nil*.

- **if <condition1> then <instructions> elif <condition2> then ... else <instructions> fi**
- Description: <condition> est une expression booléenne, c'est à dire qui vaut 0 (pour false) ou 1 (pour true), elif est la contraction de else if, ce qui permet une cascade de tests. Les instructions sont séparées par une virgule.
- Exemple(s): définition d'une fonction de t, par morceaux :

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

  pour tracer une telle fonction il est préférable de créer une macro qui représente la fonction, on peut par exemple créer une macro appelée *f* et définie par la commande if %1<=0 then 1-%1 elif %1<pi/2 then cos(%1) else %1^2 fi, le caractère %1 représente le premier paramètre de la macro. On peut ensuite créer un élément graphique Courbe en lui donnant un nom et le paramétrage suivant : t+i*f(t) ou t+i*\f(t), dans cette deuxième version, $f(t)$ est directement remplacée par son expression.

La boucle conditionnelle

C'est une version de la commande *Loop* (p. 44).

- **while <condition> do <instructions> od**
- Description: <condition> est une expression booléenne, c'est à dire qui vaut 0 (pour false) ou 1 (pour true). Les instructions sont séparées par une virgule.
- Exemple(s): Liste des cubes inférieurs à 1000 :

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

  l'exécution de cette commande (dans la ligne de commande en bas de la fenêtre) donne : [0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]. La première instruction de la boucle (x) renvoie la valeur de x, la deuxième (*Inc* (p. 43)) ajoute 1 à la variable k et renvoie *Nil*, la troisième (*:=*) affecte le cube de k à la variable x et renvoie la valeur *Nil*.

La boucle itérative

C'est une version des commandes *Seq* (p. 51) et *Map* (p. 45). Il y a deux syntaxes :

- **for <variable> in <liste valeurs> [step <pas> ou by/By <paquets de>] do <instructions> od**
- Description: Pour chaque valeur de la variable prise dans la liste, les instructions sont exécutées. Le *pas* est de 1 par défaut ce qui signifie que les valeurs de la liste sont parcourues en 1 en 1. L'option **by** (ou **By**) permet de lire les valeurs par paquets en traitant le cas de la constante *jump* : avec l'option *by* la structure renvoie un *jump* lorsqu'il est rencontré dans la liste, avec l'option **By** le *jump* n'est pas renvoyé. Lorsque le dernier paquet n'est pas complet, il n'est pas traité. Pour parcourir une liste par composante (deux composantes sont séparées par un *jump*), on utilise *by jump* ou *By jump*. Par exemple : for z in [1,2,jump,3,4,5] by jump do sum(z) od renvoie [3,jump,12], alors que for z in [1,2,jump,3,4,5] By jump do sum(z) od renvoie [3,12].
- **for <variable> from <valeur initiale> to <valeur finale> [step <pas> ou by/By <paquets de>] do <instructions> od**
- Description: Pour chaque valeur de la variable allant de la valeur initiale à la valeur finale, les instructions sont exécutées. La valeur est incrémentée du pas (1 par défaut), celui-ci peut-être négatif et réel non entier. L'option **by/By** est identique à ci-dessus.

NB : on ne peut pas utiliser by et step en même temps.
NB : lors du parcours d'une liste par composante (*by/By jump*), la constante **sep** contient la valeur du *jump* qui termine la composante en cours. Cette valeur est en fait un complexe particulier, c'est uniquement la partie réelle qui lui donne le statut de *jump* (1E308), la partie imaginaire quant à elle, peut être utilisée pour stocker une information numérique.

- Exemple(s):
  - La commande for m in [-1,-0.25,0.5,2] do Color :=4*m, Courbe(t+i*t^m) od utilisée dans un élément graphique Utilisateur permet de tracer la famille de courbes cartésiennes : $t \mapsto t^m$ pour m variant dans la liste $[-1, -0.25, 0.5, 2]$, pour chaque valeur de m on change également la couleur du tracé.
  - La commande for k from -2*pi to 2*pi step pi/2 do Droite(1,0,k) od utilisée dans un élément graphique Utilisateur va permettre de tracer les droites d'équations $x = -2\pi$, $x = -2\pi + \pi/2$, ..., $x = 2\pi$.
  - Parcourt par paquet avec l'option **by** ou **By** : la commande for z from 1 to 7 by 2 do z, jump od renvoie : [1, 2, jump, 3,4, jump, 5, 6, jump]. Dans cet exemple, la variable z prend successivement les valeurs [1, 2], [3, 4], et [5, 6], le dernier paquet n'étant pas complet, il n'est pas traité.
  - Parcourt avec condition : la commande for k in [1, 8, 4, 3, 2, 6, 5, 7] by 3 andif min(k)<=4 do k[1]+k[3] odfi renvoie [5,9].

2) Chaînes de caractères

2.1 L'évaluation alphanumérique

Pour les besoins de certaines fonctions, certains arguments sont interprétés sous forme alphanumérique (chaînes de caractères) et non pas sous forme numérique.

Si l'argument est une liste, alors **chaque élément de la liste est interprété sous forme de chaîne** et les différentes chaînes obtenues sont **concaténées**. Lors de cette interprétation il y a plusieurs cas de figure, TeXgraph peut rencontrer :

- Une chaîne : celle-ci doit être délimitée par les caractères " et ", si la chaîne doit contenir le caractère ", alors celui-ci doit être doublé : "".
- Une variable ou une constante : le contenu de celle-ci est renvoyé sous forme d'une chaîne.
- Une fonction renvoyant une chaîne.
- Une expression renvoyant une chaîne :
  - La macro **chaine()** : la commande définissant cette macro prédéfinie dans *interface.mac*, est un message. Cette macro est utilisée pour mémoriser les chaînes lors des saisies par la commande *Input* (p. 43). TeXgraph remplace *chaine()* par la chaîne correspondante.
  - Les commandes : *Map* (p. 45), *Si* (p. 52), *Loop* (p. 44), *Seq* (p. 51) : ces commandes fonctionnent comme dans leur contexte habituel à la différence près que les résultats ne sont pas évalués numériquement mais sous forme d'une chaîne. Par exemple : Map(["(", Re(z), "/", Im(z), ")"], z, [1+i,2,3-i]) renverra la chaîne : (1/1)(2/0)(3/-1), alors qu'une évaluation hors de ce contexte donnerait : ["(",1,"/",1,")","(",2,"/",0,")","(",3,"/",-1,")"]].
    Remarque : comme les structures *for*, *if*, *repeat* et *while* font appel à ces commandes, il est possible d'utiliser ces structures dans des arguments de type chaîne : Exemple la commande : Message(for z in [1+i,2,3-i] do if Im(z)>0 then "(", Re(z),",", Im(z),")" fi od) affichera (1,1).
  - Une macro-chaîne, c'est à dire une *macro renvoyant une chaîne* (p. 27).
- sinon : TeXgraph évalue numériquement l'expression et le résultat est transformé en chaîne de caractères.

Exemple(s): supposons que la macro *chaine()* soit définie avec la commande "toto", et que l'on ait défini une variable globale *A* égale à 6, alors la liste suivante :

["Notre ami ", UpperCase(chaine()), " a ", A*A, " dents"]

donnera la chaîne : Notre ami TOTO a 36 dents. Par contre, si la variable *A* n'a pas été définie, alors la chaîne obtenue sera Notre ami TOTO a dents, car la valeur de *A* est *Nil*.

2.2 Mémoriser une chaîne de caractères

Depuis la version 1.97, les variables de TeXgraph peuvent stocker des chaînes de caractères. Mais on peut aussi utiliser une macro pour jouer ce rôle de variable.

Pour créer une macro-chaîne (cette partie est laissée pour compatibilité ascendante) :

- **SetStr(<nom>, <expression> [, évaluer])**
- Description: crée la macro appelée <*nom*> et dont la commande est définie par l'<*expression*>, si <*évaluer*> vaut 1 (valeur par défaut) alors l'expression est évaluée sous forme de chaîne, sinon l'<*expression*> est copiée tel quel dans le corps de la macro. L'argument <*nom*> est évalué alphanumériquement.
- Exemple(s):
  - la commande SetStr(test, sqrt(4)) va créer une macro du nom de *test* et dont le contenu est la chaîne : "2",

– la commande SetStr(test, sqrt(4), 0) va créer une macro du nom de *test* et dont le contenu est la chaîne : sqrt(4) (sans guillemets).
– la commande SetStr(name, ["Mon nom est ", %1], 0) va créer une macro du nom de *name* et dont le contenu est la chaîne : ["Mon nom est ", %1], lors de l'exécution de Message(@name("toto")) on verra s'afficher Mon nom est toto. Ainsi une macro peut faire office de fonction à un ou plusieurs paramètres et renvoyant une chaîne.

Pour accéder au contenu d'une macro-chaîne (cette partie est laissée pour compatibilité ascendante) :

- **GetStr(<nom>)** ou **GetStr(<nom(arguments)>)**
- Description: évalue alphanumériquemment la macro appelée <*nom*> et renvoie la chaîne qui en résulte. Il existe un raccourci à cette commande, en accolant l'opérateur @ devant le <*nom*>.
- Exemple(s): Message(@nom) aura le même effet que Message(GetStr(nom)).

2.3 Commandes liées aux chaînes de caractères

- La commande **Concat(<argument 1>, <argument 2>, ..., <argument n>)** : chaque argument est interprété sous forme de chaîne, les différents résultats sont concaténés, et la commande renvoie la chaîne qui en résulte (voir la commande *Concat* (p. 37)).
- La commande **IsString(<arg>)** : renvoie 1 si <*arg*> est une une chaîne de caractères, 0 sinon. Lorsque <*arg*> est une liste, seul le premier argument est testé.
- **UpperCase(<expression>)** et **LowerCase(<expression>)** : renvoient <*expression*> respectivement en majuscules et minuscules.
- La commande **ScientificF(<réel> [, <nb décimales>])** : transforme le <*nombre*> au format scientifique et renvoie le résultat sous forme d'une chaîne de caractères.
- La commande **Str(<nom de macro>)** : représente le texte de la macro appelée <*nom de macro*> lorsque celle-ci n'est pas prédéfinie (sinon c'est la chaîne vide). L'argument <*nom de macro*> est lui-même interprété comme une chaîne de caractères.
- La commande **String(<expression>)** : si <*expression*> représente une variable, alors la commande renvoie le nom de la variable, sinon elle renvoie l'expression sous forme de chaîne de caractères.
- La commande **String2Teg(<expression>)** : cette fonction fait une évaluation alphanumérique de l'<*expression*> et renvoie le résultat sous forme de chaîne de caractères en doublant tous les caractères " rencontrés. La chaîne résultante est ainsi lisible par TeXgraph.
- **StrComp(<chaîne1>, <chaîne2>)** : renvoie 1 si les deux chaîne sont identiques, 0 sinon.
- **StrCopy(<chaîne>, <indice départ>, <quantité>)** : renvoie la chaîne résultant de l'extraction (fonctionne comme la commande *Copy* (p. 37)).
- **StrDel(<variable>, <indice départ>, <quantité>)** : modifie la <*variable*> en supprimant <*quantité*> caractères à partir de <*indice départ*> (fonctionne comme la commande *Del* (p. 38)). Si la <*variable*> contient une liste de chaînes, seule la première est modifiée. Si la <*variable*> ne contient pas de chaîne, la commande est sans effet.
- **StrEval(<"expression">)** : cette commande évalue l'<*expression*> (qui doit être une chaîne de caractères), et renvoie le résultat sous forme d'une chaîne de caractères.
- **StrLength(<chaîne>)** : renvoie le nombre de caractères de la chaîne.
- **StrPos(<motif>, <chaîne>)** : renvoie la position (entier) du premier motif dans la chaîne.
- **StrReplace(<chaîne>, <motif à remplacer>, <motif de remplacement>)** : renvoie la chaîne résultant du remplacement.
- **Args(<k>)** : s'utilise à l'intérieur d'une macro, elle évalue alphanumériquement l'argument numéro k, et renvoie la chaîne résultante. S'il n'y a pas d'argument, alors c'est la liste de tous les arguments qui est traitée.
- **StrArgs(<k>)** : s'utilise à l'intérieur d'une macro, elle renvoie l'argument numéro k sous forme d'une chaîne. S'il n'y a pas d'argument, alors c'est la liste de tous les arguments qui est traitée.

2.4 Macros renvoyant une chaîne

Les macros suivantes sont définies dans le fichier *TeXgraph.mac*.

- **coord(<z> [, décimales])** : renvoie les coordonnées du point d'affixe <z> sous forme d'un couple (x, y) avec le nombre maximal <*décimales*> demandé (4 par défaut). Cette macro est destinée à être utilisée comme une chaîne dans des fonctions ou macros ayant comme argument une chaîne de caractères. Exemple : Label(z,@coord(z)).
- **engineerF(<x>)** : renvoie le réel <x> sous forme de chaîne au format ingénieur, c'est à dire au format $\pm m \times 10^n$ où m est dans l'intervalle $[1; 1000[$ et n un entier multiple de 3. Cette macro est destinée à être utilisée comme une chaîne dans des fonctions ou macros ayant comme argument une chaîne de caractères.

- **epsCoord(<z> [, décimales])** : renvoie les coordonnées du point d'affixe <*z*> sous forme x y (coordonnées pour le format eps) avec le nombre maximal <*décimales*> demandé (4 par défaut). Cette macro est destinée à être utilisée comme une chaîne dans des fonctions ou macros ayant comme argument une chaîne de caractères.
- **label(<expression>)** : l'expression est évaluée alphanumériquement et délimitée avec le symbole $ si la variable *dollar* a la valeur 1, la macro renvoie la chaîne qui en résulte. Par exemple : [dollar :=1, @label(2+2)] renvoie "4". Cette macro est utilisée par la macro *GradDroite* (p. 94).
- **svgCoord(<z> [, décimales])** : renvoie les coordonnées du point d'affixe <*z*> sous forme x y (coordonnées pour le format svg) avec le nombre maximal <*décimales*> demandé (4 par défaut). Cette macro est destinée à être utilisée comme une chaîne dans des fonctions ou macros ayant comme argument une chaîne de caractères. Cette macro tient compte de la matrice de transformation courante.
- **texCoord(<z> [, décimales])** : renvoie les coordonnées du point d'affixe <*z*> sous forme (x, y) (coordonnées pour le format tex) avec le nombre maximal <*décimales*> demandé (4 par défaut). Cette macro est destinée à être utilisée comme une chaîne dans des fonctions ou macros ayant comme argument une chaîne de caractères. Cette macro tient compte de la matrice de transformation courante.
- **ScriptExt()** : renvoie la chaîne `".bat"` sous windows et `".sh"` sinon (extension des fichiers scripts).
- **StrNum(<valeur numérique>)** : remplace le point décimal par une virgule si la variable prédéfinie *usecomma* vaut 1 et renvoie la chaîne résultante. Le nombre de décimales est déterminé par la variable *nbdeci*, et le format d'affichage est défini par la variable *numericFormat* (0 : format par défaut, 1 : format scientifique, 2 : format ingénieur). Exemple : [usecomma :=1, nbdeci :=10, Message(@StrNum(10000*sqrt(2)))] affiche : 14142,135623731.
  Exemple : [usecomma :=1, nbdeci :=10, numericFormat :=1, Message(@StrNum(10000*sqrt(2)))] affiche : 1,4142135624E4.
  Exemple : [usecomma :=1, nbdeci :=10, numericFormat :=2, Message(@StrNum(10000*sqrt(2)))] affiche : 14,1421356237E3.
  Cette macro est utilisée par la macro *GradDroite* (p. 94).

3) Variables et constantes

3.1 Les constantes prédéfinies

- Les constantes mathématiques : i, π, e.
- Le numéro de version de TeXgraph est contenu dans la constante appelée **version**.
- La constante **Windows** contient la valeur 0 ou 1 suivant votre système d'exploitation.
- La constante **GUI** contient la valeur 0 ou 1 indiquant si on est dans l'interface graphique de TeXgraph ou non.
- La constante de saut : *jump* . Cette constante est utilisée pour séparer les différentes composantes connexes d'une ligne polygonale. Signalons au passage que les lignes polygonales sont automatiquement "clippées" par TeXgraph avec le rectangle correspondant à la fenêtre courante.
- Exemple(s): la courbe d'équation $y = 1/x$ peut être construite à partir de la ligne polygonale définie par la commande : [Seq(t+i/t,t,-5,0,0.1), jump, Seq(t+i/t,t,0,5,0.1)].
- La constante *Nil*. C'est une constante sans valeur, elle peut être utilisée pour des comparaisons, par exemple pour savoir si une variable x contient une valeur : if x<>Nil then ....
- Des constantes qui sont des chaînes de caractères :
  - **InitialPath** : chemin d'accès au répertoire de TeXgraph (celui-ci contient les exécutables et les scripts).
  - **DocPath** : chemin d'accès au répertoire doc de TeXgraph, ce répertoire contient des docs au format pdf (dont TeXgraph.pdf).
    Exemple : la commande Exec("xpdf","TeXgraph.pdf",DocPath), ouvrira le fichier TeXgraph.pdf avec le programme xpdf.
  - **UserMacPath** : chemin d'accès au répertoire contenant les macros utilisateurs. Sous linux c'est le dossier `$HOME/TeXgraphMac` et sous windows il doit être créé par l'utilisateur et le chemin d'accès doit être dans la variable d'environnement *TeXgraphMac*. Lorsque l'utilisateur charge un fichier de macros (*.mac) ou un fichier modèle (*.mod), TeXgraph cherche dans le dossier courant, puis dans le dossier *UserMacPath* et enfin dans le sous-dossier macros du dossier contenu dans la chaine *InitialPath*.
  - **TmpPath** : chemin d'accès à un répertoire temporaire. C'est le dossier `$HOME/.TeXgraph` sous linux, et `c:\tmp` sous windows.
  - **JavaviewPath** : chemin d'accès au fichier *javaview.jar* si vous l'avez installé. Sa valeur est à définir dans le fichier de configuration : menu *Paramètres/Fichier de configuration*.
  - **LF** : provoque un passage à la ligne lors de l'affichage de la chaîne.
  - **Diese** : qui renvoie le caractère du même nom (utilisé comme délimiteur dans les sources TeXgraph).
  - **DirSep** : qui renvoie le caractère séparateur utilisé par le système dans les chemins d'accès aux fichiers.

- Les constantes d'exportation : **tex**, **teg**, **pst**, **pgf**, **eps**, **psf**, **tkz**, **pdf**, **epsc**, **pdfc**, **svg** et **bmp**, ce sont les valeurs possibles que peut prendre la constante **ExportMode** (qui est déterminée par TeXgraph au moment de l'exportation). À celles-ci s'ajoutent pour la 3D, les constantes d'exportation : **obj**, **geom**, **jvx**.
- Les constantes : **Xmin**, **Xmax**, **Ymin**, **Ymax** : elles déterminent la fenêtre graphique.**Xscale** et **Yscale** : représentent (en cm) l'échelle sur l'axe Ox pour la première, et l'échelle sur Oy pour l'autre. Ces constantes sont modifiables uniquement par le menu ou la fonction *Fenetre* (p. 41).
- Les constantes : **margeG**, **margeD**, **margeH**, **margeB** : elles déterminent les marges autour du graphique (en cm). Ces constantes sont modifiables uniquement par le menu ou la fonction *Marges* (p. 45).
- Les constantes **line**, **linearc**, **bezier**, **curve**, **arc**, **ellipticArc**, **ellipse**, **circle**, **closepath**, **move** : elles sont utilisées pour construire des chemins dans la commande *Path* (p. 84).
- Les constantes graphiques :
  - Les couleurs : les couleurs font l'objet d'un chapitre *spécifique* (p. 11).
  - Styles de trait :
    - **noline** [=-1],
    - **solid** [=0],
    - **dashed** [=1],
    - **dotted** [=2],
    - **userdash** [=3], ce style utilise la variable **DashPattern** qui définit le motif, celui-ci est une liste de longueurs exprimées en points, de la forme : *[longueur trait, longueur saut,longueur trait, longueur saut, ...]*. Par exemple DashPattern :=[2,3,0.1,3] donnera une succession de traits - points.
  - Terminaison des lignes :
    - **butt** : terminaison droite au dernier point (valeur par défaut),
    - **round** : terminaison avec un arrondi après le dernier point,
    - **square** : terminaison avec un carré après le dernier point,
  - Jointure des lignes :
    - **miter** : jointure en pointe, la variable *Miterlimit* (égale à 10) permet de gérer la longueur des pointes.
    - **round** : jointure arrondie (valeur par défaut),
    - **bevel** : jointure en pointe coupée.
  - Épaisseur du trait (en **nombre entier de dixième de point** de TEX) :
    - **thinlines** [=2],
    - **thicklines** [=8],
    - **Thicklines** [=14], la variable *Width* permet également de régler l'épaisseur.
  - Styles de point (à la pstricks) :
    - **dot** [=0],
    - **dotcircle** [=1],
    - **square** [=2],
    - **square'** [=3] (carré plein),
    - **plus** [=4],
    - **times** [=5],
    - **asterisk** [=6],
    - **oplus** [=7],
    - **otimes** [=8],
    - **diamond** [=9],
    - **diamond'** [=10],
    - **triangle** [=11],
    - **triangle'** [=12],
    - **pentagon** [=13],
    - **pentagon'** [=14],
  - Styles de Label (par défaut le texte est centré horizontalement et verticalement) :
    - **left** : le point de référence est à gauche du texte,
    - **right** : le point de référence est à droite du texte,
    - **top** : le point de référence est en haut du texte,

  - **bottom** : le point de référence est en bas du texte,
  - **baseline** : le point de référence est la ligne de base du texte,
  - **framed** : le texte est encadré,
  - **special** : le texte est écrit tel quel dans le fichier exporté (il n'apparaît pas à l'écran). Cela permet d'écrire directement dans le fichier LaTeX ou pgf ou pstricks (et même eps).
  - **stacked** : le texte peut contenir des sauts de paragraphes.
    Exemple d'utilisation : LabelStyle := top+framed, le texte est centré horizontalement, le point de référence est en haut du texte et celui-ci est encadré.
- Styles de remplissage pour les polygones (les hachures sont calculées par TeXgraph pour la sortie LaTeX) :
  - **none** [=0] : pas de remplissage,
  - **full** [=1] : le polygone est rempli avec la couleur désignée par *FillColor*, ceci est sans effet avec l'export *tex*.
  - **bdiag** [=2] : hachures orientés SO -> NE (angle de 45 degrés),
  - **hvcross** [=3] : styles horizontal et vertical combinés,
  - **diagcross** [=4] : styles bdiag et fdiag combinés,
  - **fdiag** [=5] : hachures orientés NO -> SE (angle de 45 degrés),
  - **horizontal** [=6] : hachures horizontales,
  - **vertical** [=7] : hachures verticales.
- Taille des Labels :
  - **tiny**,
  - **scriptsize**,
  - **footnotesize**,
  - **small**,
  - **normalsize**,
  - **large**,
  - **Large**,
  - **LARGE**,
  - **huge**,
  - **Huge**.
- Relatives à la 3D :
  - **ortho** : type de projection,
  - **central** : type de projection,
  - **sep3D** : séparateur pour la commande *Build3D* (p. 139).

3.2 Les variables globales prédéfinies

Sont considérées comme prédéfinies : les variables ci-dessous, ainsi que toute variable globale contenue dans un fichier de macros chargé au démarrage du programme. Les variables prédéfinies n'apparaissent pas dans la fenêtre de TeXgraph, elles ne seront pas enregistrées avec le graphique.

Les variables globales suivantes correspondent aux différents "champs statiques" des éléments graphiques :

- **Arrows** : nombre de flèches, initialisée à 0,
- **AutoReCalc** : recalcul automatique des éléments graphiques, initialisée à 1 (pour *True*), elle peut également prendre la valeur 0 (pour *False*). Dans le cas ou sa valeur est nulle pour un élément graphique, seule la fonction **ReCalc()** (bouton **R**) peut forcer le recalcul de cet élément.
- **ForMinToMax** : variable contenant la valeur 0 ou 1, si sa valeur est 1 alors la variable *t* pour les courbes parcourt l'intervalle `[Xmin,Xmax]`, sinon c'est l'intervalle `[tMin,tMax]`.
- Variables relatives aux axes
  - **xylabelpos** : position des labels par rapport aux axes, initialisée à *bottom+left* (à gauche de l'axe Oy et en bas de l'axe Ox).
  - **xylabelsep** : distance (en cm) entre les labels et l'extrémité des graduations, initialisée à *0.1 cm*.
  - **xyticks** : longueur (en cm) des graduations sur les axes, initialisée à *0.2 cm*.
- **Color** : couleur, initialisée à *black*,
- **DashPattern** : définit le motif de tracé des lignes dans le style **userdash**, cette variable est une liste de longueurs exprimées en points, de la forme : *[longueur trait, longueur saut,longueur trait, longueur saut, ...]*. Par exemple DashPattern :=[2,3,0.1,3] donnera une succession de traits - points.

- **DotStyle** : style de point, initialisée à *dot*,
- **DotAngle** : angle de rotation des points (en degrés), initialisée à 0
- **DotScale** : facteur d'échelle pour les points, initialisée à [1,1] (échelle sur Ox et sur Oy),
- **DotSize** : taille des points, initialisée à 2+2i, le contenu de cette variable est un complexe $x + iy$ où x est une taille exprimée en points et y un nombre positif, le diamètre des points est calculé avec la formule : x+y*(épaisseur de ligne).
- **Eofill** : initialisée à 0, cette variable peut prendre les valeurs 0 ou 1, la valeur 1 indique que le mode de remplissage suit la règle pair-impair (ou even-odd), et la valeur 0 indique le cas contraire. Le mode even-odd (*Eofill*=1) n'est pas toujours bien géré à l'écran dans la version GUI de TeXgraph, mais il ne pose pas de problème dans les exportations.
- **FillColor** : couleur du remplissage, initialisée à *white*,
- **FillOpacity** : gestion de l'opacité/transparence lorsque *FillStyle*=*full*, c'est une valeur entre 0 et 1 initialisée à 1, la valeur 1 signifie pas de transparence. La transparence n'est pas gérée à l'écran par TeXgraph, mais elle l'est dans les exports,
- **FillStyle** : style de remplissage, initialisée à *none*,
- **IsVisible** : valeur booléenne (0 ou 1), permettant de rendre l'élément graphique visible ou invisible (initialisée à 1).
- **LabelAngle** : orientation des labels par rapport à l'horizontale, c'est un angle en degrés initialisé à 0.
- **LabelSize** : taille des labels, initialisée à *small*,
- **LabelStyle** : style de label, initialisée à 0 (centré horizontalement et verticalement),
- **LineCap** : définit le type de terminaison des lignes, initialisée à *butt* par défaut.
- **LineJoin** : définit le type de jointure des lignes, initialisée à *round* par défaut.
- **LineStyle** : style de lignes, initialisée à *solid*. Dans la version actuelle sous windows, les traits apparaissent pleins à l'écran dès que l'épaisseur dépasse un pixel même si le style est en pointillés ou tirets. Ce problème ne se pose pas dans les exportations,
- **MiterLimit** : détermine la longueur des pointes lorsque *LineJoin* est égale à *miter* (jointures en pointe), initialisée à 10 par défaut.
- **NbPoints** : nombre de points (pour les courbes), initialisée à 50,
- **PenMode** : mode de dessin, 0=mode normal, 1=mode NotXor, lorsqu'on redessine un élément graphique créé en mode NotXor, il s'efface en restituant le fond, on peut alors modifier sa position et le redessiner. Cette technique permet de faire glisser des objets sans avoir à réafficher tous les autres (ce qui évite d'avoir une image qui saute). Cette variable est initialisée à 0,
- **StrokeOpacity** : gestion de l'opacité/transparence pour les traits lorsque *LineStyle* est différent de *noline*, c'est une valeur entre 0 et 1 initialisée à 1, la valeur 1 signifie pas de transparence. La transparence n'est pas gérée à l'écran par TeXgraph, mais elle l'est dans les exports,
- **TeXLabel** : valeur booléenne (0 ou 1) indiquant si les labels doivent être affichés sous forme d'images dans l'interface graphique après compilation par TeX. Cette variable est initialisée à 0.
- **tMax** : valeur maximale du paramètre t, initialisée à 5,
- **tMin** : valeur minimal du paramètre t, initialisée à -5,
- **Width** : épaisseur du trait, exprimée en **nombre entier de dixième de point** de TeX, elle est initialisée à *thinlines*.

La création d'un élément graphique n'entraîne pas la création d'une constante portant le même nom. Il est cependant toujours possible d'accéder à la liste des points composant un élément graphique avec la fonction *Get* (p. 41). Mais cela nécessite que l'élément graphique dont on utilise le nom soit **déjà créé**, sinon la fonction *Get* renverra la valeur *Nil*.

Variables relatives à la représentation en 3D :

- **theta** et **phi** : utilisées pour les calculs de projections des surfaces, elles sont initialisées respectivement à 30 et 60 degrés, la première représente la longitude et la deuxième la colatitude. Elles sont modifiables également par l'intermédiaire d'un bouton dans la barre d'outils.
- **AngleStep** : représente le pas angulaire (en radians) lorsque l'on fait tourner un objet 3D à l'aide des boutons représentant les flèches de direction. Celle-ci est initialisée à $\pi/36$ (5 degrés).

3.3 Déclaration des variables

Lorsque TeXgraph rencontre un nom dans une expression, il regarde s'il est suivi d'une parenthèse [ex : *toto(...*] :

- si c'est le cas : il teste s'il s'agit d'une fonction prédéfinie, sinon il considère que c'est une macro [1] (même si elle n'existe pas encore).

1. Une macro sans paramètre s'utilise quand même avec deux parenthèses : *toto()*.

- si ce n'est pas le cas : alors il teste **d'abord** s'il existe une variable **locale** qui porte ce nom, dans la négative, il teste s'il existe une variable **globale** qui porte ce nom, dans la négative, il **crée** une variable **locale**[2] portant ce nom [et initialisée à *Nil*].

Il n'est donc pas nécessaire de déclarer les variables locales, la première occurrence fait office de déclaration. Cependant, il se peut que l'on ait besoin qu'une variable *x1* [par exemple] soit locale alors qu'il y a déjà une variable globale qui porte le même nom, pour obliger TeXgraph à considérer *x1* comme une variable locale, il suffit de mettre le caractère $ devant son nom : *$x1* (il suffit en fait de le mettre uniquement devant la première occurrence).

3.4 Les variables globales

- Les variables globales se déclarent par l'intermédiaire du Menu ou du bouton *Nouv.* de la zone des variables globales (à droite de la fenêtre), elles portent un nom et sont définies à partir d'une commande. Elles seront enregistrées avec le graphique dans le fichier source.
- Lorsque l'on clique sur un point de la fenêtre avec le bouton droit de la souris, TeXgraph propose d'enregistrer l'affixe de ce point sous forme de variable globale, ce qui peut être utile pour placer des labels, ou pour créer une figure sans se préoccuper des coordonnées...
- Lorsque l'utilisateur modifie leur contenu (en double-cliquant sur le nom dans la zone variable globale, ou à partir de la ligne de commande en bas de la fenêtre), les éléments graphiques sont alors remis à jour automatiquement. On peut désactiver le recalcul automatique d'un élément graphique en décochant l'option adéquate dans les attributs.

3.5 Recalcul automatique

La création/modification d'une variable globale ou d'une macro entraîne automatiquement le recalcul de tout le graphique c'est à dire :

- de toutes les variables globales non prédéfinies,
- de toutes les macros non prédéfinies,
- de tous éléments graphiques qui sont en mode *Recalcul Automatique*.

Remarque : La modification de la fenêtre par le menu entraîne aussi le recalcul automatique.

3.6 Les variables des fichiers TeXgraph.mac et interface.mac

Ces fichiers sont chargés automatiquement lors du lancement du programme (ainsi que *color.mac* et *scene3d.mac*). Leur contenu est considéré comme prédéfini (c'est le savoir faire de base), il n'apparaît pas à l'écran, il n'est pas enregistré avec les graphiques, et il est présent en mémoire jusqu'à la fermeture du programme.

Voici la liste des principales variables (les variables qui servent d'options dans certaines macros ne sont pas citées ici) :

- **stock**, **stock1** à **stock5** (=*Nil*) : variables de stockage.
- **mm** (=Ent(7227/254)) : nombre entier de dixième de points (de TeX) correspondant à 1 millimètre. Utile pour l'épaisseur des lignes qui sont en nombre entiers de dixième de points, par exemple Width :=1.5*mm donnera une épaisseur de 1.5 mm.
- **backcolor** (=white) : contient la couleur du fond, elle est mise à jour par la macro *background* (p. 91), et elle est utilisée par certains exports.
- **deg** (=pi/180) : conversion degrés vers radians, par exemple : alpha :=40*deg.
- **rad** (=180/pi) : conversion radians vers degrés, par exemple : LabelAngle :=pi/16*rad.
- **tailleB** (=145+i*30) : longueur d'un bouton et hauteur en pixels.
- **DeltaB** (=32*i) : écart entre deux boutons + hauteur d'un bouton.
- **RefPoint** (=2+5*i) : point de référence pour le premier bouton.
- **NbBoutons** (=0) : compteur de boutons.
- **Xfact** (=1.1) et **Yfact** (=1.1) : variables utilisées lors des zooms (boutons + et - de la barre d'outils).
- **usecomma** (=0) : cette variable est utilisée par la macro *GradDroite* (p. 94), avec la valeur 1, le point est remplacé par une virgule dans les affichages numériques associés aux graduations. Le remplacement est fait par la macro *StrNum* (p. 29).
- **numericFormat** (=0) : cette variable est utilisée par la macro *StrNum* (p. 29). Elle indique si l'affichage numérique se fait au format par défaut (valeur 0), au format scientifique (valeur 1) ou au format ingénieur (valeur 2).
- **nbdeci** (=2) : nombre de décimales dans les affichages numériques, cette variable est utilisée par la macro *StrNum* (p. 29), elle-même utilisée par la macro *GradDroite* (p. 94).

2. Locale à l'expression en cours d'analyse, cette analyse transforme l'expression en arbre, lorsque cet arbre est détruit, les variables locales correspondantes sont également détruites.

- **maxGrad** (=100) : cette variable est utilisée par la macro *GradDroite* (p. 94), elle indique le nombre maximal de graduations.

Variables liées à la 3D :

- **Origin** (= [0,0]) : l'origine,
- **vecI** (= [1,0]) : premier vecteur de base,
- **vecJ** (= [i,0]) : deuxième vecteur de base,
- **vecK** (= [0,1]) : trosième vecteur de base,
- **Xinf** (= -5), **Xsup** (= 5), **Yinf** (= -5), **Ysup** (= 5), **Zinf** (= -5), **Zsup** (= 5) : fenêtre 3D

4) Les macros

Une macro est une fonction créée par l'utilisateur et qui renvoie un résultat (liste de complexes ou chaînes, ou bien *Nil*). TeXgraph distingue trois sortes de macros :

- celles qui sont chargées au lancement du programme : celles-ci sont considérées comme **prédéfinies** et n'apparaissent pas dans la liste des macros modifiables, on ne peut pas les supprimer et elles ne sont pas enregistrées non plus dans les fichiers sources **.teg*.
- celles qui sont chargées par le menu avec l'option *Fichier/ Charger des macros*, ou par l'instruction *InputMac* (p. 43) : celles-ci sont considérées comme **prédéfinies** et n'apparaissent pas dans la liste des macros modifiables, elles ne sont pas enregistrées dans les fichiers sources **.teg*, mais elles seront supprimées de la mémoire au prochain changement de fichier.
- celles qui sont créées pendant l'exécution du programme : celles-ci sont modifiables et sont enregistrées dans les fichiers sources **.teg*.

Un fichier de macros est un fichier texte **.mac* qui ne contient que des macros et éventuellement des variables globales. On peut créer/modifier un fichier de macros directement dans TeXgraph ou bien avec l'éditeur de son choix, à condition d'utiliser l'encodage UTF8.

4.1 Création d'une macro

- Une macro est définie par un nom et une commande. Une macro peut posséder des variables locales et des paramètres, ceux-ci se notent ainsi : *%1, %2, ...*, il n'est pas nécessaire de déclarer les paramètres.
- Afin que le texte de la macro ne soit pas enregistré sur une seule ligne dans le fichier **.teg*, il faut formater le texte en insérant des sauts de ligne [avec la touche *Entrée*] lors de la saisie[3], cela ne peut que faciliter la lisibilité. De plus il est possible de documenter une commande en insérant des commentaires, il y a deux méthodes pour cela, soit entre deux accolades : `{c'est un commentaire }`, soit une ligne de commentaires commençant par `//`.
- Exemple(s): voici la commande définissant une macro appelée *racine* qui donne la liste des racines n-ièmes d'un complexe :

```
{utilisation: racine(n,z), donne la liste des racines nièmes de 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
```

  - on teste si le premier paramètre (qui représente n) est un entier strictement positif, auquel cas on stocke dans une variable locale la racine n-ième du module de z (deuxième paramètre) puis on donne la liste des solutions (sinon la macro renvoie *Nil*).
  - l'exécution de [$a :=3, racine(a,i)] donne : [0.866025+0.5*i, -0.866025+0.5*i, -i].
- TeXgraph ne teste pas le nombre d'arguments, la valeur implicite des arguments manquants est *Nil*, s'il y en a trop, ceux qui sont en surplus sont ignorés.

4.2 Développement différé ou immédiat

- Comme une commande se présente sous la forme d'une chaîne de caractères, avant même de pouvoir exécuter la commande, TeXgraph doit analyser cette chaîne avant de la transformer en arbre. C'est lors de cette analyse qu'une macro peut être développée tout de suite ou non.

3. Ceci est également valable pour la commande des éléments graphiques *Utilisateurs*.

- Lors de **l'analyse** de [$a :=3, racine(a,i)] : TeXgraph construit l'arbre correspondant en conservant le mot *racine*, lorsqu'il évalue l'arbre, il fait une copie de l'expression de la macro *racine* en remplaçant le paramètre %1 par la variable a [4] et le paramètre %2 par i, puis évalue l'expression ainsi obtenue [5] et détruit la copie : **c'est le développement différé**.
- Lors de **l'analyse** de [$a :=3, \racine(a,i)] : TeXgraph remplace *racine* par l'expression de la macro en remplaçant le paramètre %1 par la variable a et le paramètre %2 par i, ce qui revient à analyser la commande :

```
[$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]
```

  c'est le développement immédiat. On remarquera que cette fois-ci il y a une seule variable a, ce qui fait que cette commande ne donnera pas le bon résultat (elle donne i). Par contre la commande [$b :=3, \racine(b,i)] donne le bon résultat ([0.866025403784+0.5*i,-0.866025403784+0.5*i,-i]). Le développement immédiat ne peut avoir lieu que si la macro existe déjà, sinon c'est un développement différé.
- Le développement immédiat est à proscrire lorsque la macro possède des variables locales et qu'il y a un risque d'homonymie avec les variables de l'expression appelante. Cependant il y a des cas ou celui-ci est plus intéressant que le développement différé, par exemple si on définit la macro appelée *f* par la commande %1*arctan(%1)/(1+%1^2) et si on crée l'élément graphique *Courbe/Paramétrée* avec l'expression t+i*\f(t), alors l'expression sera en réalité t+i*t*arctan(t)/(1+t^2) et comme cette expression va être évaluée un "grand nombre" de fois, ce sera plus rapide à l'exécution que l'expression t+i*f(t), car dans celle-ci (développement différé) la macro *f* sera appelée à chaque évaluation de l'expression.
  D'un autre côté, le développement immédiat permet aussi d'utiliser les macros comme des variables ou comme des **raccourcis**.
- Les macros peuvent être récursives.

4. Ce n'est pas la valeur de a qui remplace %1 mais l'adresse de a.
5. Dans cette expression il y a en fait deux variables a mais il n'y a pas d'ambiguïté car l'une est "branchée" sur les variables locales de la macro, et l'autre sur les variables locales de l'expression "appelante".

Chapitre V

Liste des commandes

Notations :
<argument> : signifie que l'argument est **obligatoire**.
[argument] : signifie que l'argument est **facultatif**.

1) Args

- **Args(<entier>)**.
- Description: cette fonction n'a d'effet que dans une macro, elle évalue et renvoie l'argument numéro <*entier*> avec lequel la macro a été appelée. Hors de ce contexte, elle renvoie la valeur *Nil*. Voir également la commande *StrArgs* (p. 53)
- Exemple(s): Voir la fonction *Nargs* (p. 46).

2) Assign

- **Assign(<expression>, <variable>, <valeur>)**.
- Description: cette fonction évalue <*valeur*> et l'affecte à la variable nommée <*variable*> dans <*expression*> [1]. La fonction *Assign* renvoie la valeur *Nil*. Cette fonction est utile dans l'écriture de macros prenant une expression comme paramètre et qui doivent l'évaluer.
- Exemple(s): voici une macro Bof qui prend une fonction f(t) en paramètre et qui calcule la liste $[f(0), f(1), ..., f(5)]$:

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

%1 représente le premier paramètre de la macro (c'est à dire f(t)), la boucle : pour k allant de 0 à 5 elle exécute la commande [Assign(%1, t, k), %1], celle-ci assigne la valeur de k à la variable *t* dans l'expression %1, puis évalue %1. L'exécution de Bof(t^2) donne : [0,1,4,9,16,25]. L'exécution de Bof(x^2) donne *Nil*.

3) Attributs

- **Attributs()**.
- Description: cette fonction ouvre la fenêtre permettant de modifier les attributs d'un élément graphique. Cette fonction renvoie la valeur 1 si l'utilisateur a choisi *OK*, elle renvoie la valeur 0 s'il a choisi *Cancel*. Si l'utilisateur a choisi *OK*, alors les variables globales correspondant aux attributs sont modifiées en conséquence.

4) Border

- **Border(<0/1>)**
- Description: cette fonction détermine si un cadre doit être dessiné ou non autour des marges du graphique dans les exportations. Lorsque la valeur de l'argument est nulle (valeur par défaut), le cadre n'est pas dessiné.
  Lorsque l'argument est vide, cette fonction renvoie l'état de la bordure à l'exportation (0 ou 1). Sinon, elle renvoie la valeur *Nil*.

1. C'est la première occurrence de <*variable*> dans <*expression*> qui est assignée, car toutes les occurrences pointent sur la même <case mémoire>, sauf éventuellement pour les macros après l'affectation des paramètres.

5) ChangeAttr

- **ChangeAttr(<element1>, ..., <elementN>)**
- Description: cette fonction permet de modifier les attributs des éléments graphiques <*element1*>, ..., <*elementN*>, en leur affectant la valeur des attributs en cours. Les arguments sont interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*.

6) Clip2D

- **Clip2D(<ligne polygonale>, <contour convexe> [, close(0/1)])**.
- Description: cette fonction clippe la <*ligne polygonale*> qui doit être une variable contenant une liste de complexes, avec le <*contour convexe*>, qui est lui aussi une liste de complexes. La fonction calcule la ligne polygonale qui en résulte modifie la variable <*ligne polygonale*> en conséquence, le dernier agument <*close*> (0 par défaut) permet de préciser si la <*ligne polygonale*> doit être refermée ou non. La fonction renvoie *Nil*.

7) CloseFile

- **CloseFile()** ou **CloseFile(<"fichier 1">, <"fichier 2">,...,<"fichier N">)**.
- Description: cette fonction permet de fermer les fichiers dont les noms (chaînes de caractères) sont cités en argument. Dans le cas où il n'y a pas d'arguments, c'est le dernier fichier ouvert qui sera refermé. C'est à ce moment que l'écriture physique dans le fichier se produit. Les fichiers doivent avoir été ouverts par la commande *OpenFile* (p. 48).

8) ComposeMatrix

- **ComposeMatrix(<[z1, z2, z3]>)**
- Description: cette fonction permet de composer la matrice courante (celle-ci affecte tous les éléments graphiques sauf les axes et les grilles dans la version actuelle) avec la matrice <*[z1, z2, z3]*>. Cette matrice représente l'expression analytique d'une application affine du plan, c'est une liste de trois complexes : $z1$ qui est l'affixe du vecteur de translation, $z2$ qui est l'affixe du premier vecteur colonne de la matrice de la partie linéaire dans la base (1,i), et $z3$ qui est l'affixe du deuxième vecteur colonne de la matrice de la partie linéaire. Par exemple, la matrice de l'identité s'écrit ainsi : [0,1,i] (c'est la matrice par défaut). (Voir aussi les commandes *GetMatrix* (p. 42), *SetMatrix* (p. 51), et *IdMatrix* (p. 43)).

9) Concat

- **Concat(<argument 1>, <argument 2>, ..., <argument n>)**.
- Description: cette commande interprète chaque argument sous forme de chaîne, concatène les différents résultats, et renvoie la chaîne qui en résulte.

10) Copy

- **Copy(<liste>, <index depart>, <nombre>)**.
- Description: cette fonction renvoie la liste constituée par les <*nombre*> éléments de la <*liste*> à partir de l'élément numéro <*depart*> [inclus]. Si <*nombre*> est nul, alors la fonction renvoie tous les éléments de la liste à partir de l'élément numéro <*départ*>.
  Si le numéro <*depart*> est négatif, alors la liste et parcourue de droite à gauche en partant du dernier, le dernier élément a l'index -1, l'avant-dernier a l'index -2 $\cdots$ etc. La fonction renvoie les <*nombre*> éléments de la liste (ou toute la liste si <*nombre*> est nul) en allant vers la gauche, mais la liste renvoyée est dans le même sens que la <*liste*>, et cette dernière n'est pas modifiée.
- Exemple(s):
  - Copy([1,2,3,4],2,2) renvoie [2,3].
  - Copy([1,2,3,4],2,5) renvoie [2,3,4].
  - Copy([1,2,3,4],2,0) renvoie [2,3,4].
  - Copy([1,2,3,4],-1,2) renvoie [3,4].

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

NB : pour des raisons de compatibilité avec l'ancienne version, l'index 0 correspond aussi au dernier élément de la liste.

11) DefaultAttr

- **DefaultAttr()**
- Description: cette fonction met toutes les variables correspondant aux attributs (*Color*, *Width*, ...) à leur valeur par défaut. Cette fonction renvoie la valeur *Nil*.

12) Del

- **Del(<liste>, <depart>, <nombre>)**.
- Description: supprime de la <*liste*> <*nombre*> éléments à partir du numéro <*départ*> [inclus]. Si <*nombre*> est nul, alors la fonction supprime tous les éléments de la liste à partir de l'élément numéro <*départ*>.
  Si le numéro <*depart*> est négatif, alors la liste et parcourue de droite à gauche en partant du dernier. Le dernier élément a l'index -1, l'avant-dernier a l'index $-2 \cdots$ etc. La fonction supprime les <*nombre*> éléments de la liste (ou toute la liste si <*nombre*> est nul) en allant vers la gauche.
  Le paramètre <*liste*> doit être **un nom de variable**, celle-ci est modifiée et la fonction renvoie *Nil*.
- Exemple(s): la commande [x :=[1,2,3,4], Del(x,2,2), x] renvoie [1,4].
  La commande [x :=[1,2,3,4], Del(x,-2,2), x] renvoie [1,4].

NB : pour des raisons de compatibilité avec l'ancienne version, l'index 0 correspond aussi au dernier élément de la liste.

13) Delay

- **Delay(<nb millisecondes>)**
- Description: permet de suspendre l'exécution du programme pendant le laps de temps indiqué (en milli-secondes).

14) DelButton

- **DelButton(<texte1>, ..., <texteN>)**
- Description: Cette fonction permet de supprimer dans la colonne à gauche de la zone de dessin, les boutons portant les inscriptions <*texte1*>, ..., <*texteN*>. Si la liste est vide (*DelButton()*), alors tous les boutons sont supprimés. Les arguments sont interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*.

15) DelGraph

- **DelGraph(<element1>, ..., <elementN>)**
- Description: Cette fonction permet de supprimer les éléments graphiques appelés <*element1*>, ..., <*elementN*>. Si la liste est vide (*DelGraph()*), alors tous les éléments sont supprimés. Les arguments sont interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*.

16) DelItem

- **DelItem(<nom1>, ..., <nomN>)**
- Description: Cette fonction permet de supprimer de la liste déroulante à gauche de la zone de dessin, les options appelées <*nom1*>, ..., <*nomN*>. Si la liste est vide (*DelItem()*), alors toute la liste est supprimée. Les arguments sont interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*.

17) DelMac

- **DelMac(<mac1>, ..., <macN>)**
- Description: Cette fonction permet de supprimer les macros (non prédéfinies) appelées <*mac1*>, ..., <*macN*>. Si la liste est vide (*DelMac()*), la commande est sans effet. Les arguments sont interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*.

18) DelText

- **DelText(<texte1>, ..., <texteN>)**
- Description: Cette fonction permet de supprimer dans la colonne à gauche de la zone de dessin, les labels <*texte1*>, ..., <*texteN*>. Si la liste est vide (*DelText()*), alors tous les labels sont supprimés. Les arguments sont interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*.

19) DelVar

- **DelVar(<var1>, ..., <varN>)**
- Description: Cette fonction permet de supprimer les variables globales (non prédéfinies) appelés <*var1*>, ..., <*varN*>. Si la liste est vide (*DelVar()*), la commande est sans effet. Les arguments sont interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*.

20) Der

- **Der(<expression>, <variable>, <liste>).**
- Description: cette fonction calcule la dérivée de <*expression*> par rapport à <*variable*> et l'évalue en donnant à <*variable*> les valeurs successives de la <*liste*>. La fonction *Der* renvoie la liste des résultats. Mais si on a besoin de *l'expression de la dérivée*, alors on préférera la fonction *Diff* (p. 39).
- Exemple(s):
  - la commande Der(1/x,x,[-1,0,2]) renvoie [-1,-0.25].
  - Voici le texte d'une macro appelée tangente qui prend une expression $f(x)$ en premier paramètre, une valeur réelle $x0$ en second paramètre et qui trace la tangente à la courbe au point d'abscisse x0 :

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

    On assigne la valeur $x0$ à la variable x dans l'expression $f(x)$, on stocke dans une variable A le point de coordonnées $(x0, f(x0))$ (sous forme d'affixe), on stocke dans une variable Df la dérivée en $x0$ ($f'(x0)$), puis on trace la droite passant par A et dirigée par le vecteur d'affixe $1 + if'(x0)$.

21) Diff

- **Diff(<nom>, <expression>, <variable> [, param1,..., paramN])**
- Description: cette fonction permet de créer une macro appelée <*nom*>, s'il existait déjà une macro portant ce nom, elle sera écrasée, sauf si c'est une macro prédéfinie auquel cas la commande est sans effet. Le corps de la macro créée correspond à la dérivée de <*expression*> par rapport à <*variable*>. Les paramètres optionnels sont des noms de variables, le nom de la variable <*param1*> est remplacé dans l'expression de la dérivée par le paramètre %1, le nom <*param2*> est remplacé par %2 ... etc. Cette fonction renvoie *Nil*.
- Exemple(s): après l'exécution de la commande (dans la ligne de commande en bas de la fenêtre) : Diff(df, sin(3*t), t), une macro appelée df est créée et son contenu est : 3*cos(3*t), c'est une macro sans paramètre qui contient une variable locale t, elle devra donc être utilisée en développement immédiat (c'est à dire précédée du symbole \) [2]. Par contre après la commande Diff(df,sin(3*t),t,t), le contenu de la macro df est : 3*cos(3*%1) qui est une macro à un paramètre.

22) Echange

- **Echange(<variable1>, <variable2>).**
- Description: cette fonction échange les deux variables, ce sont en fait les adresses qui sont échangées. Les contenus ne sont pas dupliqués alors qu'ils le seraient si on utilisait la commande :

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

  La fonction *Echange* renvoie la valeur *Nil*.

23) EpsCoord

- **EpsCoord(<affixe>)**
- Description: renvoie l'affixe exportée en eps. Pour les autres, il y a les macros *TeXCoord* (p. 68) et *SvgCoord* (p. 68).

2. Si par exemple on veut tracer la courbe représentative de cette fonction, dans l'option *Courbe/Paramétrée*, il faudra saisir la commande *t+i*\df* et non pas *t+i*df(t)*.

24) Eval

- **Eval(<expression>)**.
- Description: cette fonction évalue l'<*expression*> et renvoie le résultat. L'<*expression*> est interprétée comme une *chaîne de caractères* (p. 27).
- La fonction *Input* (p. 43) renvoie la saisie sous forme d'une chaîne dans la macro appelée *chaine()*. La fonction *Eval* évalue cette chaîne (comme n'importe quelle commande TeXgraph) et renvoie le résultat.
- Exemple(s): voici une commande demandant une valeur à l'utilisateur pour une variable *x* :

```
if Input("x=", "Entrez une valeur pour x", x )
then x:= Eval( chaine() )
fi
```

25) Exec

- **Exec(<programme> [, <argument(s)>, <répertoire de travail>, <attendre>, <voir fenêtre>])**.
- Description: cette fonction permet d'exécuter un <*programme*> (ou un script) en précisant éventuellement des <*arguments*> et un <*répertoire de travail*>, ces trois arguments suivants sont interprétés comme des chaînes de caractères. L'argument <*attendre*> doit valoir 0 (par défaut) ou 1, il indique si le programme attend ou non la fin du processus fils. Le dernier argument <*voir fenêtre*> doit valoir 0 (par défaut) ou 1, il indique si la fenêtre d'exécution doit être visible ou non, cet argument n'est valable que sous windows. La fonction renvoie la valeur *Nil*. Un message d'erreur s'affiche lorsque : les ressources sont insuffisantes, ou bien le programme est invalide, ou bien le chemin est invalide.
- La chaîne prédéfinie *TmpPath* contient le chemin vers un répertoire temporaire. La macro *Apercu* exporte le graphique courant dans ce dossier au format pgf dans le fichier *file.pgf*, puis elle exécute *pdflatex* sur le fichier *apercu.tex*, puis attend la fin de l'exécution avant de lancer le lecteur de pdf.
- Exemple(s): la macro *Apercu* contenue dans interface.mac est :

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

26) Export

- **Export(<mode>, <fichier>)**.
- Description: cette fonction permet d'exporter le graphique en cours, <*mode*> est une valeur numérique qui peut être l'une des constantes suivantes : tex, pst, pgf, tkz, eps, psf, pdf, epsc, pdfc, svg, bmp, obj, geom, jvx ou teg. L'exportation se fait dans <*fichier*> qui contient donc le nom du fichier, avec éventuellement le chemin.
  La chaîne prédéfinie *TmpPath* contient le chemin vers un répertoire temporaire. La macro *Apercu* exporte le graphique courant dans ce dossier au format pgf dans le fichier *file.pgf*, puis elle exécute *pdflatex* sur le fichier *apercu.tex*, puis attend la fin de l'exécution avant de lancer le lecteur de pdf.
- Exemple(s): la macro *Apercu* contenue dans interface.mac est :

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

27) ExportObject

- **ExportObject(<argument>)**
- Description: cette commande n'a d'effet que pendant un export. Elle permet d'exporter l'<*argument*> dans le fichier de sortie, cet <*argumet*> est soit le nom d'un élément graphique, soit une commande graphique (comme pour la fonction *Get* (p. 41)). Elle peut-être utile pour écrire des exports personnalisés, ceci est décrit dans *cette section* (p. 23).

28) Fenetre

- **Fenetre(<A>, <B> [, C]).**
- Description: cette fonction modifie la fenêtre graphique, c'est l'équivalent de l'option *Paramètres/Fenêtre*, **sauf que les éléments graphiques ne sont pas automatiquement recalculés**. Le paramètre <*A*> et le paramètre <*B*> sont les affixes de deux coins de la fenêtre diamétralement opposés, et le paramètre facultatif <*C*> représente les deux échelles, plus précisément, la partie réelle de <*C*> est l'échelle [en cm] sur l'axe des abscisses et la partie imaginaire de <*C*> est l'échelle [en cm] sur l'axe des ordonnées, ces deux valeurs doivent être strictement positives. Cette fonction renvoie la valeur *Nil*.

29) FileExists

- **FileExists(<nom fichier>)**
- Description: cette commande renvoie 1 si le fichier dont le nom est spécifié existe, 0 sinon.

30) Free

- **Free(<expression>, <variable>.**
- Description: cette fonction renvoie 1 si l'<*expression*> contient la <*variable*>, 0 sinon. Lorsque le deuxième argument n'est pas un nom de variable, la fonction renvoie *Nil*.

31) Get

- **Get(<argument> [, clip(0/1)]).**
- Description: lorsque le paramètre <*argument*> est un *identificateur*, la fonction cherche s'il y a un élément graphique dont le nom est <*argument*>, si c'est le cas, alors la fonction renvoie la liste des points de cet élément graphique, sinon elle renvoie la valeur *Nil*. Dans ce cas l'argument optionnel est ignoré.
  Lorsque <*argument*> n'est pas un identificateur, celui-ci est considéré comme une *fonction graphique*, la fonction *Get* renvoie la liste des points de l'élément graphique construit par cette fonction graphique mais sans créer l'élément en question. L'argument optionnel <*clip*> (qui vaut 1 par défaut) indique si l'élément doit être clippé par la fenêtre courante (valeur 1) ou non (valeur 0).
  Lorsque l'argument est vide : *Get()*, la fonction renvoie la liste des points de tous les éléments graphiques déjà construits, ceux qui sont cachés sont ignorés.
- Exemple(s): Get(Cercle(0,1)) renvoie la liste des points du cercle de centre 0 et de rayon 1 mais sans créer ce cercle, la liste des points est clippée par la fenêtre graphique.
- Exemple(s): utilisation des points d'un objet graphique :

```
\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}
```

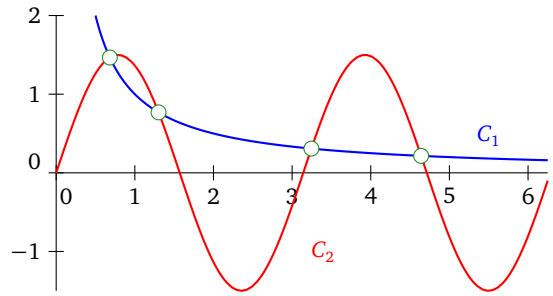


FIGURE 1: *Get*

32) GetAttr

- **GetAttr(<argument>)**
- Description: lorsque le paramètre <*argument*> est un *identificateur*, la fonction cherche s'il y a un élément graphique dont le nom est <*argument*>, si c'est le cas, alors les attributs de cet élément graphique deviennent les attributs courants, et la fonction renvoie la valeur *Nil*. Sinon, l'argument est interprété comme une chaîne de caractères puis la fonction effectue la même recherche.

33) GetMatrix

- **GetMatrix()**
- Description: cette fonction renvoie la matrice courante. (Voir aussi les commandes *ComposeMatrix* (p. 37), *SetMatrix* (p. 51), et *IdMatrix* (p. 43))

34) GetSpline

- **GetSpline(<V0>, <A0>,..., <An>, <Vn>)**
- Description: renvoie la liste des points de contrôle correspondant à la spline cubique passant par les points <*A0*> jusqu'à <*An*>. <*V0*> et <*Vn*> désignent les vecteurs vitesses aux extrémités [contraintes], si l'un d'eux est nul alors l'extrémité correspondante est considérée comme libre (sans contrainte). Le résultat doit être dessiné avec la commande graphique *Bezier* (p. 80).

35) GetStr

- **GetStr(<nom>)** ou **GetStr(<nom(arguments)>)**
- Description: évalue alphanumériquemment la macro appelée <*nom*> et renvoie la chaîne qui en résulte. Il existe un raccourci à cette commande, en accolant l'opérateur @ devant le <*nom*>.
- Exemple(s): Message(@nom) aura le même effet que Message(GetStr(nom)).

36) GrayScale

- **GrayScale(0/1)** ou **GrayScale()**.
- Description: cette fonction permet d'activer ou désactiver la conversion des couleurs en niveaux de gris. Elle équivaut à l'option *Paramètres/Gérer les couleurs* du menu de l'interface graphique.
  Lorsque l'argument est vide, la fonction renvoie l'état actuel de la conversion en niveaux de gris (0 ou 1). Sinon, elle renvoie *Nil*.

37) HexaColor

- **HexaColor(<valeur héxadécimale>)**
- Description: cette fonction renvoie la couleur correspondant à la <*valeur héxadécimale*>, cette valeur doit être passée sous forme d'une chaîne de caractères. Voir aussi la commande *Rgb* (p. 50).
- Exemple(s): `Color:=HexaColor("F5F5DC").`

38) Hide

- **Hide(<element1>, ..., <elementN>)**
- Description: Cette fonction permet de cacher les éléments graphiques appelés <*element1*>, ..., <*elementN*> en mettant leur attribut *IsVisible* à false. Les arguments sont interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*.
  Pour tout cacher on invoque la commande sans arguments : *Hide()*.
  Pour tout cacher sauf un ou plusieurs éléments, on invoque la commande : *Hide(except, element1, ..., elementN)*. Voir aussi la commande *Show* (p. 52).

39) IdMatrix

- **IdMatrix()**
- Description: change la matrice courante en la matrice identité. (Voir aussi les commandes *ComposeMatrix* (p. 37), *SetMatrix* (p. 51), et *GetMatrix* (p. 42))

40) Input

- **Input(<message> [, titre, chaîne]).**
- Description: cette fonction ouvre une boite de dialogue avec <*titre*> dans la barre de titre (par défaut le titre est vide), et dans laquelle le paramètre <*message*> est affiché, le paramètre <*chaîne*> est affiché dans la zone de saisie. Ces paramètres sont donc interprétés comme des *chaînes de caractères* (p. 27), l'utilisateur est invité à faire une saisie. S'il valide alors la fonction *Input* renvoie la valeur 1 et la chaîne saisie est **mémorisée dans la macro** *chaine()*. Si l'utilisateur ne valide pas ou si la chaîne saisie est vide, alors la fonction *Input* renvoie la valeur 0.
- Exemple(s): voir la fonction *Eval* (p. 40).

41) InputMac

- **InputMac(<nom de fichier>)** ou **Load(<nom de fichier>).**
- Description: cette fonction permet de charger en mémoire un fichier de macros (**.mac*), ou un fichier modèle (**.mod*), ou tout fichier source texgraph (**.teg*).
  Dans le premier cas (fichier **.mac*), les variables globales et les macros seront considérées comme **prédéfinies** (elles n'apparaissent pas à l'écran, elles ne seront pas enregistrées avec le graphique, mais elles sont effacées de la mémoire dès qu'on commence un nouveau graphique). Le paramètre <*nom de fichier*> est une chaîne de caractères représentant le fichier à charger avec éventuellement son chemin. Cette fonction renvoie *Nil*, et si ce fichier était déjà chargé, alors elle est sans effet. Si le fichier à charger est dans le répertoire *macros* de *TeXgraph*, ou dans le dossier *TeXgraphMac*, alors il est inutile de préciser le chemin.
- Exemple(s): InputMac("MesMacros.mac").

42) Inc

- **Inc(<variable>, <expression>).**
- Description: cette fonction évalue <*expression*> et ajoute le résultat à <*variable*>. Cette fonction est plus avantageuse que la commande variable := variable + expression, car dans cette commande la <*variable*> est évaluée [c'est à dire dupliquée] pour calculer la somme. La fonction *Inc* renvoie la valeur *Nil*.

43) Insert

- **Insert(<liste1>, <liste2> [, position]).**
- Description: cette fonction insère la <*liste2*> dans la <*liste1*> à la position numéro <*position*>. Lorsque la position vaut 0 [valeur par défaut]. La <*liste2*> est ajoutée à la fin. La <*liste1*> doit être une variable et celle-ci est modifiée. La fonction *Insert* renvoie la valeur *Nil*.
- Exemple(s): si une variable *L* contient la liste [1,4,5], alors après la commande Insert(L,[2,3],2), la variable *L* contiendra la liste [1,2,3,4,5].

44) Int

- **Int(<expression>, <variable>, <borne inf.>, <borne sup.>).**
- Description: cette fonction calcule l'intégrale de <*expression*> par rapport à <*variable*> sur l'intervalle **réel** défini par <*borne inf.*> et <*borne sup*>. Le calcul est fait à partir de la méthode de SIMPSON accélérée deux fois avec la méthode de ROMBERG, <*expression*> est supposée définie et suffisamment régulière sur l'intervalle d'intégration.
- Exemple(s): Int(exp(sin(u)),u,0,1) donne 1.63187 (Maple donne 1.631869608).

45) IsMac

- **IsMac(<nom>)**.
- Description: cette fonction permet de savoir s'il y a une macro appelée <*nom*>. Elle renvoie 1 si c'est le cas, 0 sinon.

46) IsString

- **IsString(<arg>)**.
- Description: cette fonction renvoie 1 si <*arg*> est une une chaîne de caractères, 0 sinon. Lorsque <*arg*> est une liste, seul le premier argument est testé.

47) IsVar

- **IsVar(<nom>)**.
- Description: cette fonction permet de savoir s'il y a une variable globale appelée <*nom*>. Elle renvoie 1 si c'est le cas, 0 sinon.

48) Liste

- **Liste(<argument1>, ..., <argumentn>)** ou bien **[<argument1>, ..., <argumentn>]**.
- Description: cette fonction évalue chaque argument et renvoie la liste des résultats **différents de *Nil***.
- Exemple(s): Liste(1, Arg(1+2*i), sqrt(-1), Solve(cos(x)-x,x,0,1)) renvoie le résultat [1,1.107149,0.739085].

49) ListFiles

- **ListFiles()**.
- Description: cette fonction est disponible seulement dans la version GUI de TeXgraph, elle s'utilise dans la barre de commande en bas de la fenêtre, elle affiche alors la liste des fichiers de macros (**.mac*) chargés en mémoire.

50) ListWords

- **ListWords()**.
- Description: cette fonction est disponible seulement dans la version GUI de TeXgraph, elle s'utilise dans la barre de commande en bas de la fenêtre, elle affiche alors la liste des mots en mémoire (noms des constantes, macros, commandes, variables,...).

51) LoadImage

- **LoadImage(<image>)**.
- Description: cette fonction charge le fichier <*image*>, qui doit être une image png, jpeg ou bmp. Celle-ci est affichée en image de fond et fait partie du graphique, en particulier elle est exportée dans les formats tex (visible seulement dans la version postscript), pgf, pst et teg. Pour le format pgf c'est la version png ou jpg qui sera dans le document, mais pour les versions pst et tex il faut une version eps de l'image. L'argument est interprété comme une chaîne de caractères, et la fonction renvoie la valeur *Nil*.
  Lors du chargement, la taille de l'image est adaptée à la fenêtre, mais celle-ci peut être modifiée de manière à conserver les proportions de l'image. Dès lors la position de l'image et sa taille sont fixées. On peut ensuite élargir la fenêtre si on ne veut pas que l'image occupe tout l'espace. Pour modifier la position ou la taille de l'image, il faut recharger celle-ci.

52) Loop

- **Loop(<expression>, <condition>)**.

- Description: cette fonction est une **boucle** qui construit une liste en évaluant <*expression*> et <*condition*> jusqu'à ce que le résultat de <*condition*> soit égal à 1 (pour *True*) ou *Nil*, la fonction *Loop* renvoie alors la liste des résultats de <*expression*>. Cette commande est la représentation interne de la boucle *repeat* (p. 26) dont l'utilisation est préférable pour des raisons de lisibilité.
- Exemple(s): les commandes (équivalentes) :
  [n :=1, m :=1, n, Loop([aux :=n, n :=m, m :=aux+n, n], m>100)]
  ou encore
  [n :=1, m :=1, n, while m<=100 do aux :=n, n :=m, m :=aux+n, n od]
  ou encore
  [n :=1, m :=1, n, repeat aux :=n, n :=m, m :=aux+n, n until m>100 od]
  renvoient la liste : [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89] (termes d'une suite de FIBONACCI inférieurs à 100).

53) LowerCase

- **LowerCase(<chaîne>)**.
- Description: renvoie la <*chaîne*> en minuscules.

54) Map

- **Map(<expression>, <variable>, <liste> [, mode])**.
- Description: cette fonction est une **boucle** qui construit une liste de la manière suivante : <*variable*> parcourt les éléments de <*liste*> et pour chacun d'eux <*expression*> est évaluée, la fonction *Map* renvoie la liste des résultats. Cette commande est la représentation interne de la boucle *for* (p. 26) dont l'utilisation est préférable pour des raisons de lisibilité.
  Le paramètre optionnel <*mode*> est un complexe qui vaut *Nil* par défaut, lorsque <*mode*>$= a + ib$, alors :
  - si a est un entier et $b = 0$: les éléments de la <*liste*> sont traités de a en a,
  - si a est un entier et $b = 1$: la <*liste*> est traitée par composante (deux composantes sont séparées par la constante *jump*) et les éléments de chaque composante sont traités par paquets complets de a éléments, lorsque la constante *jump* est rencontrée dans la liste, celle-ci est renvoyée dans le résultat. Un paquet non complet n'est pas traité.
  - si a est un entier et $b = -1$: la <*liste*> est traitée par composante (deux composantes sont séparées par la constante *jump*) et les éléments de chaque composante sont traités par paquets complets de a éléments, lorsque la constante *jump* est rencontrée dans la liste, celle-ci n'est pas renvoyée dans le résultat. Un paquet non complet n'est pas traité.
  - si $a =$Re(jump) : la <*liste*> est traitée par composante (deux composantes sont séparées par la constante *jump*), lorsque la constante *jump* est rencontrée dans la liste, celle-ci est renvoyée dans le résultat si $b = 1$ et n'est pas renvoyée si $b = -1$.
  - Exemple(s): voir la boucle *for* (p. 26) pour des exemples.
- Exemple(s): si L est une variable contenant une liste de points, alors la commande :
  - [sum :=0, Map(Inc(sum,z), z, L), sum] renvoie la somme des éléments de L.
  - la commande Map(z*exp(i*π/3), z, L) renvoie la liste des images des points de L par la rotation de centre O d'angle $\pi/3$.

55) Marges

- **Marges(<gauche>, <droite>, <haut>, <bas>)**
- Description: cette fonction permet de fixer les marges autour du dessin (en cm). Les nouvelles valeurs sont copiées dans les constantes *margeG*, *margeD*, *margeH* et *margeB*.

56) Merge

- **Merge(<liste>)**.
- Description: cette fonction permet de recoller des morceaux de listes pour avoir des composantes de longueur maximale, elle renvoie la liste qui en résulte.
- Exemple(s): Merge([1, 2, jump, 3, 5, jump, 3, 4, 2]) renvoie [1, 2, 3, 4, 5]. Et Merge([1, 2, jump, 3, 5, jump, 3, 4]) renvoie [1, 2, jump, 4, 3, 5].
  Attention : pour que deux extrémités soient recollées elles doivent être égales pour la machine.

57) Message

- **Message(<chaîne>)**.
- Description: cette fonction affiche le paramètre <*chaîne*> [qui est donc interprété comme une *chaîne de caractères* (p. 27)] dans une fenêtre. Quand l'utilisateur a cliqué sur *OK*, la fenêtre se referme et la fonction renvoie la valeur *Nil*.

58) Mix

- **Mix(<liste 1>, <liste 2> [, [<paquet 1>, <paquet 2>]])**.
- Description: cette fonction permet de mixer les deux <*listes*> en intercalant un élément de la deuxième après chaque élément de la première, et renvoie la liste qui en résulte. Par défaut les éléments sont comptabilisés par paquets de 1, mais on peut prendre des paquets de 2 ou plus, en modifiant le dernier argument (optionnel) <*[paquet 1, paquet 2]*>, si l'un de ces nombres est égal à *jump*, les paquets considérés seront les composantes connexes de la liste.
- Exemple(s): Mix([1,2,3], ["a","b",jump,"c","d","e",jump,"f"], [1,jump]) donne [1,"a","b",jump,2,"c","d","e",jump,3,"f"].

59) Move

- **Move(<element1>, ..., <elementN>)**.
- Description: cette fonction ne s'applique qu'aux éléments graphiques créés en mode NotXor, ce qui correspond à la valeur 1 de la variable *PenMode* (p. 32). Elle redessine les éléments graphiques <*element1*>, ..., <*elementN*>, puis les recalcule, puis les redessine, et elle renvoie la valeur *Nil*.
  Lorsqu'on redessine un élément graphique créé en mode NotXor, il s'efface en restituant le fond, on peut alors modifier sa position et le redessiner. Cette technique permet de faire glisser des objets sans avoir à réafficher tous les autres (ce qui évite d'avoir une image qui saute).
- Exemple(s): voir la fonction *Stroke* (p. 54).

60) Mtransform

- **Mtransform(<liste>, <matrice>)**.
- Description: cette fonction applique la <*matrice*> à la <*liste*> et renvoie le résultat. Si la <*liste*> contient la constante *jump*, celle-ci est renvoyée dans le résultat sans être modifiée. La <*matrice*> représente l'expression analytique d'une application affine du plan, c'est une liste de trois complexes $[z1, z2, z3]$: $z1$ est l'affixe du vecteur de translation, $z2$ est l'affixe du premier vecteur colonne de la matrice de la partie linéaire dans la base (1,i), et $z3$ est l'affixe du deuxième vecteur colonne de la matrice de la partie linéaire. Par exemple, la matrice de l'identité s'écrit ainsi : [0,1,i] (c'est la matrice par défaut).

61) MyExport

- **MyExport(<"nom">, <paramètre 1>, ..., <paramètre n>)** ou **draw(<"nom">, <paramètre 1>, ..., <paramètre n>)**
- Description: cette commande permet d'ajouter de nouveaux éléments graphiques avec un export personnalisé. Elle est décrite dans *cette section* (p. 23).

62) Nargs

- **Nargs()**.
- Description: cette fonction n'a d'effet que dans une macro, elle renvoie le nombre d'arguments avec lesquels la macro a été appelée. Hors de ce contexte, elle renvoie la valeur *Nil*. Voir aussi la fonction *Args* (p. 36).
- Exemple(s): voici le corps d'une macro *MyLabel(affixe1, texte1, affixe2, texte2, ...)* prenant un nombre indéterminé d'arguments :

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

63) NewButton

- **NewButton(<Id>, <nom>, <affixe>, <taille>, <commande> [, aide])**.
- Description: cette fonction crée dans la zone grisée à gauche dans la fenêtre un bouton dont le numéro d'identification est **l'entier** <*Id*>, le texte figurant sur le bouton est le paramètre <*nom*> qui est donc interprété comme une *chaîne de caractères* (p. 27), la position du coin supérieur gauche est donnée par le paramètre <*affixe*> qui doit être de la forme X+i*Y avec X et Y **entiers** car ce sont des coordonnées en **pixels**, la taille du bouton est donnée par le paramètre <*taille*> qui doit être de la forme *long+i*haut* où *long* désigne la longueur du bouton en pixels et *haut* la hauteur (ce sont donc des entiers), le paramètre <*commande*> est interprété comme une chaîne de caractères, c'est la commande associée au bouton, chaque clic provoquera l'exécution de cette commande. Le dernier paramètre <*aide*> est facultatif, il contient le message de la bulle d'aide s'affichant lorsque la souris passe au-dessus du bouton.
  Si on crée un bouton dont le numéro d'identification est déjà pris, alors l'ancien bouton est détruit **sauf si c'est un bouton prédéfini** (c'est à dire créé au démarrage). À chaque changement de fichier, les boutons non prédéfinis sont détruits. La fonction **NewButton** renvoie la valeur *Nil*.

64) NewGraph

- **NewGraph(<chaîne1>, <chaîne2> [, code])**.
- Description: cette fonction crée un élément graphique *Utilisateur* ayant pour nom : <*chaîne1*> et défini par la commande : <*chaîne2*>. Les deux arguments sont donc interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*. S'il existait déjà un élément graphique portant le même nom, alors celui-ci est écrasé. L'élément graphique est créé mais non dessiné, c'est la fonction *ReDraw()* qui permet de mettre l'affichage à jour.
  Le troisième paramètre <*code*> est un entier positif (optionnel), un clic gauche avec la souris sur l'élément graphique créé dans la liste des éléments graphiques, provoquera l'exécution de la macro spéciale *ClicGraph(<code>)*, cette macro n'existe pas par défaut et peut-être créée par l'utilisateur, elle est utilisée en particulier dans le fichier modèle *Mouse.mod* (dessin à la souris).
- Supposons que l'utilisateur clique sur le point d'affixe 1+i, alors une fenêtre de dialogue s'ouvre avec le message *Label=* et une ligne de saisie à remplir. Supposons que l'utilisateur entre la chaîne *Test* et valide, alors la macro va créer un élément graphique *utilisateur* portant le nom *Label1* et défini par la commande Label(1+i,"Test").
- On peut aussi utiliser la macro prédéfinie *NewLabel* et définir la macro *ClicG()* en écrivant simplement : NewLabel(%1) .
- Exemple(s): voici une macro *ClicG()* permettant la création "à la volée" de labels, on a créé auparavant une variable globale *num* initialisée à 1 :

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

65) NewItem

- **NewItem(<nom>, <commande>)**.
- Description: cette fonction ajoute dans la liste déroulante de la zone grisée à gauche dans la fenêtre, un item appelé <*nom*>, le deuxième paramètre <*commande*> est la commande associée à l'item, chaque sélection de l'item provoquera l'exécution de cette commande. Les deux arguments sont interprétés comme des chaînes de caractères. S'il existe déjà un item portant le même nom, alors l'ancien est détruit **sauf si c'est un item prédéfini** (c'est à dire créé au démarrage). À chaque changement de fichier, les items non prédéfinis sont détruits. La fonction *NewItem* renvoie la valeur *Nil*.

66) NewMac

- **NewMac(<nom>, <corps> [, param1, ..., paramN])**.
- Description: cette fonction crée une macro appelée <*nom*> et dont le contenu est <*corps*>. Les deux arguments sont donc interprétés comme des chaînes de caractères. Les paramètres optionnels sont des noms de variables, le nom de la variable <*param1*> est remplacé dans l'expression de la macro par le paramètre %1, le nom <*param2*> est remplacé par %2 ... etc. Cette fonction renvoie la valeur *Nil*. S'il existait déjà une macro portant le même nom, alors celui-ci est écrasée **si elle n'est pas prédéfinie**. Si le nom n'est pas valide, ou s'il y a déjà une macro prédéfinie portant ce nom, ou si l'expression <*corps*> n'est pas correcte, alors la fonction est sans effet.

67) NewVar

- **NewVar(<nom>, <expression>)**.
- Description: cette fonction crée une variable globale appelée <*nom*> et dont la valeur est <*expression*>. Les deux arguments sont donc interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*. S'il existait déjà une variable portant le même nom, alors celle-ci est écrasée. Si le nom n'est pas valide, ou s'il y a déjà une constante portant ce nom, alors la fonction est sans effet. Si <*expression*> n'est pas correcte, alors la valeur affectée à la variable sera *Nil*.

68) Nops

- **Nops(<liste>)**.
- Description: cette fonction évalue <*liste*> et renvoie le nombre de complexes qui la composent.
- Exemple(s): Nops([1,2,3]) renvoie la valeur 3.

69) NotXor

- **NotXor(<element1>, ..., <elementN>)**.
- Description: cette fonction ne s'applique qu'aux éléments graphiques créés en mode normal, ce qui correspond à la valeur 0 de la variable *PenMode* (p. 32). Elle change le mode des éléments graphiques <*element1*>, ..., <*elementN*> en mode NotXor puis les recalcule, et elle renvoie la valeur *Nil*.
  Lorsqu'on redessine un élément graphique qui est en mode NotXor, il s'efface en restituant le fond, on peut alors modifier sa position et le redessiner. Cette technique permet de faire glisser des objets sans avoir à réafficher tous les autres (ce qui évite d'avoir une image qui saute).
- Exemple(s): voir la fonction *Move* (p. 46).

70) OpenFile

- **OpenFile(<"nom fichier">)**.
- Description: cette fonction permet d'ouvrir un fichier texte en écriture. ATTENTION : s'il existe un fichier du même nom, alors celui-ci sera écrasé.
  En combinaison avec les commandes *WriteFile* (p. 55) et *CloseFile* (p. 37), cela permet à l'utilisateur de créer ses propres fichiers textes.

71) OriginalCoord

- **OriginalCoord(<0/1>)** ou **OriginalCoord()**
- Description: cette fonction détermine si le repère à l'exportation en *pstricks* et *tikz/pgf* est identique à celui de l'écran (ce qui correspond à la valeur 1 de l'argument, valeur par défaut), ou non. Lorsque la valeur de l'argument est nulle, le repère à l'exportation (*pstricks* et *tikz/pgf*) aura son origine en bas à gauche et l'unité sera le cm sur les deux axes. Cela est utile lorsqu'on travaille dans des répères où certaines valeurs numériques ne sont plus acceptées par TEX.
  Lorsque l'argument vaut 1, les coordonnées des points dans le fichier exporté sont les mêmes qu'à l'écran.
  Lorsque l'argument vaut 0, les coordonnées dans le fichier exporté (tex, pst, tikz/pgf) du point d'affixe z à l'écran, sont x=Re(TexCoord(z)) et y=Im(TeXCoord(z)) (et EpsCoord à la place de TeXCoord pour l'export eps).
  Lorsque l'argument est vide, la fonction renvoie l'état actuel du repère à l'exportation (0 ou 1). Sinon, elle renvoie la valeur *Nil*.

72) PermuteWith

- **PermuteWith(<liste d'index>, <liste à permuter>, [, taille des paquets ou jump])**
- Description: la <*liste à permuter*> doit être une variable, celle-ci sera permutée selon le <*liste d'index*> qui est une liste d'entiers strictement positifs. La <*liste à permuter*> est traitée par composante si elle contient la constante *jump*, et les éléments de chaque composante sont traités par paquets (de 1 par défaut) ou par composante entière (une composante se termine par *jump*), la liste est donc modifiée.
- Exemple(s): :

- [L :=[-1,0,3,5], PermuteWith([4,3,2,1], L), L] renvoie [5,3,0,-1].
- [L :=[-1,0,3,5], PermuteWith([4,3,4,1], L), L] renvoie [5,3,5,-1].
- [L :=[-1,0,3,5,6,7,8], PermuteWith([4,3,2,1], L, 2), L] renvoie [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] renvoie [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] renvoie [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] renvoie [1,-1,jump,3,0,jump,6,5,jump,9,8].

73) ReadData

- **ReadData(<fichier> [, type de lecture, séparateur]).**
- Description: cette fonction ouvre un <*fichier*> texte en lecture, celui-ci est supposé contenir une ou plusieurs listes de valeurs numériques et/ou de chaînes de caractères. Le premier argument est interprété comme une *chaîne de caractères* (p. 27) qui contient le nom du fichier (plus éventuellement son chemin). L'argument (optionnel) suivant <*type de lecture*> est une valeur numérique qui peut valoir :
  - <*type de lecture*>=0 : la fonction lit le fichier comme un fichier texte, c'est à dire une seule et même chaîne si aucun <*séparateur*> n'est spécifié, sinon, la commande renvoie une liste de chaînes,
  - <*type de lecture*>=1 : la fonction lit le fichier réel par réel et renvoie la liste ou les listes lues : `[x1, x2, ...]`,
  - <*type de lecture*>=2 : La fonction lit le fichier complexe par complexe, c'est à dire **par paquets de deux réels** et renvoie la ou les listes lues sous forme d'affixes : `[x1+i*x2, x3+i*x4, ...]`. C'est la valeur par défaut,
  - <*type de lecture*>=3 : La fonction lit le fichier par paquet de 3 réels (points de l'espace ou point3D) et renvoie la ou les listes lues sous la forme : `[x1+i*x2, x3, x4+i*x5, x6, ...]`.

  Dans les modes 1, 2, et 3, la commande peut lire une chaîne à condition qu'elle soit délimitée par le caractère ", elle sera alors insérée dans la liste. D'autre part, une ligne qui contient le caractère # sera considérée comme un commentaire à partir de celui-ci et jusqu'en fin de ligne, cette partie sera donc ignorée.
  Le troisième argument <*séparateur*>, est interprété comme une chaîne, il est supposé contenir le caractère servant à indiquer la fin de liste, entre deux listes la constante *jump* sera insérée (sauf lors de la lecture en mode texte), cet argument est facultatif et par défaut il n'y a pas de séparateur (ce qui fait donc une seule liste). Lorsque le séparateur est la fin de ligne dans le fichier, on utilisera la chaîne "LF" (*line feed*) en troisième paramètre. Lorsqu'il y a un séparateur et lorsque la lecture se fait par paquet de 2 ou 3 réels, un paquet non « complet » est ignoré.
- Exemple(s): supposons qu'un fichier texte *test.dat* contienne exactement ceci :

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

  alors l'exécution de :
  - ReadData("test.dat") donne : [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,"/") donne : [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,"/") donne : [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,"/") donne : [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") donne : [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(<fichier>).**
- Description: cette fonction ouvre un <*fichier*> en lecture, celui-ci est censé être un fichier écrit en *flattened postscript*. La fonction renvoie la liste des chemins contenus dans le fichier, le premier complexe de la liste est *largeur+i*hauteur* en cm, puis le premier complexe de chaque chemin est *Color+i*Width*. Chaque chemin se termine par un *jump* dont la partie imaginaire est un enter négatif : −1 pour eofill, −2 pour fill, −3 pour stroke et −4 pour clip.
  Il est possible de transformer un fichier pdf ou un fichier postscript en *flattened postscript* grâce à l'utilitaire *pstoedit* (`http://www.pstoedit.net/`). La macro *conv2FlatPs* (p. 76) permet cette conversion en supposant que l'utilitaire est installé sur votre système.
  La fonction *ReadFlatPs* est surtout utilisée en interne par la macro *loadFlatPs* (p. 77) qui en plus du chargement, adpate les coordonnées des points avant de renvoyer à son tour la liste des chemins que l'on peut alors dessiner avec la macro *drawFlatPs* (p. 76).
  Ce système est utilisé par la macro *NewTeXLabel* (p. 77) pour récupérer les formules TeX compilées.

75) ReCalc

- **ReCalc(<nom1>, ..., <nomN>)** ou **ReCalc()**.
- Description: cette fonction force le recalcul des éléments graphiques dont les noms sont dans la liste même si ceux-ci ne sont pas en mode *Recalcul Automatique*. Si la liste est vide (*ReCalc()*) alors tout le graphique est recalculé. Après le recalcul l'affichage est mis à jour et la fonction renvoie *Nil*.

Attention : l'utilisation de *ReCalc()* dans un élément graphique entraîne une récursion infinie et donc un plantage du programme !

76) ReDraw

- **ReDraw(<nom1>, ..., <nomN>)** ou **ReDraw()**.
- Description: cette fonction (re)dessine les éléments graphiques dont les noms sont dans la liste. Si la liste est vide (*ReDraw()*) alors tous les élments sont redessinés. Cette fonction renvoie la valeur *Nil*.

77) RenCommand

- **RenCommand(<nom>, <nouveau>)**.
- Description: cette fonction renomme la commande appelée <*nom*>. Les deux arguments sont donc interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*. Si le <*nom*> n'est pas valide, ou s'il n'y a pas de commande portant ce <*nom*>, ou s'il a déjà une commande portant le nom <*nouveau*>, alors la fonction est sans effet.

78) RenMac

- **RenMac(<nom>, <nouveau>)**.
- Description: cette fonction renomme la macro appelée <*nom*>. Les deux arguments sont donc interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*. Si le <*nom*> n'est pas valide, ou s'il n'y a pas de macro portant ce <*nom*>, ou s'il a déjà une macro portant le nom <*nouveau*>, alors la fonction est sans effet.

79) RestoreAttr

- **RestoreAttr()**.
- Description: restaure l'ensemble des attributs sauvegardés dans une pile par la commande *SaveAttr* (p. 50).

80) Reverse

- **Reverse(<liste>)**.
- Description: renvoie la <*liste*> inversée.

81) Rgb

- **Rgb(<rouge>, <vert>, <bleu>)**.
- Description: cette fonction renvoie un entier représentant la couleur dont les trois composantes sont <*rouge*>, <*vert*> et <*bleu*>, ces trois valeurs doivent être des nombres **compris entre 0 et 1**. Voir aussi la commande *HexaColor* (p. 42).
- Exemple(s): Color := Rgb(0.5,0.5,0.5)) sélectionne le gris.

82) SaveAttr

- **SaveAttr()**.
- Description: sauvegarde sur une pile l'ensemble des attributs courants. Voir aussi *RestoreAttr* (p. 50).

83) ScientificF

- **ScientificF(<réel> [, <nb décimales>]).**
- Description: transforme le <*nombre*> au format scientifique et renvoie le résultat sous forme d'une chaîne de caractères.

84) Seq

- **Seq(<expression>, <variable>, <départ>, <fin> [, pas]).**
- Description: cette fonction est une **boucle** qui construit une liste de la manière suivante : <*variable*> est initialisée à <*départ*> puis, tant que <*variable*> est dans l'intervalle (fermé) défini par <*départ*> et <*fin*>, on évalue <*expression*> et on incrémente <*variable*> de la valeur de <*pas*>. Le pas peut être négatif mais il doit être non nul. Lorsqu'il n'est pas spécifié, sa valeur par défaut est 1. Lorsque <*variable*> sort de l'intervalle, la boucle s'arrête et la fonction *Seq* renvoie la liste des résultats. Cette commande est la représentation interne de la boucle *for* (p. 26) dont l'utilisation est préférable pour des raisons de lisibilité.
- Exemple(s): Seq(exp(i*k*pi/5,k,1,5) renvoie la liste des racines cinquièmes de l'unité. La commande :
  Ligne(Seq(exp(2*i*k*pi/5, k, 1, 5), 1)
  renverra la valeur *Nil* mais dessinera un pentagone (voir *Ligne* (p. 84)) si elle est utilisée dans un élément graphique *utilisateur*.

85) Set

- **Set(<variable>, <valeur>).**
- Description: cette fonction permet d'affecter à <*variable*> [3] la <*valeur*> spécifiée. La fonction *Set* renvoie la valeur *Nil*.
  Cette commande est la représentation interne de l'affectation *:=*, dont l'utilisation est préférable pour des raisons de lisibilité.

86) SetAttr

- **SetAttr()**.
- Description: cette fonction n'a d'effet que dans un élément graphique Utilisateur. Elle permet d'affecter aux attributs de l'élément la valeur des attributs actuellement en cours. La fonction *SetAttr* renvoie la valeur *Nil*.

87) SetMatrix

- **SetMatrix(<[z1, z2, z3]>).**
- Description: cette fonction permet de modifier la matrice courante (celle-ci affecte tous les éléments graphiques sauf les axes et les grilles dans la version actuelle). Cette matrice représente l'expression analytique d'une application affine du plan, c'est une liste de trois complexes : $z1$ qui est l'affixe du vecteur de translation, $z2$ qui est l'affixe du premier vecteur colonne de la matrice de la partie linéaire dans la base (1,i), et $z3$ qui est l'affixe du deuxième vecteur colonne de la matrice de la partie linéaire. Par exemple, la matrice de l'identité s'écrit ainsi : [0,1,i] (c'est la matrice par défaut). (Voir aussi les commandes *GetMatrix* (p. 42), *ComposeMatrix* (p. 37), et *IdMatrix* (p. 43))
- Exemple(s): si $f : z \mapsto f(z)$ est une application affine, alors sa matrice est $[f(0), f(1) - f(0), f(i) - f(0)]$, ce calcul peut-être fait par la macro *matrix()* de TeXgraph.mac : SetMatrix(matrix(i*bar(z))) affecte la matrice de la symétrie orthogonale par rapport à la première bissectrice.

3. Il n'est pas nécessaire de déclarer les variables, elles sont implicitement locales et initialisées à *Nil* sauf si c'est le nom d'une variable globale ou d'une constante prédéfinie (comme i, π, e, ...).

```
\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}
```

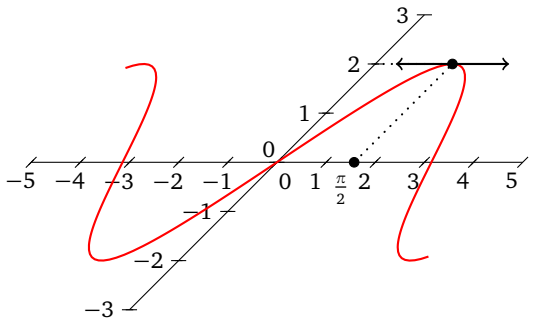


FIGURE 2: *Repère non orthogonal*

88) Show

- **Show(<element1>, ..., <elementN>).**
- Description: Cette fonction permet de montrer les éléments graphiques appelés <*element1*>, ..., <*elementN*> en mettant leur attribut *IsVisible* à true. Les arguments sont interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur *Nil*. Pour tout montrer on invoque la commande sans arguments : *Show()*. Pour tout montrer sauf un ou plusieurs éléments, on invoque la commande : Show(except, element1, ..., elementN). Voir aussi la commande *Hide* (p. 42).

89) Si

- **Si(<condition1>, <expression1>, ..., <conditionN>, <expressionN> [, sinon]).**
- Description: cette fonction évalue <*condition1*>. Une condition est une expression dont le résultat de l'évaluation doit être 0 [pour *False*] ou 1 [pour *True*], sinon il y a un échec et la fonction renvoie la valeur *Nil*. Si la condition donne la valeur 1 alors la fonction évalue <*expression1*> et renvoie le résultat, si elle vaut 0, elle évalue <*condition2*>, si celle-ci donne la valeur 1 alors la fonction évalue <*expression2*>, sinon etc... Lorsqu'aucune condition n'est remplie, la fonction évalue l'argument <*sinon*>, s'il est présent, et renvoie le résultat, sinon la fonction renvoie *Nil*. Cette commande est la représentation interne de l'alternative *if* (p. 26) dont la syntaxe est préférable pour des raisons de lisibilité.
- Exemple(s): définition d'une macro f(x) représentant une fonction f d'une variable x définie en plusieurs morceaux :
  Si(%1<-1,1-exp(pi*(%1+1)),%1<0,sin(π*%1),sh(pi*%1)),
  c'est à dire $f(x) = 1 - exp(\pi(x+1))$ si $x < -1$, $f(x) = sin(\pi x)$ si $-1 \leqslant x < 0$, $f(x) =$sh(πx) sinon.

90) Solve

- **Solve(<expression>, <variable>, <borne inf.>, <borne sup.> [, n]).**
- Description: cette fonction "résout" l'équation <*expression*>=0 par rapport à la variable **réelle** <*variable*> dans l'intervalle défini par <*borne inf.*> et <*borne sup.*>. Cet intervalle est subdivisé en <*n*> parties [par défaut n=25] et on utilise la méthode de NEWTON sur chaque partie. La fonction renvoie la liste des résultats.
- Exemple(s):
  - Solve(sin(x), x, -5, 5) donne [-3.141593, 0, 3.141593].
  - Équation : $\int_0^x \exp(t^2)dt = 1$: Solve(Int(exp(u^2),u,0,x)-1, x, 0, 1) donne 0.795172 et l'exécution de Int(exp(u^2), u, 0, 0.795172) donne 1.
  - Solve($x^2 + x + 1$, x, -1, 1) renvoie *Nil*.

91) Sort

- **Sort(<liste de complexes> [, option]).**

- Description: cette fonction trie la liste passée en argument dans l'ordre lexicographique, si l'argument <*option*> vaut 0 (valeur par défaut), ou dans l'ordre lexicographique inverse si l'argument <*option*> vaut 1. Cette liste doit être une variable, et celle-ci sera modifiée. Si la liste contient la constante *jump* alors celle-ci est recopiée telle quelle dans le résultat, et les « composantes connexes » de la liste sont triées indépendamment les unes des autres. La fonction renvoie la valeur *Nil*.
- Exemple(s): si la variable *L* contient la liste [-2,-3+i,1,1-2*i, jump, 3,5,-6] alors après l'exécution de Sort(L), la variable contiendra la liste [-3+i,-2,1-2*i,1,jump,-6,3,5], et après l'exécution de Sort(L,1), la variable contiendra la liste [1,1-2*i,-2,-3+i,jump,5,3,-6]. La méthode utilisée est le Quick Sort..

92) Special

- **Special(<chaîne>)**.
- Description: cette fonction n'aura d'effet que dans un élément graphique Utilisateur (comme les fonctions graphiques). L'argument doit être une chaîne de caractères (délimitée par " et "), si cette chaîne contient les balises \[, et \], alors tout le texte contenu entre ces deux balises est interprété par TeXgraph et le résultat est replacé dans la chaîne. Puis cette chaîne sera écrite telle quelle dans le fichier d'exportation (c'est en fait un Label créé avec LabelStyle= special).
- Exemple(s): Special("\psdot(\[1+\],\[2^3\])"), écrira dans le fichier exporté : \psdot(2,8).

93) Str

- **Str(<nom de macro>)**.
- Description: lors d'une évaluation alphanumérique, cette fonction renvoie la définition de la macro appelée <*nom de macro*> (sauf si c'est une macro prédéfinie). En dehors de ce contexte, la fonction *Str* renvoie *Nil*. L'argument <*nom de macro*> est lui-même interprété comme une chaîne de caractères.
- Exemple(s): supposons que la macro f soit définie par %1+i*sin(%1), alors la commande Label(0,["f(%1)=",Str("f")]) affichera à l'écran à l'affixe 0, le texte : f(%1)=%1+i*sin(%1).

94) StrArgs

- **StrArgs(<entier>)**.
- Description: cette fonction n'a d'effet que dans une macro, elle renvoie l'argument numéro <*entier*> avec lequel la macro a été appelée, sous forme d'une chaîne. Hors de ce contexte, elle renvoie la valeur *Nil*. Voir également la commande *Args* (p. 36)
- Exemple(s): Voir la fonction *Nargs* (p. 46).

95) StrComp

- **StrComp(<chaîne1>, <chaîne2>)**.
- Description: les deux arguments sont interprétés comme une chaîne de caractères, et les deux chaînes sont comparées, si elles sont égales alors la fonction renvoie la valeur 1, sinon la valeur 0. Depuis la version 1.97 on peut comparer deux chaînes avec le symbole =.
- Exemple(s): la combinaison de touches : Ctrl+Maj+<lettre> lance automatiquement l'exécution de la macro spéciale : *OnKey(<lettre>)*. L'utilisateur peut définir cette macro avec par exemple la commande :

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

96) StrCopy

- **StrCopy(<chaîne>, <indice départ>, <quantité>)**.
- Description: renvoie la chaîne résultant de l'extraction (fonctionne comme la commande *Copy* (p. 37)).

97) StrDel

- **StrDel(<variable>, <indice départ>, <quantité>)**.

- Description: modifie la <*variable*> en supprimant <*quantité*> caractères à partir de <*indice départ*> (fonctionne comme la commande *Del* (p. 38)). Si la <*variable*> contient une liste de chaînes, seule la première est modifiée. Si la <*variable*> ne contient pas de chaîne, la commande est sans effet.

98) StrEval

- **StrEval(<expression>)**.
- Description: cette commande évalue l'<*expression*> (qui doit être une chaîne de caractères), et renvoie le résultat sous forme d'une chaîne de caractères.

99) String

- **String(<expression mathématique>)**.
- Description: lors d'une évaluation alphanumérique, cette fonction renvoie <*expression mathématique*> sous forme d'une chaîne. En dehors de ce contexte, la fonction *String* renvoie *Nil*.

100) String2Teg

- **String2Teg(<expression>)**.
- Description: cette fonction fait une évaluation alphanumérique de l'<*expression*> et renvoie le résultat sous forme de chaîne de caractères en doublant tous les caractères " rencontrés. La chaîne résultante est ainsi lisible par TeXgraph.

101) StrLength

- **StrLength(<chaîne>)**.
- Description: renvoie le nombre de caractères de la chaîne.

102) Stroke

- **Stroke(<element1>, ..., <elementN>)**.
- Description: cette fonction recalcule les éléments graphiques <*element1*>, ..., <*elementN*>, puis les redéssine en mode NORMAL, et renvoie la valeur *Nil*.
- Exemple(s): on a créé deux variables globales : *a* et *drawing*. On va créer le cercle de centre a et de rayon 1, appelé *objet1*, on souhaite pouvoir déplacer cet objet à la souris. Pour cela, on crée la macro *ClicG* avec la commande :

```
[PenMode:=1, {mode NotXor}
 NewGraph("objet1", "Cercle(a,1)"),
 PenMode:=0, {mode normal}
 ReDraw(), {on montre}
 drawing:=1]
```

on crée la macro MouseMove avec la commande : if drawing then a :=%1 {on déplace le centre}, Move(objet1) fi,
puis la macro LButtonUp avec la commande : if drawing then Stroke(objet1), drawing :=0 fi.
La macro *ClicG* crée l'objet1 en mode NotXor, rafraîchit l'affichage graphique et passe en mode "dessin". La macro *MouseMove* permet de placer le centre à la position de la souris, puis de déplacer l'objet1. Lorsque le bouton gauche est relâché, on dessine l'objet1 en mode normal, puis on quitte le mode "dessin".

103) StrPos

- **StrPos(<motif>, <chaîne>)**.
- Description: renvoie la position (entier) du premier motif dans la chaîne.

104) StrReplace

- **StrReplace(<chaîne>, <motif à remplacer>, <motif de remplacement).**
- Description: renvoie la chaîne résultant du remplacement.

105) TeX2FlatPs

- **TeX2FlatPs(<"formule"> [, dollar(0/1)]).**
- Description: cette commande renvoie une <*formule*> TEX sous forme d'une liste de chemins, le résultat doit être dessiné avec la macro *drawFlatPs* (p. 76). La <*formule*> est écrite dans un fichier appelé *formula.tex*. Ce fichier est appelé par le fichier *formule.tex* qui se trouve dans le dossier de travail de TeXgraph, pour être compilé par TEX. Si l'option <*dollar*> vaut 1 alors la formule sera délimitée par \] et \], sinon elle est écrite telle quelle. Pour plus de renseignements sur la façon dont sont récupérés ces chemins, voir la commande *ReadFlatPs* (p. 49).

106) Timer

- **Timer(<milli-secondes>).**
- Description: règle l'intervalle de temps pour le timer, celui exécute régulièrement une certaine macro (que l'on peut définir avec la commande TimerMac). Pour stopper le timer il suffit de régler l'intervalle de temps à 0.

107) TimerMac

- **TimerMac(<corps de la macro à exécuter>).**
- Description: cette commande permet de créer une macro qui sera attachée au timer. L'argument est interprété comme une chaîne de caractères et doit correspondre au corps de la macro (celle-ci sera appélée TimerMac). Pour des raisons de performances, il est préférable d'éviter trop d'appels à d'autres macros dans celle-ci. Cette fonction renvoie la valeur 1 si la macro est correctement définie, 0 en cas d'erreur. Attention, l'exécution de TimerMac ne déclenche pas le timer ! Il faut utiliser la commande Timer pour cela.
- Exemple(s): soit *A* une variable globale (un point), soit *dotA* un élément graphique qui dessine le point, voilà une commande qui déplace *A* :

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

108) UpperCase

- **UpperCase(<chaîne>).**
- Description: renvoie la <*chaîne*> en majuscules.

109) VisibleGraph

- **VisibleGraph(<0/1>)** ou **VisibleGraph()**.
- Description: cette fonction permet d'activer ou désactiver la zone de dessin dans l'interface graphique. Lorsque celle-ci est désactivée, son contenu ne change plus car il n'y a plus de mise à jour de la zone. Désactiver l'affichage graphique peut permettre dans certains cas un gain de temps pour enregistrer une animation par exemple.
  Lorsque l'argument est vide, la fonction renvoie simplement l'état actuel de l'affichage graphique (0 ou 1). Sinon, elle renvoie *Nil*.

110) WriteFile

- **WriteFile(<argument>)** ou **WriteFile(<nom fichier>, <argument>)**.
- Description: dans la première version, cette fonction permet d'écrire soit dans le dernier fichier texte ouvert par la commande *OpenFile* (p. 48), soit dans le fichier d'exportation si l'exécution de cette commande a lieu pendant une exportation par l'intermédiare des macros *Bsave* (p. 100) et/ou *Esave* (p. 100). Dans la deuxième version, l'écriture se fait dans le fichier dont le nom est spécifié, à condition que celui-ci soit effectivement ouvert. Les deux arguments sont évalués alphanumériquement.

- Exemple(s): voici ce que pourrait être la macro *Bsave* pour modifier la taille des flêches en pstricks :

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

Chapitre VI

Les opérations et les fonctions mathématiques

1) Les opérations

1.1 Opérations usuelles

- Ce sont les opérations : *+, -, *, /*. Ces symboles sont obligatoires dans les expressions, par exemple : 2x à la place de 2*x va générer une erreur.
- On peut ajouter deux listes : [1,2,3]+[4,5] donnera [5,7,3].
- On peut soustraire deux listes : [1,2,3]-[4,5,6,7] donnera [-3,-3,-3,-7].
- On peut multiplier une liste par un complexe : 5*[1,2,3] donnera [5,10,15], mais [1,2,3]*5 donnera 5.
- On peut diviser une liste par un complexe : [1,2,3]/2 donnera [0.5,1,1.5].
- On dispose en plus de l'opération x^y qui correspond à la fonction puissance x^y. L'exposant doit être réel, mais lorsque x est complexe non réel, l'exposant y doit être entier.

1.2 Opérations logiques

- Il s'agit des opérations **And** et **Or**, les valeurs booléennes True et False correspondent respectivement aux valeurs numériques 1 et 0. La macro *not()* (p. 61) permet de prendre la négation.
- Exemple(s): 1 And 0 donne 0, mais 2 Or 1 donne *Nil*.

1.3 Opérations de comparaison

Il s'agit d'opérations dont le résultat est une valeur booléenne (0 ou 1), voici la liste :

- **Egal** (ou encore =) : teste l'égalité entre deux objets dont la valeur peut être soit une liste, soit la valeur *Nil*.
- **Negal** (ou encore <>) : teste la différence entre deux objets dont la valeur peut être soit une liste, soit la valeur *Nil*.
- **Inf** (ou encore <) : teste la relation "strictement inférieur à" (entre deux réels).
- **InfOuE** (ou encore <=) : teste la relation "inférieur ou égal à".
- **Sup** (ou encore >) : teste la relation "strictement supérieur à".
- **SupOuE** (ou encore >=) : teste la relation "supérieur ou égal à".
- **Inside** : teste si le premier argument (qui doit être un affixe) est à l'intérieur (bord exclu) du polygone représenté par le deuxième argument (qui doit donc être une liste fermée).
- Exemple(s): 1 Inside [-1,2+3*i,4-i,-1] donne 1 et i Inside [-1,2+3*i,4-i,-1] donne 0.

1.4 Opérations d'intersection

Elles sont au nombre de deux :

- **Inter** : les deux arguments doivent être des listes de deux éléments (il peut y en avoir plus, mais seuls les deux premiers sont pris en compte), ils sont alors interprétés comme deux droites [définies par deux points], l'opération *Inter* détermine et renvoie le point d'intersection. Lorsque les deux droites sont parallèles, le résultat est *Nil*.
- **InterL** : les deux arguments doivent être des listes d'au moins deux éléments, ils sont alors interprétés comme deux lignes polygonales, l'opération *InterL* détermine et renvoie la liste des points d'intersection de ces deux lignes. **Les points d'intersection sont rangés suivant le même "sens de parcours" que le premier argument** (et si plusieurs points sont sur le même segment alors ils sont rangés dans le sens de parcours du deuxième argument).

1.5 Opérations de coupure

Elles sont au nombre de deux :

- **CutA** : (cut after) le premier argument doit être une liste et le second un complexe (qui est censé être sur la ligne polygonale définie par les points de la liste). L'opération *CutA* détermine et renvoie les points de la liste situés **avant le complexe**.
  Exemple(s): [1,2,3,4,5] CutA 3.5 donne [1,2,3,3.5] et [1,2,3,4,5] CutA 6 donne *Nil*.
- **CutB** : (cut before) le premier argument doit être une liste et le second un complexe (qui est censé être sur la ligne polygonale définie par les points de la liste). L'opération *CutB* détermine et renvoie les points de la liste situés **après le complexe**.

2) Les fonctions mathématiques prédéfinies

Ce sont des fonctions d'une variable **réelle** ou **complexe** suivant les cas, et qui renvoient un complexe.

2.1 abs

- **abs(<argument>)**.
- Description: c'est la fonction module des complexes.

2.2 arccos, arccsin, arctan, arccot

- **arccos(<argument>)**, **arcsin(<argument>)**, **arctan(<argument>)** et **arccot(<argument>)**.
- Description: ce sont les fonctions circulaires réciproques usuelles à variable réelle.

2.3 Arg

- **Arg(<argument>)**.
- Description: c'est la fonction argument principal (dans l'intervalle]-π ;π]).

2.4 argch, argsh, argth, argcth

- **argch(<argument>)**, **argsh(<argument>)**, **argth(<argument>)** et **argcth(<argument>)**.
- Description: ce sont les fonctions hyperboliques réciproques usuelles à variable réelle.

2.5 bar

- **bar(<argument>)**.
- Description: c'est la conjugaison des complexes.

2.6 ch, cos

- **ch(<argument>)** et **cos(<argument>)**.
- Description: cosinus hyperbolique et cosinus trigonométrique à variable réelle.

2.7 Ent

- **Ent(<argument>)**.
- Description: c'est la fonction partie entière à variable réelle.

2.8 exp

- **exp(<argument>)**.
- Description: c'est la fonction exponentielle à variable **complexe**.

2.9 Im

- **Im(<argument>)**.
- Description: fonction partie imaginaire, l'argument est un **complexe**.

2.10 ln

- **ln(<argument>)**.
- Description: fonction logarithme népérien, l'argument est un réel.

2.11 M

- **M(<a>, <b>)** ou **M(<a>, <b>, <c>)**.
- Description: les arguments sont des réels, cette fonction renvoie le complexe a+ib ou bien le point de l'espace [a+ib,c]. L'interêt de cette fonction est un codage plus compact en mémoire.

2.12 opp

- **opp(<argument>)**.
- Description: fonction opposée, l'argument est un **complexe**.

2.13 Rand

- **Rand([argument])**.
- Description: Cette fonction génère un nombre aléatoire : si l'<*argument*> est omis (*Rand()*) alors la valeur renvoyée est un nombre de l'intervalle [0 ;1[, sinon la valeur renvoyée est un entier compris entre 0 et la valeur absolue de l'<*argument*> (exclue).
- Exemple(s): Rand(256) renvoit en entier entre 0 et 255.

2.14 Re

- **Re(<argument>)**.
- Description: fonction partie réelle, l'argument est un **complexe**.

2.15 Round

- **Round(<complexe> [, nb décimales])**.
- Description: Cette fonction renvoie le <*complexe*> en arrondissant au plus proche les parties réelle et imaginaire avec le nombre de décimales souhaité (0 par défaut).

2.16 sh, sin

- **sh(<argument>)** et **sin(<argument>)**.
- Description: sinus hyperbolique et sinus trigonométrique, à variable réelle.

2.17 sqr

- **sqr(<argument>)**.
- Description: fonction carré, l'argument est un **complexe**.

2.18 sqrt

- **sqrt(<argument>)**.
- Description: fonction racine carrée, l'argument est un réel.

2.19 tan, th, cot, cth

- **tan(<argument>)**, **th(<argument>)**, **cot(<argument>)** et **cth(<argument>)**.
- Description: tangente trigonométrique, tangente hyperbolique, cotangente trigonométrique et cotangente hyperbolique,à variable réelle.

Chapitre VII

Les macros mathématiques de TeXgraph.mac

1) Opérations arithmétiques et logiques

1.1 Ceil

- **Ceil(<x>)**.
- Description: renvoie le plus petit entier supérieur ou égal au réel <x>.

1.2 div

- **div(<x>, <y>)**.
- Description: renvoie l'unique entier k tel que x-ky soit dans l'intervalle $[0;|y|[$.

1.3 mod

- **mod(<x>, <y>)**.
- Description: renvoie l'unique réel r de l'intervalle [0 ;|y|[tel que x=ky+r avec k entier.

1.4 not

- **not(<expression booléenne>)**
- Description: renvoie la valeur booléenne de la négation.

1.5 pgcd

- **pgcd(<a>, <b> [, <u>, <v>])**
- Description: renvoie la valeur d du pgcd de <a> et <b>, ainsi que deux coefficients de Bézout dans les variables <u> et <v> (si celles-ci sont présentes), de telle sorte que $au + bv = d$.

1.6 ppcm

- **ppcm(<a>, <b>)**
- Description: renvoie la valeur du ppcm de <a> et <b>.

2) Opérations sur les variables

2.1 Abs

- **Abs(<affixe>)**.
- Description: cette macro donne la norme en cm.

2.2 free

- **free(<x>).**
- Description: libère la variable <*x*> en la mettant à *Nil*. Depuis la version 1.93 on peut faire directement l'affectation à la constante *Nil*, par exemple : x :=Nil.

2.3 IsIn

- **IsIn(<affixe> [, <epsilon>]).**
- Description: renvoie 1 si l'<*affixe*> est dans la fenêtre graphique, 0 sinon. Cette macro tient compte de la matrice courante, le test se fait à <*epsilon*> près et <*epsilon*> vaut 0.0001 cm par défaut.

2.4 nil

- **nil(<x>).**
- Description: renvoie 1 si la variable <*x*> est à *Nil*, 0 sinon. Depuis la version 1.93 on peut faire directement le test d'égalité avec la constante *Nil*.

2.5 round

- **round(<liste> [, décimales])**
- Description: tronque les complexes de la <*liste*> en arrondissant au plus proche les parties réelles et imaginaires avec le nombre de <*décimales*> demandé (0 par défaut). Si la <*liste*> contient le constante *jump*, alors celle-ci est renvoyée dans la liste des résultats. Cette macro utilise la commande *Round* (p. 59) (qui ne s'applique qu'à un complexe et non une liste).

3) Opérations sur les listes

3.1 bary

- **bary(<[affixe1, coef1, affixe2, coef2, ...]>).**
- Description: renvoie le barycentre du système pondéré <*[(affixe1, coef1), (affixe2, coef2),...]*>.

3.2 del

- **del(<liste>, <liste des index à supprimer>, <quantité à supprimer>).**
- Description: renvoie la liste après avoir supprimer les éléments dont l'index figure dans la <*liste des index à supprimer*>. La <*quantité à supprimer*> (à chaque fois) est de 1 par défaut, cet argument peut être une liste lui aussi.
- Exemple(s):
  - del([1,2,3,4,5,6,7,8], [2,6,8]) donne [1,3,4,5,7].
  - del([1,2,3,4,5,6,7,8], [2,6,8], 2) donne [1,4,5].
  - del([1,2,3,4,5,6,7,8], [2,6,8], [1,2]) donne [1,3,4,5].
  - del([1,2,jump,3,4,5,jump,6,7,8],[3,7]) donne [1,2,3,4,5,6,7,8].

3.3 getdot

- **getdot(<s> , <ligne polygonale>).**
- Description: renvoie le point de la <*ligne polygonale*> ayant <*s*> comme abscisse curviligne. Le paramètre <*s*> doit être dans l'intervalle $[0;1]$, 0 pour le premier point, et 1 pour le dernier.

3.4 IsAlign

- **IsAlign(<liste points 2D> [, epsilon]).**
- Description: renvoie 1 si les points sont sur une même droite, 0 sinon. Par défaut la tolérance <*epsilon*> vaut 1E-10. La <*liste*> ne doit pas contenir la constante *jump*.

3.5 isobar

- **isobar(<[affixe1, affixe2, ...]>)**.
- Description: renvoie l'isobarycentre du système <*[affixe1, affixe2, ...]*>.

3.6 KillDup

- **KillDup(<liste> [, epsilon])**.
- Description: renvoie la liste sans doublons. Les comparaisons se font à <*epsilon*> près (qui vaut 0 par défaut).
- Exemple(s): KillDup([1.255,1.258,jump,1.257,1.256,1.269,jump] ,1.5E-3) renvoie [1.255,1.258,1.269].

3.7 length

- **length(<liste>)**.
- Description: calcule la longueur de la <*liste*> en cm.

3.8 permute

- **permute(<liste>)**.
- Description: modifie la <*liste*> en plaçant le premier élément à la fin, <*liste*> doit être une variable.
- la commande [x := [1,2,3,4], permute(x), x] renvoie [2,3,4,1].

3.9 Pos

- **Pos(<élément>, <liste>, [, epsilon])**.
- Description: renvoie la liste des positions de l'<*élément*> (chaîne ou complexe) dans la <*liste*>, la comparaison se fait à <*epsilon*> près pour les valeurs numériques, par défaut <*epsilon*> vaut 0.
- la commande Pos(2, [1,2,3,2,4]) renvoie [2,4], mais Pos(5, [1,2,3,2,4]) renvoie *Nil*.

3.10 rectangle

- **rectangle(<liste>)**.
- Description: détermine le plus petit rectangle contenant la liste, cette macro renvoie une liste de deux complexes qui représentent l'affixe du coin inférieur gauche suivi de celle du coin supérieur droit.

3.11 replace

- **replace(<liste>, <position>, <nouveau>)**.
- Description: modifie la variable <*liste*> en remplaçant l'élément numéro <*position*> par le <*nouveau*>, cette macro renvoie *Nil*.

3.12 reverse

- **reverse(<liste>)**.
- Description: renvoie la liste après avoir inverser chaque composante (deux composantes sont séparées par un *jump*).

3.13 SortWith

- **SortWith(<liste clés>, <liste>, <taille des paquets> [, mode])**.
- Description: trie la **variable** <*liste*> suivant les <*clés*>). Les éléments de la <*liste*> sont traités par <*paquets*>, l'élément numéro *i* de la liste des clés, est à la clé du paquet numéro *i* de la variable <*liste*>, la <*taille des paquets*> est de 1 par défaut ; celle-ci peut être égale à *jump* pour un traitement par composante. Si un paquet n'est pas complet, il n'est pas traité. Si la liste contient la constante *jump*, alors toutes les composantes sont triées chacune leur tour. Le dernier paramètre détermine le type de tri : <*mode*>=0 pour ordre croissant (valeur par défaut), <*mode*>=1 pour décroissant.

4) Gestion des listes par composantes

La convention adoptée est que deux composantes sont séparées par la constante *jump*. Une composante peut être vide.

4.1 CpCopy

- **CpCopy(<liste>, <index départ>, <nombre>)**.
- Description: cette fonction renvoie la liste constituée par les <*nombre*> composantes de la <*liste*> à partir de la composante numéro <*départ*> [inclus]. Si <*nombre*> est nul, alors la fonction renvoie toutes les composantes de la liste à partir de la composante numéro <*départ*>.
  Si le numéro <*départ*> est négatif, alors la liste et parcourue de droite à gauche en partant du dernier, la dernière composante a l'index -1, l'avant-dernière a l'index $-2\cdots$ etc. La fonction renvoie les <*nombre*> composantes de la liste (ou toute la liste si <*nombre*> est nul) en allant vers la gauche, mais la liste renvoyée est dans le même sens que la <*liste*>, et cette dernière n'est pas modifiée.
- Exemple(s):
  - CpCopy([1,2,jump,3,7,8,jump,4,jump,5,6], 2, 1) renvoie [3,7,8].
  - CpCopy([1,2,jump,3,7,8,jump,4,jump,5,6], -1, 2) renvoie [4,jump,5,6].
  - CpCopy([1,2,jump,3,7,8,jump,4,jump,5,6], -3, 0) renvoie [1,2,jump,3,7,8].

4.2 CpDel

- **CpDel(<variable liste>, <index départ>, <nombre>)**.
- Description: cette fonction détruit dans la <*variable liste*>, les <*nombre*> composantes à partir de la composante numéro <*départ*> [inclus]. Si <*nombre*> est nul, alors la fonction détruit toutes les composantes de la <*variable liste*> à partir de la composante numéro <*départ*>.
  Si le numéro <*départ*> est négatif, alors la liste et parcourue de droite à gauche en partant du dernier, la dernière composante a l'index -1, l'avant-dernière a l'index $-2\cdots$ etc. La fonction détruit les <*nombre*> composantes de la <variable liste> (ou toute la liste si <*nombre*> est nul) en allant vers la gauche.
  Le paramètre <*variable liste*> doit être **un nom de variable**, celle-ci est modifiée et la macro renvoie *Nil*.

4.3 CpNops

- **CpNops(<liste>)**.
- Description: cette fonction évalue <*liste*> et renvoie le nombre de composantes qui la composent.
- Exemple(s):
  - CpNops([1,2,jump,3]) renvoie la valeur 2.
  - CpNops([1,2,jump,3,jump]) renvoie la valeur 3.
  - CpNops([jump]) renvoie la valeur 2.

4.4 CpReplace

- **CpReplace(<variable liste>, <position>, <nouveau>)**.
- Description: modifie la <*variable liste*> en remplaçant la composante numéro <*position*> par <*nouveau*>, cette macro renvoie *Nil*.

4.5 CpReverse

- **CpReverse(<liste>)**.
- Description: renvoie la <*liste*> avec les composantes dans l'ordre inverse.
- Exemple(s): CpReverse([1,2,jump,3,4,5,jump]) renvoie [jump,3,4,5,jump,1,2].

5) Gestion des listes de chaînes

Cette partie est laissée pour compatibilité ascendante car depuis la version 1.97 TeXgraph gère nativement les listes de chaînes de caractères. Dans les anciennes précédentes, une telle liste est en réalité une **macro**, les éléments sont indexés à partir de 1, et la chaîne numéro *k* est donnée par `@nomListe(k)`, alors que la longueur de la liste (nombre d'éléments) est donnée par `nomListe(0)`.

5.1 StrListInit

- **StrListInit(<nomListe>, <"chaîne1">, <"chaîne2">, ...).**
- Description: crée une liste de chaînes sous forme d'une macro appelée <*nomListe*>, les arguments suivants sont interprétés comme des chaînes, ils constituent les éléments de la liste et sont indexés à partir de 1.
- Exemple(s): Après la commande StrListInit(essai, "toto", ["toto",2/4], 24), une macro du nom de **essai** est créée et son contenu est :

```
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
```

- Exemple(s): Afficher des labels avec une affixe et une orientation pour chacun, en utilisant une seule fois LabelDot.

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

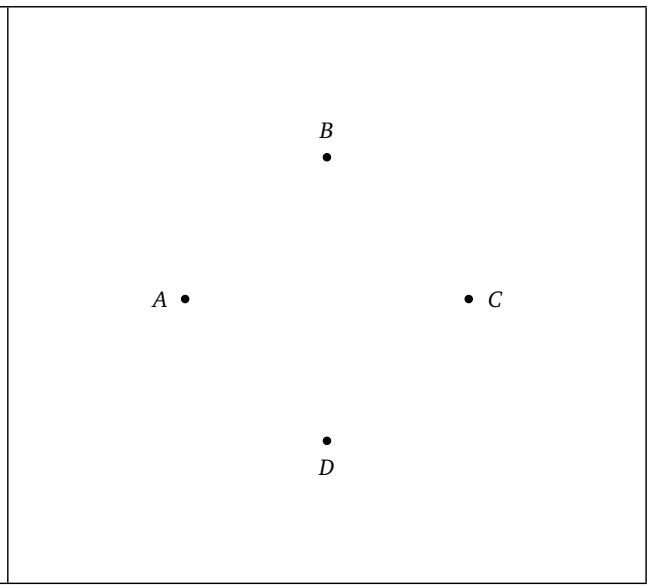


FIGURE 1: *Utilisation de StrListInit*

Une autre solution consiste à faire trois listes : nom, position, orientation :

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

5.2 StrListAdd

- **StrListAdd(<nomListe>, <"chaine1">, <"chaine2">, ...).**
- Description: cette macro ajoute à la fin de la liste de chaînes appelée <*nomListe*>, les arguments suivants (qui sont interprétés comme des chaînes). Cela suppose que la liste <*nomListe*> existe déjà, cette liste est en fait une macro qui va être entièrement réécrite pour lui ajouter les éléments supplémentaires. Il est plus rapide de définir la liste en une seule fois avec la macro *StrListInit* (p. 65) quand c'est possible.

5.3 StrListCopy

- **StrListCopy(<nomListe>, <nouvelleListe> [, index départ, nombre]).**

- Description: cette macro crée une nouvelle liste de chaînes appelée <*nouvelleListe*> en copiant <*nombre*> éléments de la liste <*nomListe*> en partant de <*index départ*>. L'argument <index départ> peut être négatif (−1 désigne le dernier élément, −2 l'avant-dernier, ...), par contre les éléments sont toujours parcourus de gauche à droite quand <nombre> est positif, et dans le sens inverse quand <nombre> est négatif. Par défaut l'<*index départ*> vaut 1 et le <*nombre*> vaut 0 (ce qui signifie « tous les éléments »).

5.4 StrListDelKey

- **StrListDelKey(<nomListe>, <index départ>, <nombre>)**.
- Description: cette macro supprime de <*nomListe*> <*nombre*> éléments à partir de l'<*index départ*>. Comme dans la commande Del, l'argument <*index départ*> peut être négatif (−1 designe le dernier élément, −2 l'avant-dernier, ...), par contre les éléments sont toujours parcourus de gauche à droite quand <nombre> est positif, et dans le sens inverse quand <nombre> est négatif. Cette macro renvoie *Nil*.

5.5 StrListDelVal

- **StrListDelVal(<nomListe>, <val1>, <val2>, ...)**.
- Description: cette macro supprime de <*nomListe*> les chaînes <*val1*>, <*val2*>..., sans avoir à connaître leurs index.

5.6 StrListGetKey

- **StrListGetKey(<nomListe>, <chaîne>)**.
- Description: cette macro renvoie l'index de la <*chaîne*> dans la liste <*nomListe*>, si elle n'y figure pas, la macro renvoie *Nil*.

5.7 StrListInsert

- **StrListInsert(<nomListe1>, <chaîne> [, <index>])**.
- Description: cette macro modifie la liste de chaînes <*nomListe*>, en insérant une nouvelle <*chaîne*> à la position <*index*>. Par défaut la valeur de <*index*> est nulle ce qui représente la fin de la liste, cette valeur peut être négative (−1 est l'index du dernier élément, −2 l'avant-dernier, ...).

5.8 StrListKill

- **StrListKill(<nomListe1>, <nomListe2>, ...)**.
- Description: cette macro détruit les listes de chaînes <*nomListe1*>, <*nomListe2*>, ...

5.9 StrListReplace

- **StrListReplace(<nomListe>, <ancienne chaîne>, <nouvelle>)**.
- Description: cette macro remplace dans la liste appelée <*nomListe*>, l'<*ancienne chaîne*> par la <*nouvelle*>.

5.10 StrListReplaceKey

- **StrListReplaceKey(<nomListe>, <index>, <nouvelle chaîne>)**.
- Description: cette macro remplace dans la liste appelée <*nomListe*>, la chaîne dont le numéro est<*index*>, par la <*nouvelle chaîne*>.

5.11 StrListShow

- **StrListShow(<nomListe> [, <index départ>, <nombre>])**.
- Description: cette macro renvoie la chaîne obtenue en copiant <*nombre*> éléments de la liste <*nomListe*> en partant de <*index départ*>, mais sans les concaténer. La chaîne renvoyée est de la forme : `"chaîne", "chaîne", ...`. L'argument <index départ> peut être négatif (−1 désigne le dernier élément, −2 l'avant-dernier, ...), par contre les éléments sont toujours parcourus de gauche à droite quand <*nombre*> est positif, et dans le sens inverse quand <nombre> est négatif. Par défaut l'<index départ> vaut 1 et le <*nombre*> vaut 0 (ce qui signifie « tous les éléments »).

6) Fonctions statistiques

6.1 Anp

- **Anp(<n>, <p>)**.
- Description: renvoie le nombre d'arrangements de <*p*> parmi <*n*>.

6.2 binom

- **binom(<n>, <p>)**.
- Description: renvoie le coefficient binomial (ou combinaison) <*p*> parmi <*n*>.

6.3 ecart

- **ecart(<liste de réels>)**.
- Description: renvoie l'écart type d'une liste numérique, les chaînes de caractères et la constante *jump* sont ignorées.

6.4 fact

- **fact(<n>)**.
- Description: renvoie la valeur de $n!$ (fonction factorielle).

6.5 max

- **max(<liste de complexes>)**.
- Description: renvoie le plus grand élément d'une liste numérique (ordre lexicographique), les chaînes de caractères et la constante *jump* sont ignorées.

6.6 min

- **min(<liste de complexes>)**.
- Description: renvoie le plus petit élément d'une liste numérique (ordre lexicographique), les chaînes de caractères et la constante *jump* sont ignorées.

6.7 minmax

- **minmax(<liste de complexes>)**.
- Description: renvoie le plus petit élément et le plus grand élément d'une liste numérique (ordre lexicographique), les chaînes de caractères et la constante *jump* sont ignorées.

6.8 median

- **median(<liste de complexes>)**.
- Description: renvoie l'élément médian d'une liste numérique (ordre lexicographique), les chaînes de caractères et la constante *jump* sont ignorées.

6.9 moy

- **moy(<liste de complexes>)**.
- Description: renvoie la moyenne d'une liste numérique, les chaînes de caractères et la constante *jump* sont ignorées.

6.10 prod

- **prod(<liste de complexes>)**.
- Description: renvoie le produit des éléments d'une liste numérique, les chaînes de caractères et la constante *jump* sont ignorées.

6.11 sum

- **sum(<liste de complexes>)**.
- Description: renvoie la somme des éléments d'une liste numérique, les chaînes de caractères et la constante *jump* sont ignorées.

6.12 var

- **var(<liste de réels>)**.
- Description: renvoie la variance d'une liste numérique, les chaînes de caractères et la constante *jump* sont ignorées.

7) Fonctions de conversion

7.1 RealArg

- **RealArg(<affixe>)**
- Description: renvoie l'argument (en radians) de l'affixe réelle d'un vecteur en tenant compte de la matrice courante.

7.2 RealCoord

- **RealCoord(<affixe écran>)**
- Description: renvoie l'affixe réelle d'un point compte tenu des échelles et de la matrice courante.

7.3 RealCoordV

- **RealCoordV(<affixe écran>)**
- Description: renvoie l'affixe réelle d'un vecteur compte tenu des échelles de la matrice courante.

7.4 ScrCoord

- **ScrCoord(<affixe réelle>)**
- Description: renvoie l'affixe écran d'un point en tenant compte des échelles et de la matrice courante.

7.5 ScrCoordV

- **ScrCoordV(<affixe réelle>)**
- Description: renvoie l'affixe écran d'un vecteur en tenant compte des échelles et de la matrice courante.

7.6 SvgCoord

- **SvgCoord(<screen affixe>)**
- Description: renvoie l'affixe exportée en svg en tenant compte des échelles et de la matrice courante.

7.7 TeXCoord

- **TeXCoord(<screen affixe>)**
- Description: renvoie l'affixe exportée en tex, pst et pgf en tenant compte des échelles et de la matrice courante. Pour l'eps il y a la commande *EpsCoord* (p. 39).

8) Transformations géométriques planes

8.1 affin

- **affin(<liste> , <[A, B]>, <V>, <lambda>)**
- Description: renvoie la liste des images des points de <*liste*> par l'affinité de base la droite <*(AB)*>, de rapport <*lambda*> et de direction le vecteur <*V*>.

8.2 defAff

- **defAff(<nom>, <A>, <A'>, <partie linéaire>)**
- Description: cette fonction permet de créer une macro appelée <*nom*> qui représentera l'application affine qui transforme <*A*> en <*A'*>, et dont la partie linéaire est le dernier argument. Cette partie linéaire se présente sous la forme d'une liste de deux complexes : [Lf(1), Lf(i)] où Lf désigne la partie linéaire de la transformation.

8.3 ftransform

- **ftransform(<liste>, <f(z)>)**
- Description: renvoie la liste des images des points de <*liste*> par la fonction <*f(z)*>, celle-ci peut-être une expression fonction de z ou une macro d'argument z.

8.4 hom

- **hom(<liste>, <A>, <lambda>)**
- Description: renvoie la liste des images de la <*liste*> par l'homothétie de centre <*A*> et de rapport <*lambda*>.

8.5 inv

- **inv(<liste>, <A>, <R>)**
- Description: renvoie la liste des images des points de <*liste*> par l'inversion de cercle de centre <*A*> et de rayon <*R*>.

8.6 mtransform

- **mtransform(<liste>, <matrice>)**
- Description: renvoie la liste des images des points de <*liste*> par l'application affine f définie par la <*matrice*>. Cette *matrice* (p. 70) est de la forme [f(0), Lf(1), Lf(i)] où Lf désigne la partie linéaire.

8.7 proj

- **proj(<liste>, <A>, <B>)** ou **proj(<liste>, <[A,B]>)**
- Description: renvoie la liste des projetés orthogonaux des points de <*liste*> sur la droite (*AB*).

8.8 projO

- **projO(<liste>, <[A,B]>, <vecteur>)**
- Description: renvoie la liste des projetés des points de <*liste*> sur la droite <*(AB)*> dans la direction du <*vecteur*>.

8.9 rot

- **rot(<liste>, <A>, <alpha>)**
- Description: renvoie la liste des images des points de <*liste*> par la rotation de centre <*A*> et d'angle <*alpha*>.

8.10 shift

- **shift(<liste>, <vecteur>)**
- Description: renvoie la liste des translatés des points de <*liste*> avec le <*vecteur*>.

8.11 simil

- **simil(<liste> , <A>, <lambda>, <alpha>)**
- Description: renvoie la liste des images des points de <*liste*>, par la similitude de centre <*A*>, de rapport <*lambda*> et d'angle <*alpha*>.

8.12 sym

- **sym(<liste>, <A>, <B>)** ou **sym(<liste>, <[A,B]>)**
- Description: renvoie la liste des symétriques des points de <*liste*>, par rapport à la droite (*AB*).

8.13 symG

- **symG(<liste>, <A>, <vecteur>)**
- Description: symétrie glissée : renvoie la liste des images des points de <*liste*>, par la symétrie orthogonale d'axe la droite passant par <*A*> et dirigée par <*vecteur*>, composée avec la translation de <*vecteur*>.

8.14 symO

- **symO(<liste>, <[A, B]>, <vecteur>)**
- Description: renvoie la liste des symétriques des points de <*liste*> par rapport à la droite <*(AB)*> et dans la direction du <*vecteur*>.

9) Matrices de transformations 2D

Une transformation affine f du plan complexe peut être représentée par son expression analytique dans la base canonique $(1, i)$, la forme générale de cette expression est :

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

cette expression analytique sera représentée par la liste [t1+i*t2, a+i*c, b+i*d] c'est à dire : [f(0), f(1)-f(0), f(i)-f(0)], cette liste sera appelée plus brièvement (et de manière abusive) *matrice* de la transformation f. Les deux derniers éléments de cette liste : [a+i*c, b+i*d], représentent la matrice de la partie linéaire de f : Lf= $f - f(0)$.

9.1 ChangeWinTo

- **ChangeWinTo(<[xinf+i*yinf, xsup+i*ysup]> [, ortho])**
- Description: modifie la matrice courante de manière à transformer la fenêtre courante en la fenêtre de grande diagonale <*[xinf+i*yinf, xsup+i*ysup]*>, la fenêtre sera orthonormée ou non en fonction de la valeur du paramètre optionnel <*ortho*> (0 par défaut).

```
\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}
```

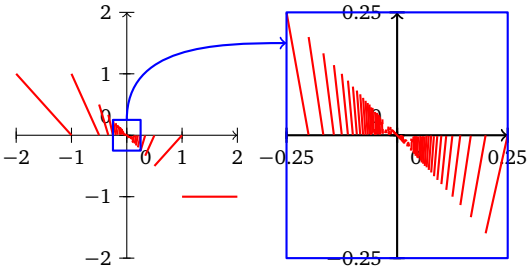


FIGURE 2: *Utilisation de ChangeWinTo*

9.2 invmatrix

- **invmatrix(<[f(0), Lf(1), Lf(i)]>)**
- Description: renvoie l'inverse de la matrice *<[f(0), Lf(1), Lf(i)]>*, c'est à dire la matrice $[f^{-1}(0), Lf^{-1}(1), Lf^{-1}(i)]$ si elle existe.

9.3 matrix

- **matrix(<fonction affine>, [variable])**
- Description: renvoie la matrice de la *<fonction affine>*, par défaut la *<variable>* est z. Cette matrice se présente sous la forme [f(0), Lf(1), Lf(i)], où f désigne l'application affine et Lf sa partie linéaire, plus précisément : Lf(1)=f(1)-f(0) et Lf(i)=f(i)-f(0).
- Exemple(s): matrix(i*bar(z)) renvoie [0,i,1].

9.4 mulmatrix

- **mulmatrix(<[f(0), Lf(1), Lf(i)]>, <[g(0), Lg(1), Lg(i)]>)**
- Description: renvoie la matrice de la composée : $f \circ g$, où f et g sont les deux applications affines définies par les matrices passées en argument.

10) Constructions géométriques planes

Ces macros définissent des objets graphiques mais ne les dessinent pas, elles renvoient une liste de points représentant ces objets.

10.1 bissec

- **bissec(<B>, <A>, <C>, <1 ou 2>)**
- Description: renvoie une liste de deux points de la bissectrice, 1=intérieure.

10.2 cap

- **cap(<ensemble1>, <ensemble2>)**
- Description: renvoie le contour de l'intersection de *<ensemble1>* avec *<ensemble2>* sous forme d'une liste de points. Ces deux ensembles sont des lignes polygonales représentant des courbes fermées, orientées dans le meme sens, ayant une forme relativement simple. La macro *set* (p. 97) permet de définir et dessiner des ensembles.
- Exemple(s): intersection de deux ensembles :

```
\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}
```

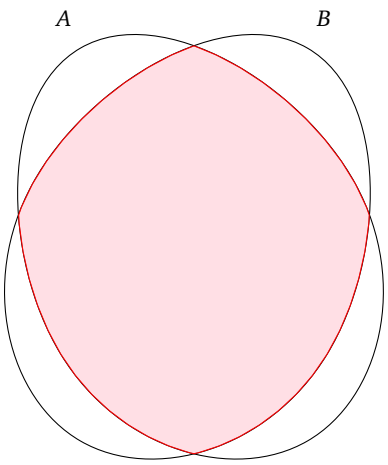


FIGURE 3: *macro cap*

10.3 capB

- **capB(<ensemble1>, <ensemble2>)**
- Description: renvoie le contour de l'intersection de <*ensemble1*> avec <*ensemble2*> sous forme d'une liste de points de contrôles qui doit être dessinée avec la macro *drawSet* (p. 94). Ces deux ensembles sont doivent également être deux listes de points de contrôle représentant des courbes fermées, orientées dans le meme sens, ayant une forme relativement simple. La macro *setB* (p. 97) permet de définir et dessiner des ensembles.
- Exemple(s): intersection de deux ensembles :

```
\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}
```

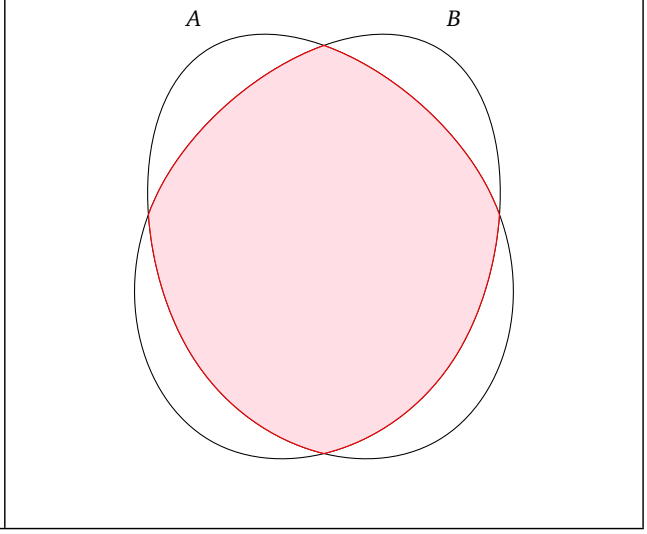


FIGURE 4: *macro capB*

10.4 carre

- **carre(<A>, <B>, <1 ou -1>)**
- Description: renvoie la liste des sommets du carré de sommets consécutifs A et B, 1=sens direct.

10.5 cup

- **cup(<ensemble1>, <ensemble2>)**
- Description: renvoie le contour de la réunion de <*ensemble1*> avec <*ensemble2*> sous forme d'une liste de points. Ces deux ensembles doivent être des courbes fermées, orientées dans le meme sens, ayant une forme relativement simple. La macro *set* (p. 97) permet de définir et dessiner des ensembles.
- Exemple(s): réunion de deux ensembles :

```
\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}
```

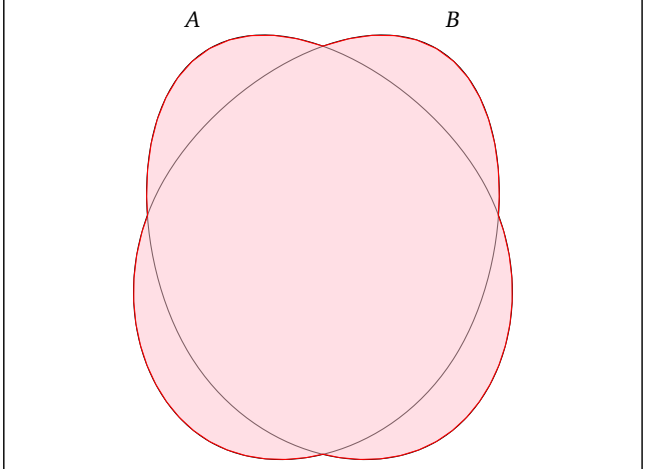


FIGURE 5: *macro cup*

10.6 cupB

- **cupB(<ensemble1>, <ensemble2>)**
- Description: renvoie le contour de la réunion de <*ensemble1*> avec <*ensemble2*> sous forme d'une liste de points de contrôles qui doit être dessinée avec la macro *drawSet* (p. 94). Ces deux ensembles sont doivent également être deux listes de points de contrôle représentant des courbes fermées, orientées dans le meme sens, ayant une forme relativement simple. La macro *setB* (p. 97) permet de définir et dessiner des ensembles.
- Exemple(s): intersection de deux ensembles :

```
\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}
```

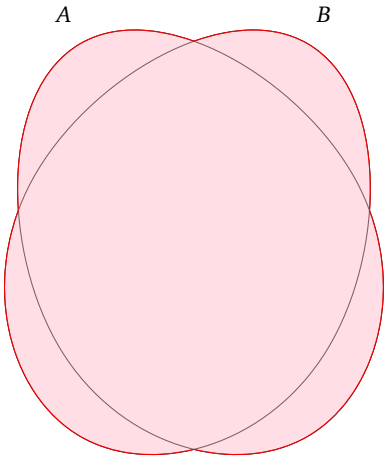


FIGURE 6: *macro cupB*

10.7 cutBezier

- **cutBezier(<courbe de Bézier>, <point>, <avant(0/1)>)**
- Description: renvoie un arc de bézier correspondant à la <*courbe de Bézier*> coupée avant ou après le <*point*>, en fonction du paramètre <*avant*>. La <*courbe de Bézier*> est une liste de points [A1,C1,C2,A2,C3,C4,A3,...] qui représente une succession de courbes de Bézier avec deux points de contrôle : $[A_i, C_k, C_{k+1}, A_{i+1}]$. Le résultat doit être dessiné par la commande *Bezier* (p. 80).

10.8 Cvx2d

- **Cvx2d(<liste>)**
- Description: renvoie l'enveloppe convexe de la <*liste*> selon l'algorithme de RONALD GRAHAM. La <*liste*> ne doit pas contenir la constante *jump*.
- Exemple(s): on choisit aléatoirement 10 points dans le pavé $[-4,4] \times [-4,4]$ que l'on place dans une variable P tout en dessinant chacun d'eux avec son numéro, puis on dessine l'enveloppe convexe.

```
\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}
```

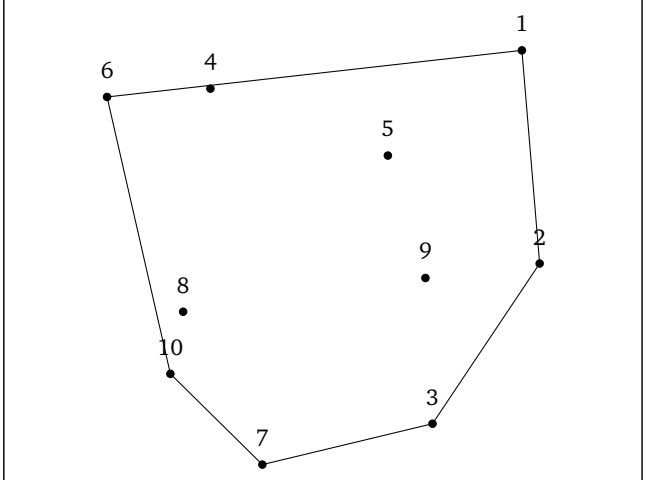


FIGURE 7: *macro Cvx2d*

10.9 Intersec

- **Intersec(<objet1>, <objet2>)**
- Description: renvoie la liste des points d'intersection des deux objets graphiques. Ces deux objets peuvent être soit des commandes graphiques (Cercle(), Droite(), ...) ou bien le nom d'un élément graphique déjà créé.
- Exemple(s): la commande Intersec(Cercle(0, 1), Droite(-1,i/2)) renvoie :
  [0.59851109463416+0.79925554731708*i, -0.99794539275033+0.00102730362483*i].

10.10 med

- **med(<A>, <B>)**
- Description: renvoie une liste de deux points de la médiatrice de $[A, B]$.

10.11 parallel

- **parallel(<[A,B]>, <C>)**
- Description: renvoie une liste de deux points de la parallèle à (AB) passant par C.

10.12 parallelo

- **parallelo(<A>, <B>, <C>)**
- Description: renvoie la liste des sommets du parallèlogramme de sommets consécutifs A, B, C.

10.13 perp

- **perp(<[A, B]>, <C>)**
- Description: renvoie une liste de deux points de la perpendiculaire à (AB) passant par C.

10.14 polyreg

- **polyreg(<A>, <B>, <nombre de cotés>)**
- Description: renvoie la liste des sommets du polygône régulier de centre A, passant par B et avec le nombre de côtés indiqué.

ou

- **polyreg(<A>, <B>, <nombre de cotés + i*sens>)** avec sens = +/-1
- Description: renvoie la liste des sommets du polygone régulier de sommets consécutifs A et B, avec le nombre de côtés indiqué et dans le sens indiqué (1 pour le sens trigonométrique).

10.15 pqGoneReg

- **pqGoneReg(<centre>, <sommet>, <[p,q]>)**
- Description: renvoie la liste des sommets du <*p/q*>-gone régulier défini par le <*centre*> et un <*sommet*>.
- Exemple(s): voir *ici* (p. 94).

10.16 rect

- **rect(<A>, <B>, <C>)**
- Description: renvoie la liste des sommets du rectangle de sommets consécutifs *A*, *B*, le côté opposé passant par *C*.

10.17 setminus

- **setminus(<ensemble1>, <ensemble2>)**
- Description: renvoie le contour de la différence <*ensemble1*> - <*ensemble2*> sous forme d'une liste de points. Ces deux ensembles doivent être des courbes fermées, orientées dans le même sens, ayant une forme relativement simple. La macro *set* (p. 97) permet de définir et dessiner des ensembles.
- Exemple(s): différence de deux ensembles :

```
\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}
```

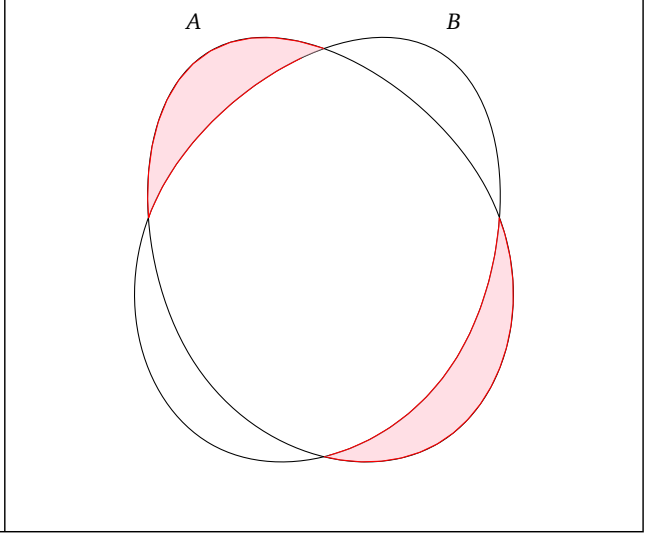


FIGURE 8: *macro setminus*

10.18 setminusB

- **setminusB(<ensemble1>, <ensemble2>)**
- Description: renvoie le contour de la différence <*ensemble1*> - <*ensemble2*> sous forme d'une liste de points de contrôles qui doit être dessinée avec la macro *drawSet* (p. 94). Ces deux ensembles sont doivent également être deux listes de points de contrôle représentant des courbes fermées, orientées dans le même sens, ayant une forme relativement simple. La macro *setB* (p. 97) permet de définir et dessiner des ensembles.
- Exemple(s): différence de deux ensembles :

```
\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}
```

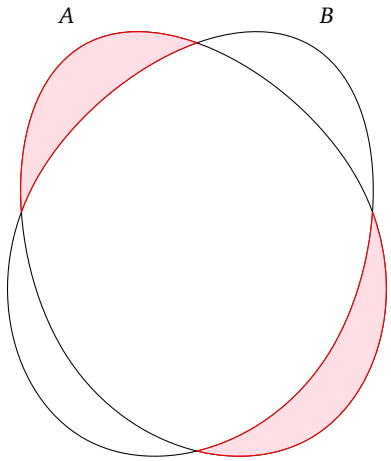


FIGURE 9: *macro setminusB*

11) Gestion du flattened postscript

Il est possible de transformer un fichier pdf ou un fichier postscript en *flattened postscript* grâce à l'utilitaire *pstoedit* (http://www.pstoedit.net/). Dans le fichier obtenu, tout est chemin, y compris le texte. TeXgraph peut récupérer tous les chemins d'un fichier écrit en *flattened postscript*. C'est que proposent les macros de cette section.

11.1 conv2FlatPs

- **conv2FlatPs(<fichier entrée>, <fichier sortie> [, dossier de travail])**
- Description: cette macro invoque l'utilitaire *pstoedit* pour transformer le <*fichier entrée*> en *flattened postscript* dans le <*fichier sortie*>. Le fichier <*fichier entrée*> doit être un fichier pdf ou ps.

11.2 drawFlatPs

- **drawFlatPs(<affixe>, <chemins lus par loadFlatPs> [, options])**
- Description: cette macro dessine à l'écran l'ensemble des chemins lus dans un fichier en *flattened postscript* par la macro *loadFlatPs* (p. 77). L'affichage se fait à l'<*affixe*> demandé. Le paramètre <*options*> est une liste (facultative) de la forme *[option1 := valeur1, ..., optionN :=valeurN]*, les options sont :
  - scale := ⟨ nombre positif ⟩ : échelle, 1 par défaut.
  - position := ⟨ center/left/right/... ⟩ : position de l'affixe par rapport à l'image, center par défaut (fonctionne comme la variable LabelStyle).
  - color := ⟨ couleur ⟩ : pour imposer une couleur, *Nil* par défaut ce qui signifie qu'on prend la couleur d'origine.
  - rotation := ⟨ angle en degrés ⟩ : 0 par défaut.
  - hollow := ⟨ 0/1 ⟩ : avec la valeur 0 (par défaut) les remplissages pleins sont effectués, sinon ce sont les valeurs courantes de *FillStyle* et *FillColor* qui sont prise en compte.
  - select := ⟨ liste des numéros de chemin à montrer ⟩ : *Nil* par défaut, ce qui signifie tous les chemins.
  - drawbox := ⟨ 0/1 ⟩ : dessine la boite englobante ou non (0 par défaut).
  - flip := ⟨ 0/1 ⟩ : applique ou non une symétrie horizontale (0 par défaut).
  - mirror0/1 : applique ou non une symétrie verticale (0 par défaut).

11.3 drawTeXlabel

- **drawTeXlabel(<affixe>, <variable contenant la formule TeX lue par loadFlatPs>, [, options])**
- Description: cette macro invoque la macro *drawFlatPs* (p. 76) pour dessiner une expression qui a été au préalable compilée par TEX. Le paramètre <*options*> est une liste (facultative) de la forme *[option1 := valeur1, ..., optionN :=valeurN]*, les options sont :
  - scale := ⟨ nombre>0 ⟩ : échelle, 1 par défaut.

– hollow := ⟨ 0/1 ⟩ : avec la valeur 0 (par défaut) les remplissages sont effectués.
Cette macro est utilisée en interne par la macro *NewTeXlabel* (p. 77).

11.4 extractFlatPs

- **extractFlatPs(<variable contenant un flattened postscript>, <liste de numéros de chemins>, [options])**
- Description: sélectionne des chemins dans une variable contenant un fichier "flattened postscript" lu par *loadFlatPs* (p. 77), le résultat est une liste : le premier complexe de la liste est largeur+i*hauteur en cm, puis le premier complexe de chaque chemin est Color+i*Width. Le résultat peut-être dessiné par *drawFlatPs* (p. 76). Le paramètre <*options*> est une liste (facultative) de la forme *[option1 := valeur1, ..., optionN :=valeurN]*, les options sont :
  – width := ⟨ nombre>0 ⟩ : largeur en cm, *Nil* par défaut pour la largeur naturelle.
  – height := ⟨ nombre>0 ⟩ : hauteur en cm, *Nil* par défaut pour la hauteur naturelle.

11.5 loadFlatPs

- **loadFlatPs(<"nom fichier en flattened postscript">, [, options])**
- Description: cette macro charge un <*fichier en flattened postscript*>, adapte les coordonnées des points et renvoie la liste des chemins (que l'on peut alors dessiner avec la macro *drawFlatPs* (p. 76)). Le paramètre <*options*> est une liste (facultative) de la forme *[option1 := valeur1, ..., optionN :=valeurN]*, les options sont :
  – width := ⟨ nombre>0 ⟩ : largeur en cm, *Nil* par défaut pour la largeur naturelle.
  – height := ⟨ nombre>0 ⟩ : hauteur en cm, *Nil* par défaut pour la hauteur naturelle.
- supposons que vous ayez le fichier *circuit.pdf* dans le dossier temporaire de TeXgraph, la commande suivante dans un élément graphique Utilisateur :

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

va permettre de charger et dessiner le contenu de ce fichier dans TeXgraph, sans faire les remplissages.

11.6 NewTeXlabel

- **NewTeXlabel(<nom>, <affixe>, <formule TeX>, [, options])**
- Description: cette macro va demander à TEX de compiler la formule dans un fichier pdf, ce fichier sera ensuite converti en un fichier eps par pstoedit, puis celui-ci sera chargé par loadFlatPs et stocké dans une variable globale appelée TeX_+nom. Un élément graphique appelé <*nom*> est créée pour dessiner la formule avec drawTeXLabel. Le paramètre <*options*> est une liste (facultative) de la forme *[option1 := valeur1, ..., optionN :=valeurN]*, les options sont :
  – dollar := ⟨ 0/1 ⟩ : indique à TeXgraph s'il doit ajouter les délimiteurs \[et \] autour de la formule, 1 par défaut.
  – scale := ⟨ nombre>0 ⟩ : échelle, 1 par défaut.
  – hollow := ⟨ 0/1 ⟩ : avec la valeur 0 (par défaut) les remplissages sont effectués.
  Dans les options, les attributs suivants peuvent également être utilisés : *LabelStyle*, *LabelAngle* et *Color*.
  Voici la définition de cette macro :

```
[dollar:=1, scale:=1, hollow:=0, $options:=%4,
 $aux:=OpenFile([TmpPath,"formula.tex"]),
 if dollar then WriteFile(["\[",%3,"\]"]) else WriteFile(%3) fi,
 CloseFile(),
 Exec("pdflatex","-interaction=nonstopmode tex2FlatPs.tex",TmpPath,1),
 Exec("pstoedit -dt -pta -f ps -r2400x2400","tex2FlatPs.pdf tex2FlatPs.eps",TmpPath,1),
 NewVar(["TeX_",%1],loadFlatPs([TmpPath,"tex2FlatPs.eps"])),
 NewGraph(%1, ["drawTeXlabel(",%2,", TeX_",%1,", [scale:=",scale,", hollow:=",hollow,"])"]),
 ReDraw()
]
```

La formule est écrite dans le fichier *formula.tex*, puis on compile le fichier *tex2FlatPs.tex* suivant :

```
\documentclass[12pt]{article}
\usepackage{amsmath,amssymb}
\usepackage{fourier}
\pagestyle{empty}
\begin{document}
\large
```

```
\input{formula.tex}%
\end{document}
```

et on convertit le résultat en *flattened postscript* avant de le charger.
Cette macro s'utilise dans la ligne de commande ou bien dans des macros qui créent des éléments graphiques, mais pas directement dans un élément graphique Utilisateur, exemple :

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

12) Autres

12.1 pdfprog

- **pdfprog()**.
- Description: cette macro est utilisée en interne pour mémoriser le programme utilisé pour faire la conversion du format eps vers le format pdf. Par défaut, cette macro contient la chaîne : *"epstopdf"*. En éditant le fichier TeXgraph.mac, vous pouvez modifier le programme utilisé.

Chapitre VIII

Fonctions et macros graphiques

Ces fonctions et macros créent un élément graphique au moment de leur évaluation et renvoient un résultat égal à *Nil*, elles ne sont utilisables **que lors de la création d'un élément graphique "Utilisateur"** [1].

Elles peuvent être utilisées dans des macros, mais elles ne seront évaluées que si ces macros sont exécutées lors de la création d'un élément graphique "Utilisateur".

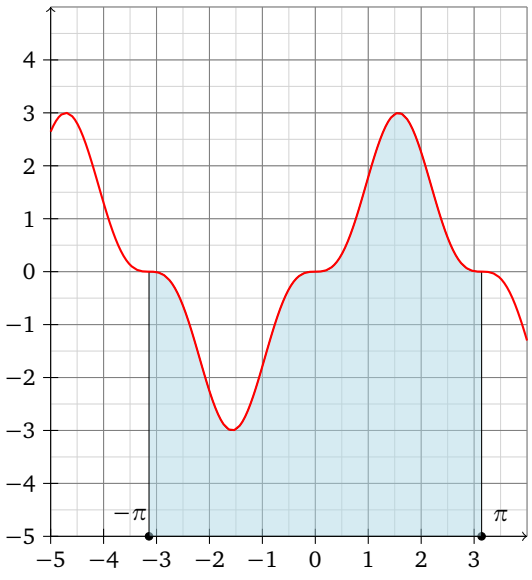
1) Fonctions graphiques prédéfinies

Notations :
<argument> : signifie que l'argument est **obligatoire**.
[, argument] : signifie que l'argument est **facultatif**.

1.1 Axes

- **Axes(<origine>, <graduationX + i*graduationY> [, position label origine]).**
- Description: dessine les axes, <*origine*> est l'affixe du point d'intersection des axes, <*graduationX*> est le pas pour les graduations sur l'axe Ox, et <*graduationY*> celui de l'axe Oy, une graduation égale à zéro signifie l'absence de graduation. La longueur des graduations est dans la variable globale xyticks que l'on peut modifier, la distance des labels à l'extrémité des graduations est dans la variable xylabelsep qui peut également être modifiée.
  Le troisième paramètre est facultatif, il permet de préciser la position des labels de l'origine (à l'intersection des axes), c'est un complexe : $a + ib$, la partie réelle concerne l'abscisse de l'origine et l'autre concerne l'ordonnée. Ces deux nombres peuvent prendre trois valeurs :
  - 0 : le label n'apparaît pas,
  - 1 : le label est affiché comme celui des autres graduations,
  - 2 : le label est décalé pour ne pas chevaucher l'autre axe (valeur par défaut).
- On peut modifier dans les *Attributs* : le style de ligne, l'épaisseur, la couleur et la taille des labels.

```
\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}
```



1. Option *Elément graphique/Créer/Utilisateur* du menu.

FIGURE 1: *Commande Axes*

1.2 (Poly-)Bézier

- **Bezier(<liste de points>).**
- Description: dessine une succession de courbes de BÉZIER (avec éventuellement des segments de droite). Il y a plusieurs possibilités pour la liste de points :

  1. une liste de trois points $[A, C, B]$, il s'agit alors d'une courbe de Bézier d'origine <A> et d'extrémité <B> avec un point de contrôle <C>, c'est la courbe paramétrée par :

  $$(1-t)^2A + 2t(1-t)C + t^2B$$

  2. une liste de 4 points ou plus : [A1, C1, C2, A2, C3, C4, A3...] : il s'agit alors d'une succession de courbes de Bézier à 2 points de contrôles, la première va de A1 à A2, elle est contrôlée par C1, C2 (paramétrée par $(1-t)^3tA1 + 3(1-t)^2tC1] + 3(1-t)t^2C2 + t^3A2$), la deuxième va de A2 à A3 et est contrôlée par C3,C4 ...etc. Une exception toutefois, on peut remplacer les deux points de contrôle par la constante *jump*, dans ce cas on saute directement de A1 à A2 en traçant un segment de droite.

- Le nombre de points calculés (par courbe) est modifiable dans les *Attributs* (variable NbPoints).

```
\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}
```

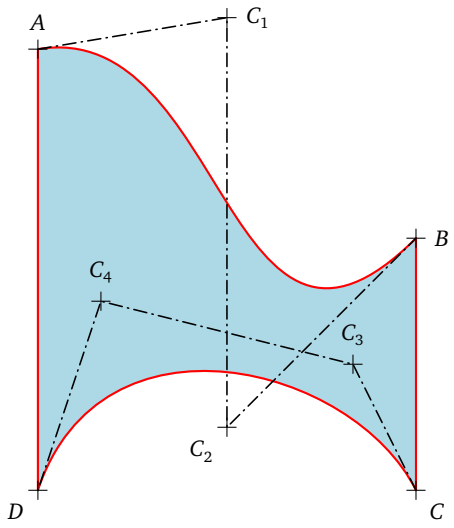


FIGURE 2: *Commande Bezier*

1.3 Cartesienne

- **Cartesienne(<f(x)> [, n, 1]).**
- Description: trace la courbe cartésienne d'équation $y = f(x)$. Le paramètre optionnel <n> est un entier (égal à 5 par défaut) qui permet de faire varier le pas de la manière suivante : lorsque la distance entre deux points consécutifs est supérieur à un certain seuil alors on calcule un point intermédiaire [par dichotomie], ceci peut être répété n fois. Si au bout de n itérations la distance entre deux points consécutifs est toujours supérieure au seuil, et si la valeur optionnelle 1 est présente, alors une discontinuité (*jump*) est insérée dans la liste des points.

```
\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}
```

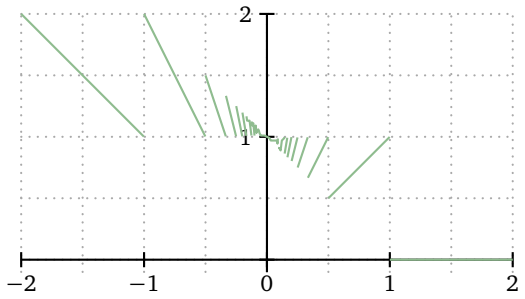


FIGURE 3: *Courbe avec discontinuités*

1.4 Courbe

- **Courbe(<f(t)> [, n, 1]).**
- Description: trace la courbe paramétrée par *<f(t)>* où f est à valeurs complexes.
  Le paramètre optionnel *<n>* est un entier (égal à 5 par défaut) qui permet de faire varier le pas de la manière suivante : lorsque la distance entre deux points consécutifs est supérieur à un certain seuil alors on calcule un point intermédiaire (par dichotomie), ceci peut être répété n fois. Si au bout de n itérations la distance entre deux points consécutifs est toujours supérieure au seuil, et si la valeur optionnelle 1 est présente, alors une discontinuité (*jump*) est insérée dans la liste des points.

1.5 Droite

- **Droite(<A>, <B> [, C]).**
- Description: trace la droite (AB) lorsque le troisième argument *<C>* est omis, sinon c'est la droite d'équation cartésienne *<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), {normale à l'ellipse}
od,
Width :=8, Color :=red,
Ellipse(0, 4, 3),
LabelDot(F, "$F$", "S", 1),
LabelDot(F', "$F'$", "S", 1)
] ;
\end{texgraph}
```

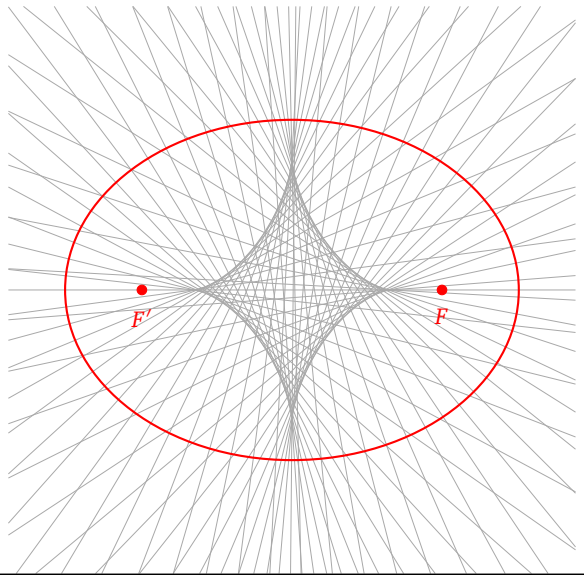


FIGURE 4: *Développée d'une ellipse*

1.6 Ellipse

- **Ellipse(<A>, <Rx>, <Ry> [, inclinaison]).**
- Description: trace une ellipse de centre *<A>* de rayons *<Rx>* et *<Ry>* sur les axes respectifs Ox et Oy. Le dernier paramètre *<inclinaison>* est un angle en degrés (nul par défaut) qui indique l'inclinaison de l'ellipse par rapport à l'horizontale.

```
\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}
```

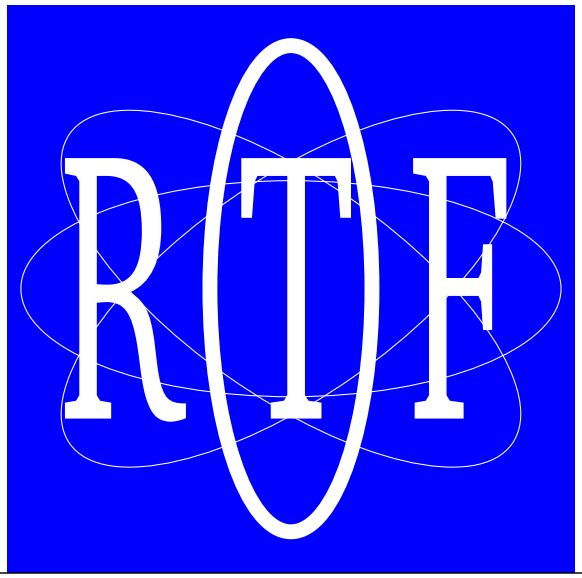


FIGURE 5: *Ellipses*

1.7 EllipticArc

- **EllipticArc(<B>, <A>, <C>, <Rx>, <Ry> [, sens]).**
- Description: trace un arc d'ellipse dont les axes sont Ox et Oy et le centre <*A*>, le rayon sur Ox est <*Rx*>, et celui sur Oy est <*Ry*>. L'arc est tracé partant de la droite (AB) jusqu'à la droite (AC), l'argument facultatif <*sens*> indique : le sens trigonométrique si sa valeur est 1 (valeur par défaut), le sens contraire si sa valeur est -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}
```

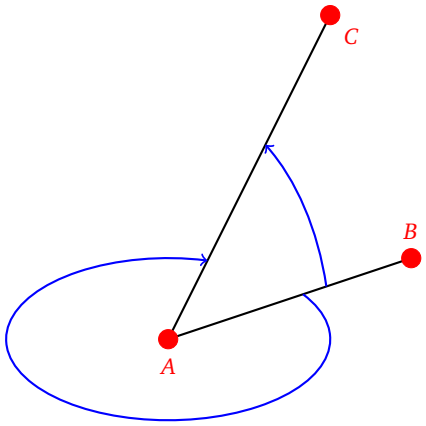


FIGURE 6: *Commande EllipticArc*

Remarque : pour un arc de cercle, il suffit de prendre <*Rx*> et <*Ry*> égaux. Mais le plus simple est d'utiliser la macro *Arc* (p. 88) qui remplace la commande **Arc** de l'ancienne version.

1.8 EquaDif

- **EquaDif(<f(t,x,y)>, <t0>, <x0 + i*y0> [, mode]).**
- Description: trace une solution approchée de l'équation différentielle : $x'(t)+iy'(t)=f(t,x,y)$ avec la condition initiale $x(t_0)=x0$ et $y(t_0)=y_0$. Le dernier paramètre est facultatif et peut valoir 0, 1 ou 2 :
  - <*mode*>=0 : la courbe représente les points de coordonnées $(x(t),y(t))$, c'est la valeur par défaut.
  - <*mode*>=1 : la courbe représente les points de coordonnées $(t,x(t))$.
  - <*mode*>=2 : la courbe représente les points de coordonnées $(t,y(t))$.

  C'est la méthode de RUNGE-KUTTA d'ordre 4 qui est utilisée.

- Exemple(s): l'équation $x'' - x' - tx = \sin(t)$ avec la condition initiale $x(0) = -1$ et $x'(0) = 1/2$, se met sous la forme :

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

en posant $X = x$ et $Y = x'$:

```
\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,
 "$x"-tx'-x=\sin(t)$\\
avec $x(0)=-1$ et $x'(0)=\frac12$")
] ;
\end{texgraph}
```

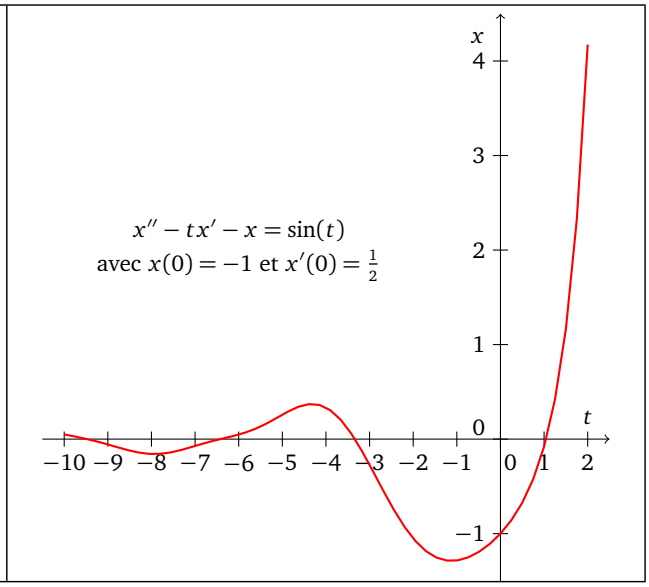


FIGURE 7: *Équation différentielle*

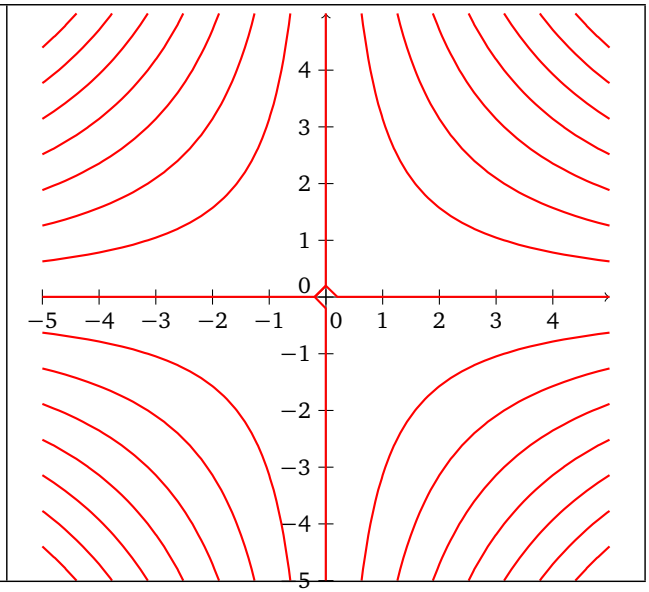
1.9 Grille

- **Grille(<origine>, <graduationX + i*graduationY>).**
- Description: dessine une grille, <*origine*> est l'affixe du point considéré comme origine, <*graduationX*> est le pas des graduations sur l'axe Ox, et <*graduationY*> celui de l'axe Oy, une graduation égale à zéro signifie l'absence de graduation.
- On peut modifier dans les *Attributs* le style de ligne, l'épaisseur et la couleur. La grille ne fait pas apparaître de graduations, on peut dessiner des axes par dessus.

1.10 Implicit

- **Implicit(<f(x,y)> [, n, m]).**
- Description: trace la courbe implicite d'équation $f(x, y) = 0$. L'intervalle des abscisses est subdivisé en <*n*> parties et l'intervalle des ordonnées en <*m*> parties, par défaut $n = m = 50$. Sur chaque pavé ainsi obtenu on teste s'il y a un changement de signe, auquel cas on applique une dichotomie sur les bords du pavé.

```
\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}
```



[TEXGRAPH 1.97]

FIGURE 8: *Équation* $\sin(x\,y)=0$

1.11 Label

- **Label(<affixe1>, <texte1>,..., <affixeN>, <texteN>).**
- Description: place la chaîne de caractères <*texte1*> à la position <*affixe1*> ... etc. Les paramètres <*texte1*>,..., <*texteN*> sont donc interprétés comme des *chaînes de caractères* (p. 27).

```
\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}
```

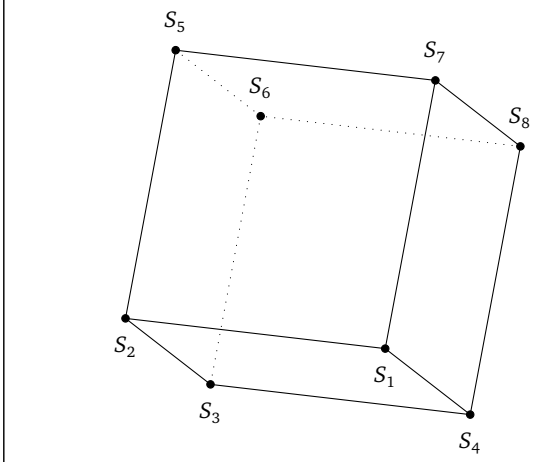


FIGURE 9: *Nommer des points*

1.12 Ligne

- **Ligne(<liste>, <fermée> [, rayon]).**
- Description: trace la ligne polygonale définie par la liste, si le paramètre <*fermée*> vaut 1, la ligne polygonale sera fermée, si sa valeur est 0 la ligne est ouverte. Si l'argument <*rayon*> est précisé (0 par défaut), alors les "angles" de la ligne polygonale sont arrondis avec un arc de cercle dont le rayon correspond à l'argument <*rayon*>.

```
\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}
```

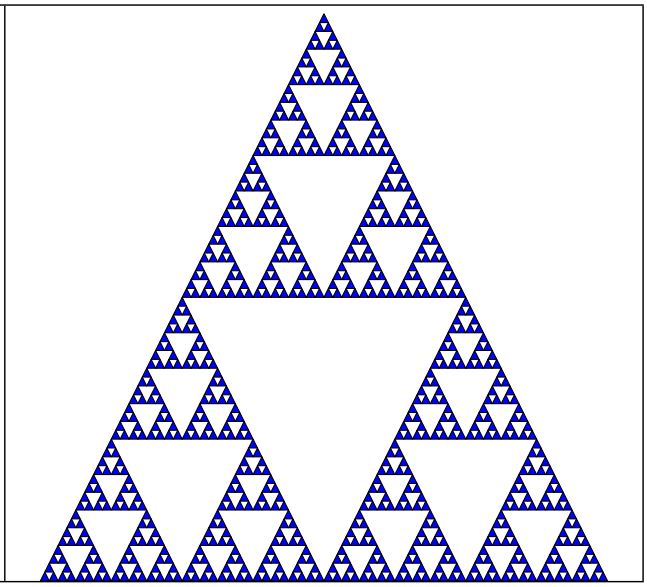


FIGURE 10: *Triangle de* SIERPINSKI

1.13 Path

- **Path(<liste> [, fermé (0/1)]**

- Description: trace le chemin représenté par <*liste*> et ferme la dernière composante de celui-ci si l'argument optionnel vaut 1 (sa valeur par défaut est 0). La liste est une succession de points (affixes) et d'instructions indiquant à quoi correspondent ces points, ces instructions sont :
  - **line** : relie les points par une ligne polygonale,
  - **linearc** : relie les points par une ligne polygonale mais les angles sont arrondis par un arc de cercle, la valeur précédent la commande linearc est interprétée comme le rayon de ces arcs.
  - **arc** : dessine un arc de cercle, ce qui nécessite quatre arguments : 3 points (départ, centre, arrivée) et le rayon, plus éventuellement un cinquième argument : le sens (+/- 1), le sens par défaut est 1 (sens trigonométrique).
  - **ellipticArc** : dessine un arc d'ellipse, ce qui nécessite cinq arguments : 3 points, le rayonX, le rayonY, plus éventuellement un sixième argument : le sens (+/- 1), le sens par défaut est 1 (sens trigonométrique), plus éventuellement un septième argument : l'inclinaison en degrés du grand axe par rapport à l'horizontale.
  - **curve** : relie les points par une spline cubique naturelle.
  - **bezier** : relie le premier et le quatrième point par une courbe de Bézier (les deuxième et troisième points sont les points de contrôle).
  - **circle** : dessine un cercle, ce qui nécessite deux arguments : un point et le centre, ou bien trois arguments qui sont trois points du cercle.
  - **ellipse** : dessine une ellipse, les arguments sont : un point, le centre, rayon rX, rayon rY, inclinaison du grand axe en degrés par rapport à l'horizontale (facultatif).
  - **move** : indique un déplacement sans tracé.
  - **closepath** : ferme la composante en cours.

  Par convention, le premier point du tronçon numéro n+1 est le dernier point du tronçon numéro n.
- Exemple(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}
```

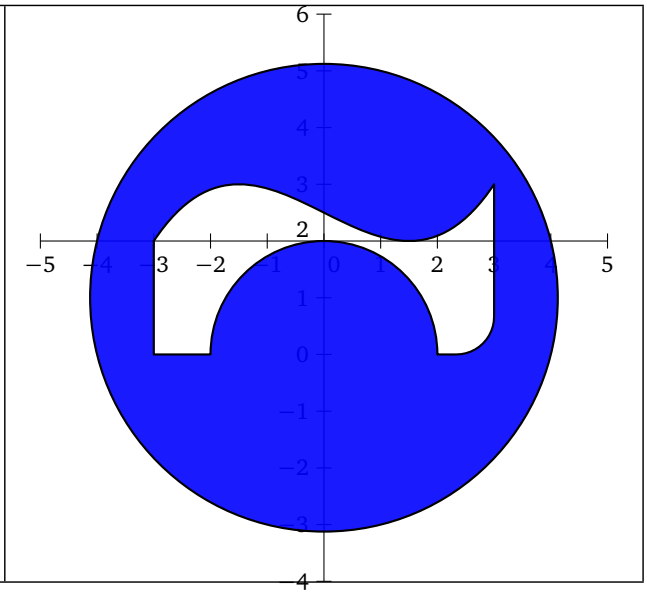


FIGURE 11: *Commande Path et Eofill*

1.14 Point

- **Point(<A1>, ..., <An>)**.
- Description: représente le nuage de points <*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}
```

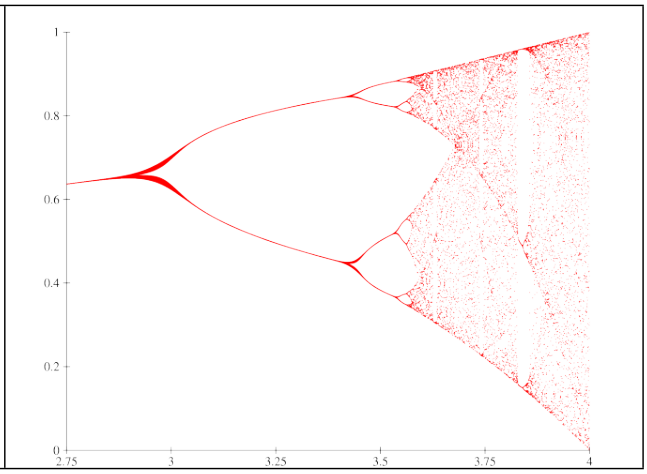


FIGURE 12: *Diagramme de bifurcation de la suite* $u_{n+1} = ru_n(1-u_n)$

1.15 Polaire

- **Polaire(<r(t)> [, n, 0/1]).**
- Description: trace la courbe polaire d'équation $\rho = r(t)$, <*expression*>. Le paramètre optionnel <*n*> est un entier (égal à 5 par défaut) qui permet de faire varier le pas de la manière suivante : lorsque la distance entre deux points consécutifs est supérieur à un certain seuil alors on calcule un point intermédiaire (par dichotomie), ceci peut être répété *n* fois. Si au bout de *n* itérations la distance entre deux points consécutifs est toujours supérieure au seuil, et si la valeur optionnelle 1 est présente, alors une discontinuité (*jump*) est insérée dans la liste des points.

```
\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}
```

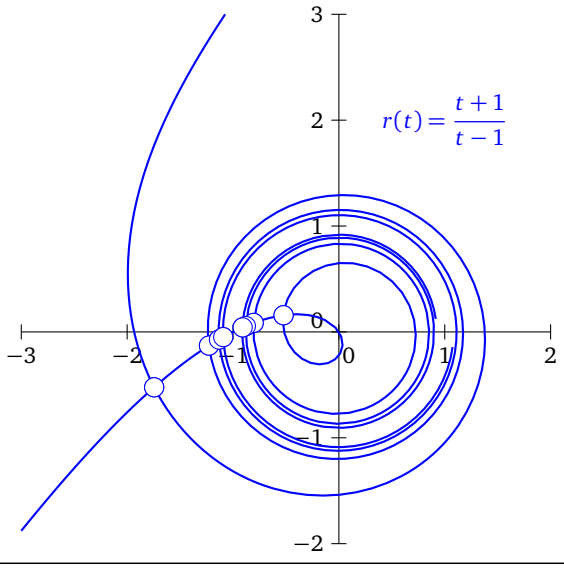


FIGURE 13: *Courbe polaire et points doubles*

1.16 Spline

- **Spline(<V0>, <A0>,..., <An>, <Vn>).**
- Description: trace la spline cubique passant par les points <*A0*> jusqu'à <*An*>. <*V0*> et <*Vn*> désignent les vecteurs vitesses aux extrémités [contraintes], si l'un d'eux est nul alors l'extrémité correspondante est considérée comme libre (sans contrainte).

```
\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, "libre"),
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, "contrainte"),
Width :=4,
Arrows :=1, Ligne([A, A+2, jump, D, D+2*i], 0)
] ;
\end{texgraph}
```

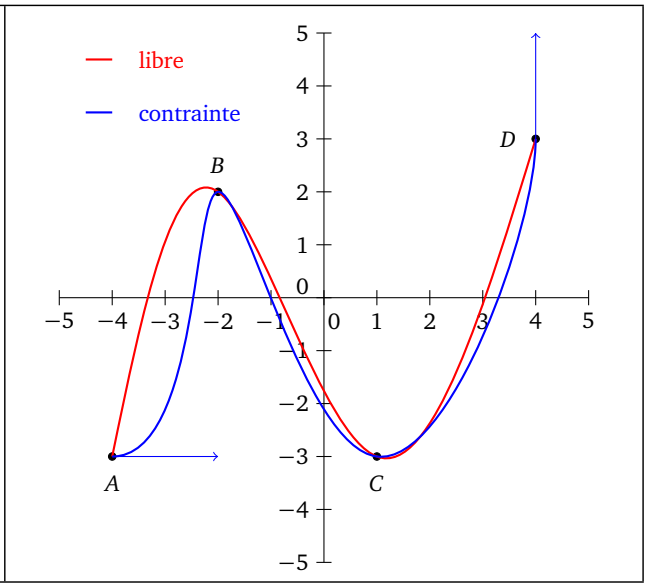


FIGURE 14: *Commande Spline*

2) Commandes de dessin bitmap

La version 1.97 propose quelques commandes de base pour faire du dessin bitmap. Ce dessin bitmap peut être enregistré (au format *bmp*) mais il n'est pas pris en compte par les autres exports du logiciel. Ces commandes ne sont fonctionnelles qu'avec la version GUI de TeXgraph. Chaque pixel est repéré son affixe $x+iy$ où x et y sont deux entiers, l'origine est en haut à gauche de la zone de dessin **marges exclues**, l'axe Ox est dirigé vers la droite et l'axe Oy vers le bas.

2.1 DelBitmap

- **DelBitmap()**.
- Description: détruit le bitmap en cours.

2.2 GetPixel

- **GetPixel(<liste d'affixes de pixels>)**.
- Description: renvoie la liste des couleurs des pixels de la <*liste*> de pixels. Les affixes des pixels sont de la forme $a+ib$ avec a et b des entiers positifs ou nuls, l'origine étant le coin supérieur gauche de la zone graphique **marges exclues**.

2.3 MaxPixels

- **MaxPixels()**.
- Description: renvoie la taille de la zone graphique en pixels sous la forme : *maxX+i*maxY*, ainsi pour les coordonnées des pixels (coordonnées entières), l'intervalle des abscisses est *[0 .. maxX]* et celui des ordonnées *[0 .. maxY]*. Chaque pixel est repéré par des coordonnées entières, donc chaque pixel a une affixe $a+ib$ avec a dans *[0 .. maxX]* et b dans *[0 .. maxY]*. L'origine étant le coin supérieur gauche de la zone graphique **marges exclues**.

2.4 NewBitmap

- **NewBitmap([fond])**.
- Description: permet de créer un nouveau bitmap (vide). Par défaut la couleur du fond est le blanc.

2.5 Pixel

- **Pixel(<liste d'affixes de pixels>)**.
- Description: permet de d'allumer une <*liste*> de pixels. Cette liste est de la forme : *[affixe, couleur, affixe, couleur, ...]*. Les affixes des pixels sont de la forme $a+ib$ avec a et b des entiers positifs ou nuls, l'origine étant le coin supérieur gauche de la zone graphique **marges exclues**.

2.6 Pixel2Scr

- **Pixel2Scr(<affixe>).**
- Description: permet de convertir l'<*affixe*> d'un pixel (coordonnées entières) en affixe dans le repère de l'utilisateur à l'écran.

2.7 Scr2Pixel

- **Scr2Pixel(<affixe>).**
- Description: permet de convertir l'<*affixe*> d'un point dans le repère de l'utilisateur à l'écran en coordonnées entières (affixe du pixel correspondant au point).
- Exemple(s): un ensemble de Julia, la commande est à placer dans un élément graphique utilisateur. L'image *png* a été obtenue à partir du bouton *snapshot* de l'interface graphique en prenant l'export *bmp* puis une conversion en *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}
```

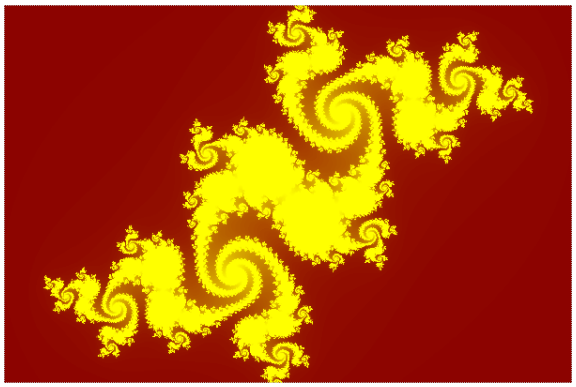


FIGURE 15: *Un ensemble de Julia*

3) Macros graphiques de TeXgraph.mac

3.1 angleD

- **angleD(<B>, <A>, <C>, <r>).**
- Description: dessine l'angle $\widehat{BAC}$ avec un parallèlogramme de coté r.

3.2 Arc

- **Arc(<B>, <A>, <C>, <R> [, sens]).**
- Description: trace un arc de cercle de centre <*A*> et de rayon <*R*>. L'arc est tracé partant de la droite (*AB*) jusqu'à la droite (*AC*), l'argument facultatif <*sens*> indique : le sens trigonométrique si sa valeur est 1 (valeur par défaut), le sens contraire si valeur est -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}
```

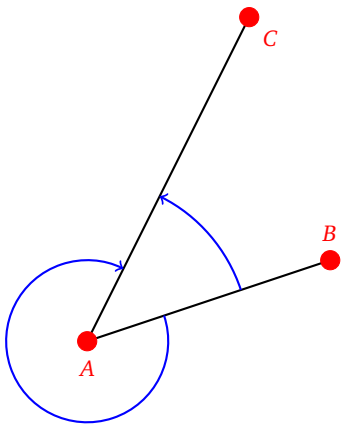


FIGURE 16: *Commande Arc*

3.3 arcBezier

- **arcBezier(<B>, <A>, <C>, <r> [,sens]).**
- Description: a le même effet que la macro graphique *Arc* mais l'arc est dessiné avec des courbes de Bezier.

3.4 axes

- **axes(<[origine, etendueX, etendueY]>, <gradX+i*gradY> [, subdivX+i*subdivY, posOriginX+i*posOriginY, num, "texte", den, firstnum]).**
- Description: pour tracer et graduer les axes passant par <*origine*> (affixe), elle s'utilise comme la commande *Axes* (p. 79) et utilise donc les variables *xylabelpos* et *xyticks*. Par défaut les axes occupent toute la fenêtre si les paramètres optionnels <*etendueX*> et <*etendueY*> sont omis. Le paramètre <*etendueX*> représente sous forme complexe l'intervalle des abcisses : `xmin+i*xmax`, de même pour les ordonnées avec le paramètre <*etendueY*>, les axes sont alors limités à ces intervalles. Remarque : pour préciser une valeur de <*etendueY*> sans préciser de valeur pour <*etendueX*>, il suffit de mettre *jump* à la place de <*etendueX*> (et non pas *Nil* !).
- Le paramètre optionnel <*subdivX+i*subdivY*> indique le nombre de subdivisions par unité sur chaque axe (0 par défaut).
- Le paramètre optionnel <*posOriginX+i*posOriginY*> permet de positionner le label de l'origine :
  - <*posOriginX*>=0 : pas de label à l'origine (idem pour <*posOriginY*>=0),
  - <*posOriginX*>=1 : label normal à l'origine (idem pour <*posOriginY*>=1),,
  - <*posOriginX*>=2 : label décalé à droite de l'origine et en haut pour <*posOriginY*>=2 (valeurs par défaut),
  - <*posOriginX*>=-2 label décalé à gauche de l'origine et en bas pour <*posOriginY*>=-2.
- Sur les 2 axes, chaque label est multiplié par la fraction <*num/den*> (1 par défaut), ajouté à <*firstnum/den*> (l'origine par défaut) et accompagné du <*"texte"*> au numérateur. Cette macro fait appel à la macro *GradDroite* (p. 94), elle utilise donc les variables : *usecomma* (0/1 : pour que le séparateur décimal soit la virgule ou le point), *dollar* (0/1 : pour ajouter ou non des $ autour des labels des graduations), *numericFormat* (0/1/2 : pour gérer le format numérique décimal(0), scientifique(1), ou ingénieur(2)), et *nbdeci* (qui fixe le nombre de décimales affichées).
- Exemple(s): pour avoir des axes gradués en $\pi/2$ en $\pi/2$: `axes(0, pi*(1+i)/2, 1+i, 2+2*i, 1, "\pi", 2, 0)`. Contrairement à la commande Axes, cette macro est sensible aux modifications de la matrice courante. Elle fait appel à la macro *GradDroite* (p. 94) qui utilise les variables *dollar*, *numericFormat* et *nbdeci*.

3.5 axeX

- **axeX(<[origine, posOrigine, etendue]>, <Xpas> [, Subdiv, labelPos , num, ""texte"", den, firstnum]).**
- Description: pour tracer et graduer un axe des abscisses passant par <*origine*> et avec un pas de <*Xpas*>. Le paramètre <*etendue*> représente sous forme complexe l'intervalle des abscisses *xmin+i*xmax*, si celui-ci est omis, alors le tracé occupe la fenêtre. Remarque : pour préciser une valeur de <*etendue*> sans préciser de valeur pour <*posOrigine*>, il suffit de mettre *jump* à la place de <*posOrigine*> (et non pas *Nil* !).

- <*Subdiv*> est le nombre de subdivisions par unité, chaque abscisse est multipliée par la fraction <*num/den*> (1 par défaut), ajoutée à <*firstnum/den*> (l'origine par défaut) et accompagnée du <*"texte"*> au numérateur. Cette macro fait appel à la macro *GradDroite* (p. 94), elle utilise donc les variables : *usecomma* (0/1 : pour que le séparateur décimal soit la virgule ou le point), *dollar* (0/1 : pour ajouter ou non des $ autour des labels des graduations), *numericFormat* (0/1/2 : pour gérer le format numérique décimal(0), scientifique(1), ou ingénieur(2)), et *nbdeci* (qui fixe le nombre de décimales affichées).
- Les paramètres optionnels <*posOrigine*> et <*labelpos*> permettent de positionner les labels :
  - <*posOrigine*>=0 : pas de label à l'origine,
  - <*posOrigine*>=1 : label normal à l'origine,
  - <*posOrigine*>=2 : label décalé à droite à l'origine (valeur par défaut),
  - <*posOrigine*>=-2 label décalé à gauche à l'origine,
  - <*labelPos*>=0 : pas de label du tout,
  - <*Re(labelpos)*>=top : labels au-dessus,
  - Re(<*labelPos*>)=bottom : labels en-dessous (valeur par défaut),
  - Im(<*Im(labelPos*>)=1 : labels orthogonaux à l'axe.

3.6 axeY

- **axeY(<[origine, posOrigine, etendue]>, <Ypas> [, Subdiv, labelPos , num, ""texte"", den, firstnum])**.
- Description: pour tracer et graduer un axe des abscisses passant par <*origine*> et avec un pas de <*Ypas*>. Le paramètre <*etendue*> représente sous forme complexe l'intervalle des ordonnées *ymin+i*ymax*, si celui-ci est omis, alors le tracé occupe la fenêtre. Remarque : pour préciser une valeur de <*etendue*> sans préciser de valeur pour <*posOrigine*>, il suffit de mettre *jump* à la place de <*posOrigine*> (et non pas *Nil* !).
- <*Subdiv*> est le nombre de subdivisions par unité, chaque ordonnée est multipliée par la fraction <*num/den*> (1 par défaut), ajoutée à <*firstnum/den*> (l'origine par défaut) et accompagnée du <*"texte"*> au numérateur. Cette macro fait appel à la macro *GradDroite* (p. 94), elle utilise donc les variables : *usecomma* (0/1 : pour que le séparateur décimal soit la virgule ou le point), *dollar* (0/1 : pour ajouter ou non des $ autour des labels des graduations), *numericFormat* (0/1/2 : pour gérer le format numérique décimal(0), scientifique(1), ou ingénieur(2)), et *nbdeci* (qui fixe le nombre de décimales affichées).
- Les paramètres optionnels <*posOrigine*> et <*labelpos*> permettent de positionner les labels :
  - <*posOrigine*>=0 : pas de label à l'origine,
  - <*posOrigine*>=1 : label normal à l'origine,
  - <*posOrigine*>=2 : label décalé vers le haut à l'origine (valeur par défaut),
  - <*posOrigine*>=-2 label décalé vers le bas à l'origine,
  - <*labelPos*>=0 : pas de label du tout,
  - <*Re(labelpos)*>=left : labels à gauche de l'axe (valeur par défaut),
  - Re(<*labelPos*>=right : labels à droite de l'axe,
  - Im(<*labelPos*>)=1 : labels orthogonaux à l'axe.

```
\begin{texgraph}[name=axeXY, file]
Graph image = [
view(-5, 5, -5, 5), size(7.5),
LabelSize :=footnotesize,
//graphique du haut
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(),
//graphique du bas
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}
```

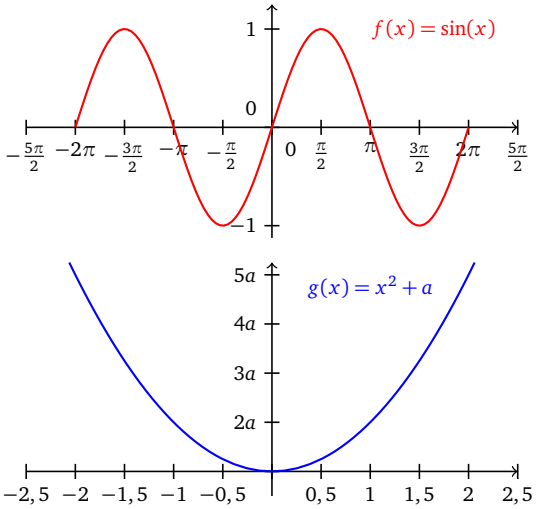


FIGURE 17: *Utilisation de axeX, axeY*

3.7 background

- **background(<fillstyle>, <fillcolor>)**.
- Description: permet de remplir le fond de la fenêtre graphique avec le style et la couleur demandée. Cette macro met à jour la variable *backcolor*.

3.8 bbox

- **bbox()**.
- Description: permet d'ajuster la fenêtre à la "bounding box" autour du dessin courant. Cette macro est destinée à être utilisée dans la ligne de commande en bas de la fenêtre principale, et non pas dans un élément graphique.

3.9 centerView

- **centerView(<affixe>)**.
- Description: permet de centrer la fenêtre graphique sur le point représenté par <*affixe*>, sans changer les dimensions courantes du graphique. Cette macro est plutôt destinée à être utilisée dans la ligne de commande en bas de la fenêtre principale.

3.10 Cercle

- **Cercle(<A>, <r> [, B])**.
- Description: trace un cercle de centre <*A*> et de rayon <*r*> lorsque le troisième paramètre est omis, sinon c'est le cercle défini par les trois points <*A*>, <*r*> et <*B*>.
  Pour les macros *Arc* et *Cercle*, on peut s'attendre à des surprises dans le résultat final si le repère n'est pas orthonormé ! Le repère est orthonormé lorsque les variables *Xscale* et *Yscale* sont égales, voir option *Paramètres/Fenêtre* (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}
```

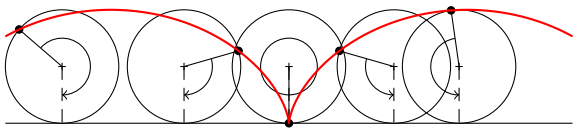


FIGURE 18: *La cycloïde*

3.11 Clip

- **Clip(<liste>)**.
- Description: permet de clipper les éléments graphiques déjà dessinés avec la <*liste*> qui doit être une courbe fermée et simple. La macro peint l'extérieur de la courbe représentée par la <*liste*>.

3.12 Dbissec

- **Dbissec(<B>, <A>, <C>, <1 ou 2>)**.
- Description: dessine la bissectrice de l'angle $\widehat{BAC}$, intérieure si le dernier paramètre vaut 1 et extérieure pour la valeur 2.

3.13 Dcarre

- **Dcarre(<A>, <B>, <+/-1> [, rayon])**.
- Description: dessine de carré de sommets consécutifs <*A*> et <*B*> dans le sens direct si le troisième paramètre vaut 1 (indirect pour -1). Si le paramètre <*rayon*> est présent, alors les « coins » de la figure seront arrondis par un arc de cercle ayant le rayon mentionné.

3.14 Ddroite

- **Ddroite(<A>, <B>)**.
- Description: dessine la demi-droite $[A, B)$.

3.15 Dmed

- **Dmed(<A>, <B> [, angle droit(0/1)])**.
- Description: dessine la médiatrice du segment $[A, B]$. Si le troisième paramètre vaut 1 (0 par défaut) alors un angle droit est dessiné.

3.16 domaine1

- **domaine1(<f(x)> [, a, b])**.
- Description: dessine la partie du plan comprise entre la courbe Cf, l'axe Ox et les droites $x = a$, $x = b$ si a et b sont précisés, sinon $x =$*tMin* et $x =$*tMax*.

3.17 domaine2

- **domaine2(<f(x)>, <g(x)> [, a, b])**.
- Description: dessine la partie du plan comprise entre les courbes Cf, Cg et les droites $x = a$, $x = b$ si a et b sont précisés, sinon $x =$*tMin* et $x =$*tMax*.

3.18 domaine3

- **domaine3(<f(x)>, <g(x)>).**
- Description: délimite la partie du plan comprise entre les courbes Cf et Cg avec x dans l'intervalle *[tMin,tMax]*, en recherchant les points d'intersection.

```
\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,
Cartesienne(sin(x)), tMax :=5,
Cartesienne(2*cos(x)), tMin :=-3, tMax :=3,
Cartesienne(x^2), A :=-4.5+(Ymin+1.75)*i,
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 sur $[-4\,;\, -1.5]$ "),
Inc(A, -0.75*i), FillColor :=blue,
Dcarre(A+0.25*(-1+i), A+0.25*(1+i), -1),
Label(A+0.4, "domaine3 entre $2\cos(x)$ et $x^2$ "),
Inc(A, -0.75*i), FillColor :=red,
Dcarre(A+0.25*(-1+i), A+0.25*(1+i), -1),
Label(A+0.4, "domaine1 sur $[1.1\,;\, 4]$"),
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}
```

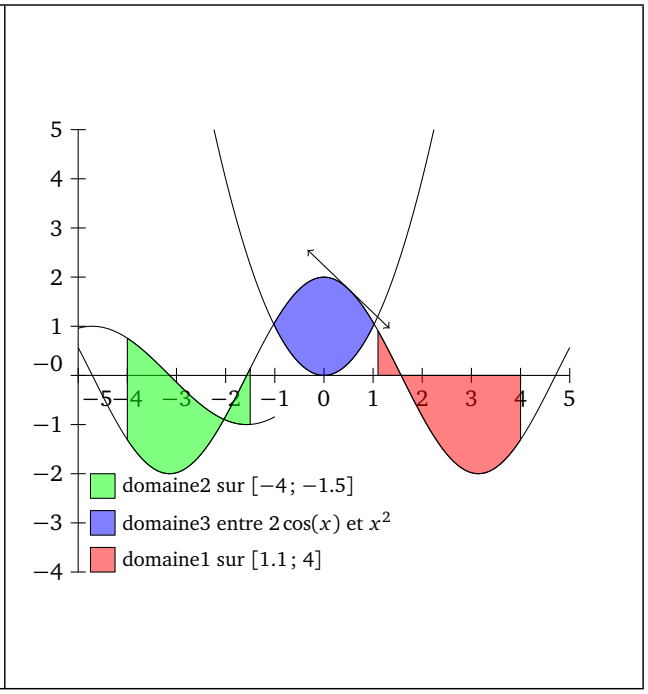


FIGURE 19: *Exemple avec domaine1, 2 et 3*

3.19 Dparallel

- **Dparallel(<[A, B]> , <C>).**
- Description: dessine la parallèle à la droite <*[A,B]*> passant par <*C*>.

3.20 Dparallelo

- **Dparallelo(<A>, <B> , <C> [, rayon]).**
- Description: dessine le parallèlogramme de sommets consécutifs <*A*>, <*B*> et <*C*>. Si le paramètre <*rayon*> est présent, alors les « coins » de la figure seront arrondis par un arc de cercle ayant le rayon mentionné.

3.21 Dperp

- **Dperp(<[A, B]> , <C> [, angle droit(0/1)]).**
- Description: dessine la perpendiculaire à la droite <*[A,B]*> passant par <*C*>. Si le troisième paramètre vaut 1 (0 par défaut) alors un angle droit est dessiné.

3.22 Dpolyreg

- **Dpolyreg(<centre> , <sommet>, <nombre de côtés> [, rayon]).**
- Description: dessine le polygône régulier défini par le <*centre*>, un <*sommet*> et le <*nb de côtés*>. Si le paramètre <*rayon*> est présent, alors les « coins » de la figure seront arrondis par un arc de cercle ayant le rayon mentionné.

ou

- **Dpolyreg(<sommet1>, <sommet2>, <nombre de cotés +sens*i> [, rayon]).**
- Description: dessine le polygône régulier défini par deux sommets consécutifs <*sommet1*> et <*sommet2*>, le <*nb de côtés*>, et le <*sens*> (1 pour direct et −1 pour indirect). Si le paramètre <*rayon*> est présent, alors les « coins » de la figure seront arrondis par un arc de cercle ayant le rayon mentionné.

3.23 DpqGoneReg

- **DpqGoneReg(<centre> , <sommet>, <[p,q]>).**
- Description: dessine le <*p/q*>-gône régulier défini par le <*centre*> et un <*sommet*>.

```
\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}
```

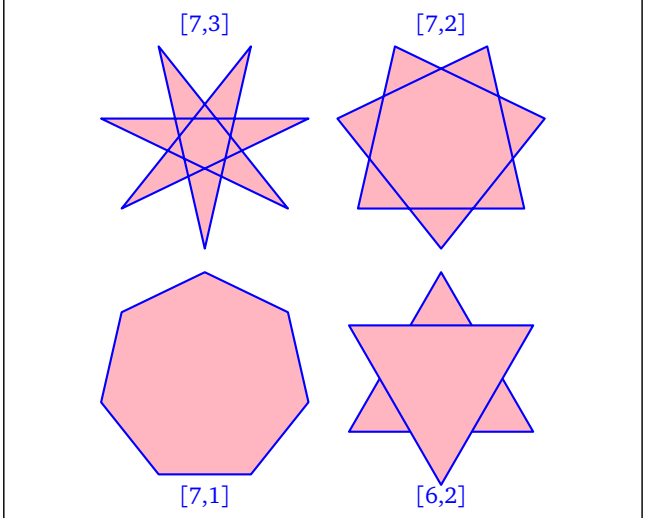


FIGURE 20: *DpqGoneReg : exemple*

3.24 drawSet

- **drawSet(<ensemble>).**
- Description: dessine un ensemble produit par les macros *capB* (p. 72), *cupB* (p. 73) ou *setminusB* (p. 75).

3.25 Drectangle

- **Drectangle(<A>, <B> , <C> [, rayon]).**
- Description: dessine le rectangle de sommets consécutifs <*A*>, <*B*>, le côté opposé passant par <*C*>. Si le paramètre <*rayon*> est présent, alors les « coins » de la figure seront arrondis par un arc de cercle ayant le rayon mentionné.

3.26 ellipticArc

- **ellipticArc(<B>, <A>, <C>, <RX>, <RY>, <sens(+/-1)> [, inclinaison]).**
- Description: dessine un arc d'ellipse de centre <*A*>, allant de <*B*> à <*C*> de rayons <*RX*> et <*RY*>, l'axe portant le rayon <*RX*> ayant une certaine <*inclinaison*> par rapport à l'horizontale, celle-ci est en degrés et vaut 0 par défaut, le paramètre <*sens*> indique le sens de rotation, 1 pour le sens trigonométrique.

3.27 flecher

- **flecher(<liste>, <pos1, ..., posN>).**
- Description: dessine des flèches le long de la ligne polygonale <*liste*>, la position de chaque flèche (pos1, ...) est un nombre entre 0 et 1 (0 pour début de la ligne et 1 pour fin de ligne), les flèches sont dessinées dans le sens de parcourt de la ligne, pour inverser une flèche on ajoute +i à la position.
- Exemple(s): flecher(Get(Cercle(0,3)), [0,0.5])

3.28 GradDroite

- **GradDroite(<[A, origine + i*posOrigine, etendue]>, <[u, unit]>, <hautDiv>, <subdiv> [, poslab, orientation, num, ""texte"", den, firstnum])**
- Description: gradue la droite passant par <*A*> et dirigée par le vecteur <*u*> (celui-ci n'est pas forcément unitaire), <*hautdiv*> est la hauteur des graduations en cm, <*subdiv*> est le nombre de subdivisions par unité.
  Paramètres optionnels :

- <*origin*> : indique la graduation de l'origine <*A*> (0 par défaut),
- <*posOrigin*> : indique la position du label à l'origine <*A*>, plusieurs cas :
  - <*posOrigin*>=0 : pas de label à l'origine,
  - <*posOrigin*>=1 : label normal à l'origine (comme les autres)
  - <*posOrigin*>=2 : label à l'origine décalé dans le sens du vecteur <*u*> (valeur par défaut),
  - <*posOrigin*>=-2 : label à l'origine décalé dans le sens opposé du vecteur <*u*>.
- <*etendue*> : représente l'intervalle des graduations sous forme complexe : *min+i*max*, le tracé de l'axe sera limité à cet intervalle. Si ce paramètre est omis, le tracé occupera la fenêtre.
- <*unit*> : indique le pas des graduations sur l'axe (1 par défaut), cette valeur doit être positive.
- <*poslab*> indique la position des labels par rapport à l'axe, ce paramètre prend les valeurs top ou bottom,
- <*orientation*> : orientation des labels, la valeur *i* signifie que les labels sont orthogonaux à l'axe, sinon l'orientation représente le *LabelStyle* (left, right, top, ...),
- chaque graduation est multipliée par la fraction <*num/den*> (1 par défaut), ajoutée à <*firstnum/den*> (l'origine par défaut) et accompagnée du <*"texte"*> au numérateur. Cette macro utilise les variables : *usecomma* (0/1 : pour que le séparateur décimal soit la virgule ou le point), *dollar* (0/1 : pour ajouter ou non des $ autour des labels des graduations), *numericFormat* (0/1/2 : pour gérer le format numérique décimal(0), scientifique(1), ou ingénieur(2)), *nbdeci* (qui fixe le nombre de décimales affichées) et *maxGrad* (qui fixe le nombre maximal de graduations, celui-ci est de 100 par défaut).

- Exemple(s): GradDroite([0,1+2*i],[1,0.5], xyticks, 1, bottom, i) : signifie que la graduation de l'origine sera 1 avec un label décalé à droite, les graduations iront de 0.5 en 0.5, les labels seront sous l'axe et orthogonaux à l'axe.

3.29 LabelArc

- **LabelArc(<B>, <A>, <C>, <R>, <sens>, <"texte">, [, options]).**
- Description: cette macro dessine un arc de cercle de centre <*A*>, de rayon <*R*> partant de la droite (*AB*) jusqu'à la droite (*AC*), l'argument facultatif <*sens*> indique : le sens trigonométrique si sa valeur est 1 (valeur par défaut), le sens contraire si valeur est -1. La macro ajoute également le <*"texte"*>. Le paramètre <*options*> est une liste (facultative) de la forme *[option1 := valeur1, ..., optionN :=valeurN]*, les options sont :
  - labelpos := ⟨ inside/outside ⟩ : positionnement du label (outside par defaut)),
  - labelsep := ⟨ distance en cm ⟩ : distance du label à l'arc (0.25cm par défaut).
  - rotation := ⟨ nombre ⟩ : angle en degrés que fait le label par rapport à l'horizontale (0 par défaut).

  Il est possible dans la liste des options, de modifier des attributs comme *Color* par exemple.

3.30 LabelAxe

- **LabelAxe(<x ou y>, <affixe>, <label> [, [labelPos,décalage en cm], mark(0/1)]).**
- Description: permet d'ajouter un label sur un des axes <*x ou y*>, <*affixe*> désigne l'affixe du point où se fait l'ajout, <*label*> contient le texte à ajouter. Paramètres optionnels, <*[labelPos, décalage en cm]*> et <*mark*> :
  - Re(<*labelpos*>)=1 signifie en dessous pour Ox et à droite pour Oy (par défaut pour Ox),
  - Re(<*labelpos*>)=2 signifie au dessus pour Ox et à gauche pour Oy (par défaut pour Oy),
  - Im(<*labelpos*>)=-1 signifie un décalage sur la gauche pour Ox, vers le bas pour Oy, si le décalage n'est pas précisé, il vaut 0.25 cm par défaut,
  - Im(<*labelpos*>)=1 signifie un décalage sur la droite pour Ox, vers le haut pour Oy, si le décalage n'est pas précisé, il vaut 0.25 cm par défaut,
  - Im(<*labelpos*>)=0 signifie pas de décalage (valeur par défaut),
  - <*mark*> : indique si le point doit être marqué (dans le dotsyle courant).

3.31 LabelDot

- **LabelDot(<affixe>, <"texte">, <orientation> [, DrawDot, distance]).**
- Description: cette macro affiche un texte à coté du point <*affixe*>. L'orientation peut être "N" pour nord, "NE" pour nord-est ...etc, ou bien une liste de la forme [longueur, direction] où direction est un complexe, dans ce deuxième cas, le paramètre optionnel <*distance*> est ignoré. Le point est également affiché lorsque <*DrawDot*> vaut 1 (0 par défaut) et on peut redéfinir la <*distance*> en cm entre le point et le texte (0.25cm par défaut).

3.32 LabelSeg

- **LabelSeg(<A>, <B>, <"texte">, [, options]).**
- Description: cette macro dessine le segment défini par <*A*> et <*B*>, et ajoute le <*"texte"*>. Le paramètre <*options*> est une liste (facultative) de la forme *[option1 := valeur1, ..., optionN :=valeurN]*, les options sont :
  - labelpos := ⟨ center/top/bottom ⟩ : positionnement du label (center par défaut),
  - labelsep := ⟨ distance en cm ⟩ : distance du label au segment lorsque labelpos vaut top ou bottom (0.25cm par défaut).
  - rotation := ⟨ nombre ⟩ : angle en degrés que fait le label par rapport à l'horizontale (par défaut le label est parallèle au segment).

  Il est possible dans la liste des options, de modifier des attributs comme *Color* par exemple.

3.33 markangle

- **markangle(<B>, <A>, <C>, <r>, <n>, <espacement>, <longueur>).**
- Description: même chose que *markseg* (p. 96) mais pour marquer un arc de cercle.

3.34 markseg

- **markseg(<A>, <B>, <n>, <espacement>, <longueur> [, angle]).**
- Description: marque le segment $[A,B]$ avec <*n*> petits segments, l'<*espacement*> est en unité graphique, et la <*longueur*> en cm. Le paramètre optionnel <*angle*> permet de définir en degré l'angle que feront les marques par rapport à la droite (AB) (45 degrés par défaut).

3.35 periodic

- **periodic(<f(x)>, <a>, <b> [, divisions, discontinuités]).**
- Description: trace la courbe de la fonction périodique définie par $y = f(x)$ sur la période $[a;b]$, puis translate le motif pour couvrir l'intervalle [tMin ; tMax]. Les deux paramètres optionnels sont identiques à ceux des courbes paramétrées (nombre de divisions et discontinuités).

```
\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}
```

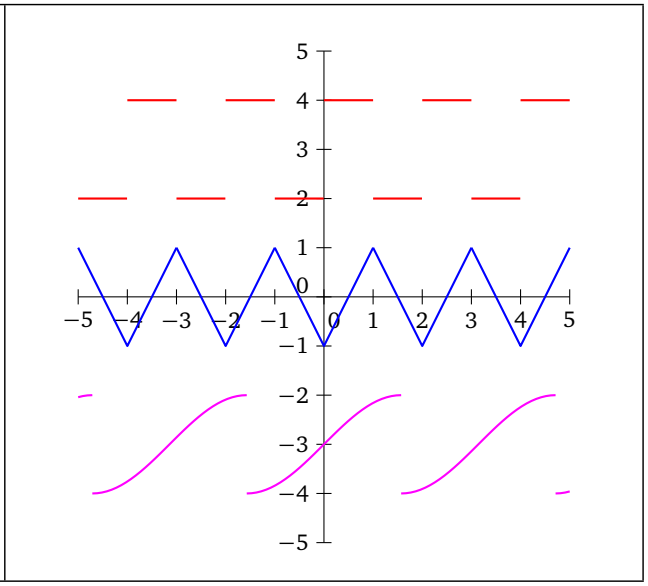


FIGURE 21: *Fonctions périodiques*

3.36 Rarc

- **Rarc(<B>, <A>, <C>, <R>, <sens>).**
- Description: comme la macro *Arc* (p. 88) sauf que l'arc de cercle est rond même si le repère n'est pas orthonormé, le rayon <*R*> est en centimètres.

3.37 Rcercle

- **Rcercle(<A>, <R>)** ou **Rcercle(<A>, <B>, <C>)**.
- Description: dessine un cercle rond même si le repère n'est pas orthonormé. Dans la première forme, le rayon <*R*> est en centimètres.

3.38 Rellipse

- **Rellipse(<O>, <RX>, <RY> [, inclinaison])**.
- Description: comme la commande *Ellipse* (p. 81) sauf que celle-ci est insensible au repère écran, les rayons sont en centimètres.

3.39 RellipticArc

- **RellipticArc(<B>, <A>, <C>, <RX>, <RY>, <sens(+/-1)> [, inclinaison])**.
- Description: comme la macro *ellipticArc* (p. 94) sauf que celle-ci est insensible au repère écran, les rayons sont en centimètres.

3.40 RestoreWin

- **RestoreWin()**.
- Description: restaure la fenêtre graphique ainsi que la matrice 2D enregistrées lors du dernier appel à la macro *SaveWin* (p. 97).

3.41 SaveWin

- **SaveWin()**.
- Description: enregistre la fenêtre graphique ainsi que la matrice 2D courantes, sur une pile. Cette macro va de paire avec la macro *RestoreWin* (p. 97).
- Exemple(s): plusieurs graphiques sur un seul : voir *cet exemple* (p. 91).

3.42 Seg

- **Seg(<A>, <B>)**.
- Description: dessine le segment $[A,B]$.

3.43 set

- **set(<nom>, <affixe centre> [, options])**.
- Description: dessine un ensemble en forme de patatoïde, <*affixe centre*> désigne le centre de cet ensemble, et le paramètre <*nom*> est une chaîne contenant le nom de cet ensemble. Le paramètre <*options*> est une liste (facultative) de la forme *[option1 := valeur1, ..., optionN :=valeurN]*, les options sont :
  - scale := ⟨ entier positif ⟩ : représente l'échelle (1 par défaut),
  - rotation := ⟨ angle en degrés ⟩ : permettant d'incliner le dessin (0 degré par défaut),
  - labels := ⟨ 0/1 ⟩ : pour afficher ou non le nom de l'ensemble.
  - labelsep := ⟨ distance en cm ⟩ : distance du label au bord de l'ensemble (0.45cm par défaut)

  Il est possible dans la liste des options, de modifier des attributs comme LabelStyle par exemple.
- La macro renvoie en résultat la liste des points de la courbe dessinant l'ensemble.

3.44 setB

- **setB(<nom>, <affixe centre> [, options])**.
- Description: dessine un ensemble en forme de patatoïde à l'aide de courbes de Bézier, <*affixe centre*> désigne le centre de cet ensemble, et le paramètre <*nom*> est une chaîne contenant le nom de cet ensemble. Le paramètre <*options*> est une liste (facultative) de la forme *[option1 := valeur1, ..., optionN :=valeurN]*, les options sont :
  - scale := ⟨ entier positif ⟩ : représente l'échelle (1 par défaut),
  - rotation := ⟨ angle en degrés ⟩ : permettant d'incliner le dessin (0 degré par défaut),

– labels := ⟨ 0/1 ⟩ : pour afficher ou non le nom de l'ensemble.

– labelsep := ⟨ distance en cm ⟩ : distance du label au bord de l'ensemble (0.45cm par défaut)
Il est possible dans la liste des options, de modifier des attributs comme LabelStyle par exemple.

- La macro renvoie en résultat la liste des points de contrôle de la courbe représentant l'ensemble. Cette liste peut-être utilisée ensuite pour déterminer une intersection (voir *capB* (p. 72)), une réunion (voir *capB* (p. 72)) ou une différence (voir *setminusB* (p. 75)).

3.45 size

- **size(<largeur + i*hauteur> [, ratio(Xscale/Yscale)])**
- Description: permet de fixer les tailles du graphique : <*largeur*> et <*hauteur*> (marges incluses) en cm. Si le paramètre <*hauteur*> est nul, alors on considère que hauteur=largeur.
Si le paramètre <*ratio*> est omis, les échelles sur les deux axes sont calculées pour que la figure entre au plus juste dans le cadre fixé, tout en conservant le ratio courant.
Si <*ratio*> est égal à 0 alors les échelles sont calculées de manière à obtenir exactement la taille souhaitée (le ratio courant n'est donc vraisemblablement pas conservé).
Le repère est orthonormé lorsque le paramètre <*ratio*> vaut 1.
NB : un appel aux fonctions *Fenetre Marges* ou à la macro *view*, modifiera la taille du graphique. Il est donc préférable de déterminer les marges et la fenêtre graphique **avant** de fixer la taille.
La largeur d'un graphique est donnée par la formule :
*(Xmax-Xmin)*Xscale+margeG+margeD*
et la hauteur est donnée par :
*(Ymax-Ymin)*Yscale+margeH+margeB*

3.46 suite

- **suite(<f(x)>, <u0>, <n>).**
- Description: représentation graphique de la suite définie par $u_{n+1} = f(u_n)$, de premier terme <*u0*> et jusqu'au rang <*n*>. Cette macro ne représente que les "escaliers".

```
\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}
```

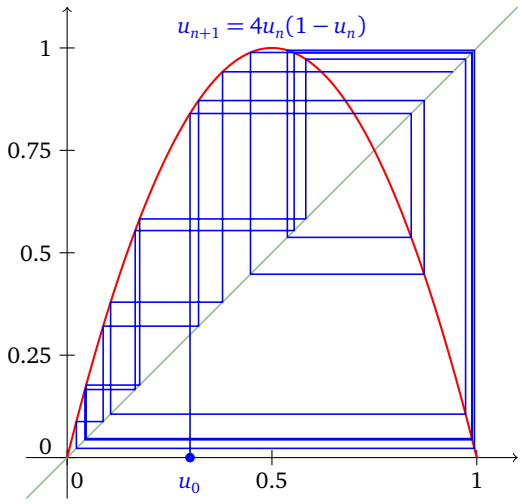


FIGURE 22: *Utilisation de la macro suite*

3.47 tangente

- **tangente(<f(x)>, <x0> [, longueur]).**
- Description: trace la tangente à la courbe cartésienne $y = f(x)$ au point d'abscisse <*x0*>, on trace un segment de la <*longueur*> indiquée (en cm) ou la droite entière si la longueur est omise.

3.48 tangenteP

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

- Description: trace la tangente à la courbe paramétrée par <*f(t)*> au point de paramètre <*t0*>, on trace un segment de la <*longueur*> indiquée (en cm) ou la droite entière si la longueur est omise.

3.49 view

- **view(<xmin>, <xmax>, <ymin>, <ymax>)** ou **view(<[xmin+i*ymin, xmax+i*ymax]>)**
- Description: change la fenêtre graphique courante et conserve l'échelle. Attention : ceci change la taille du graphique, celle-ci peut-être modifiée avec la macro *size* (p. 98).
- Exemple(s): dans un élément graphique utilisateur, la commande [view(-4, 4,-3, 3), size(12)] va fixer la fenêtre graphique à $[-4,4]\times[-3,3]$, et la taille du graphique à 12cm en conservant le ratio courant. Il est important de respecter l'ordre (view avant size).

3.50 wedge

- **wedge(<B>, <A>, <C>, <r>)**
- Description: dessine le secteur angulaire défini par l'angle $\widehat{BAC}$ avec un rayon <*r*>.

3.51 zoom

- **zoom(<+/-1>)**
- Description: permet de faire un zoom arrière/avant.

Chapitre IX

Les macros "spéciales"

1) Macros spéciales

Il s'agit des macros *Init()*, *Exit()*, *Bsave()*, *Esave()*, *TegWrite()*, *ClicGraph*, *ClicG()*, *ClicD()*, *LButtonUp()*, *RButtonUp()*, *MouseMove()*, *MouseWheel()*, *CtrlClicG()*, *CtrlClicD()* et *OnKey()* qui ont un comportement différents des autres macros.

1.1 La macro Init()

Si un fichier source **.teg*, ou un fichier modèle **.mod*, ou un fichier de macros **.mac*, contient une macro intitulée *Init*, alors celle-ci sera automatiquement exécutée dès la fin du chargement du fichier. Cette macro peut être utilisée pour faire certaines initialisations ou par exemple pour demander à l'utilisateur des valeurs.

1.2 La macro Exit()

Si un fichier contient une macro intitulée *Exit*, alors celle-ci est stockée dans une pile lors du chargement du fichier, et sera exécutée lors du prochain changement de fichier, ou lors de la fermeture du programme. Cette macro est surtout destinée à être utilisée dans les fichiers de macros (**.mac*), elle permet par exemple de restituer un contexte dans son état d'origine.

1.3 Les macros Bsave(), Esave() et TegWrite()

La macro **Bsave** est automatiquement exécutée avant l'exportation du graphique en cours, tandis que la macro **Esave** est automatiquement exécutée après l'exportation du graphique en cours.

L'utilisation de ces deux macros est plutôt réservée aux fichiers de macros car il faut tenir compte de leur éventuelle existence avant de les redéfinir. Elles sont d'ailleurs déjà définies dans le fichier *TeXgraph.mac*, la première ne fait qu'appeler la macro *userBsave()*, et la deuxième appelle *userEsave()*. Ces deux dernières n'existent pas, et comme leur nom le suggère, elles peuvent être créées par l'utilisateur dans son fichier source.

La constante *ExportMode* permet de connaître le mode d'exportation, sa valeur peut-être une des constantes suivantes : *tex*, *pgf*, *tkz*, *pst*, *eps*, *psf*, *epsc*, *pdf*, *pdfc*, *svg* ou *teg*.

La macro **TegWrite** est un peu particulière car celle-ci n'est jamais exécutée ! Plus précisément, lors de la sauvegarde du graphique on enregistre successivement :

- La fenêtre.
- Les marges
- La valeur de θ et de φ (pour la 3D).
- Les variables globales.
- Les fichiers de macros à charger.
- Les macros.
- Les éléments graphiques.

Juste avant la sauvegarde des variables globales, on regarde s'il existe une macro appelée *TegWrite*, si c'est le cas, alors la commande définissant cette macro est enregistrée dans le fichier de sauvegarde sous forme d'une commande. Ce qui fait que lors de l'ouverture de ce fichier, cette commande va être exécutée avant la lecture des variables globales et de ce qui suit.

1.4 Les macros ClicG(), ClicD(), LButtonUp(), RButtonUp(), MouseMove(), MouseWheel(), CtrlClicG() et CtrlClicD()

Un clic gauche de la souris provoque automatiquement l'exécution de la macro **ClicG(<affixe>)** avec l'affixe du point cliqué comme paramètre si la touche *Ctrl* n'est pas enfoncée, sinon c'est la macro **CtrlClicG(<affixe>)**. Ces macros, qui

n'existent pas par défaut, peuvent être créées par l'utilisateur.

Lorsque le bouton gauche est relaché cela provoque l'exécution de la macro **LButtonUp(<affixe>)** avec l'affixe du point cliqué comme paramètre. Cette macro, qui n'existe pas par défaut, peut être créée par l'utilisateur.

Un clic droit de la souris provoque automatiquement l'exécution de la macro **ClicD(<affixe>)** avec l'affixe du point cliqué comme paramètre si la touche *Ctrl* n'est pas enfoncée, sinon c'est la macro **CtrlClicD(<affixe>)**. Par défaut, la macro *ClicD(<affixe>)* permet de créer une variable globale.

Lorsque le bouton droit est relâché cela provoque l'exécution de la macro **RButtonUp(<affixe>)** avec l'affixe du point cliqué comme paramètre. Cette macro, qui n'existe pas par défaut, peut être créée par l'utilisateur.

Un déplacement de la souris provoque l'exécution de la macro **MouseMove(<affixe>)** avec l'affixe du point cliqué comme paramètre. Cette macro, qui n'existe pas par défaut, peut être créée par l'utilisateur.

Une rotation de la molette de la souris provoque l'exécution de la macro **MouseWheel(<delta>)** avec *delta* un entier qui est strictement positif si la molette a été poussée vers l'avant, strictement négatif dans le cas contraire. Par défaut, la macro **MouseWheel(<delta>)** permet de faire des zooms avant/arrière sur le graphique.

Exemple(s): construire une ligne polygonale à la souris :

- On crée une variable globale *L* initialisée par exemple à *Nil*.
- On crée un élément graphique *Ligne polygonale* appelé *ligne* et défini par la commande L.
- On crée la macro *ClicG()* avec la commande : [Insert(L, %1), ReCalc(ligne)].
- On crée la macro *ClicD()* avec la commande : [Del(L, -1, 1), ReCalc(ligne)] (cela efface le dernier élément de la liste).

À chaque clic gauche, le point cliqué est ajouté à la liste *L* et la commande ReCalc(ligne) force le recalcul de l'élément graphique *ligne*, on construit ainsi une ligne polygonale à la souris.

1.5 Les macros ClicGraph() et OnKey()

Un clic gauche de la souris sur un élément de la liste des éléments graphiques (en haut à droite) provoque l'exécution de la macro **ClicGraph(<code>)** avec le code de l'élément cliqué, ce code est défini lors de la création de l'élément avec la fonction *NewGraph* (p. 47). Cette macro, qui n'existe pas par défaut, peut être créée par l'utilisateur.

La combinaison de touches `Ctrl+Maj+<lettre>` provoque l'exécution de la macro **OnKey(<lettre>)**, l'argument est une chaîne d'un seul caractère. Cette macro, qui n'existe pas par défaut, peut être créée par l'utilisateur.

2) Les macros spéciales de interface.mac

Ces macros ne sont pas destinées à être utilisées dans des éléments graphiques, mais dans la ligne de commande ou en association avec un bouton ou une option de la liste déroulante de l'interface graphique.

2.1 Apercu

- **Apercu()**.
- Description: création et affichage d'un aperçu A4 à partir d'un export pdf. Cette macro est associée au bouton en forme d'œil dans la barre d'outils : Standard.

2.2 Bouton

- **Bouton(<position>, <nom>, <macro>)**.
- Description: création d'un bouton, la <*position*> est un complexe $x+iy$ avec x et y en pixels, le nom et la macro associée sont deux chaînes de caractères.
- Exemple(s): création (dans la ligne de commande) d'un bouton pour faire un snapshot en png et l'afficher :

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

- Pour supprimer les boutons, voir la commande *DelButton* (p. 38).

2.3 geomview

- **geomview()**.
- Description: permet de visionner dans geomview la scène 3D courante construite avec *Build3d* (p. 139). Cela suppose que ce programme est installé sur votre machine et que son chemin d'accès est connu de votre système.
- Cette macro est associée à un bouton de la barre *Suppléments 3D*.

2.4 help

- **help(<fichier pdf> [, dossier])**.
- Description: permet d'ouvrir un <*fichier pdf*> dans le <*dossier*> indiqué. Le nom du fichier est sans extension, sans chemin et sans guillemets, par exemple : help(TeXgraph) ouvrira le fichier *TeXgraph.pdf* qui est dans le dossier *DocPath*, c'est la valeur par défaut de l'argument <*dossier*>. Autre exemple : help(povray, UserMacPath).

2.5 javaview

- **javaview()**.
- Description: permet de visionner dans javaview la scène 3D courante construite avec *Build3d* (p. 139). Cela suppose d'une part que ce programme *java* est installé sur votre machine, et d'autre part que le chemin d'accès à l'archive *javaview.jar* ait été renseigné dans le fichier de configuration (menu : *Paramètres/Fichier de configuration*, un redémarrage du programme est nécessaire).
- Cette macro est associée à un bouton de la barre *Suppléments 3D*.

2.6 MouseZoom

- **MouseZoom(<+/-1>)**
- Description: permet de faire des zooms avant/arrière sur le graphique. Par défaut, cette macro est associée au mouvement de la molette de la souris (événement MouseWheel).

2.7 NewLabel

- **NewLabel(<affixe>)**.
- Description: création d'un label à l'<*affixe*> indiquée, cette macro ouvre la fenêtre de saisie pour demander le texte du label. Cette macro est destinée initialement à être utilisée dans la macro *ClicG()*.

2.8 NewLabelDot

- **NewLabelDot(<affixe>, <"nom">, <orientation> [, DrawDot, distance])**.
- Description: cette macro crée une variable globale appelée <*"nom"*> et dont la valeur est <*affixe*>. Elle crée également un élément graphique affichant le nom de cette variable à coté du point <*affixe*>. L'orientation peut être "N" pour nord, "NE" pour nord-est ...etc, ou bien une liste de la forme [longueur, direction] où direction est un complexe, dans ce deuxième cas, la paramètre optionnel <*distance*> est ignoré. Le point est également affiché lorsque <*DrawDot*> vaut 1 (valeur par défaut) et on peut redéfinir la <*distance*> en cm entre le point et le texte (0.25cm par défaut). L'élément graphique créé fait appel à la macro *LabelDot* (p. 95).
- Cette macro est associée à un bouton de la barre d'outils : Supplément 2D.

2.9 NewLabelDot3D

- **NewLabelDot3D(<coordonnées>, <"nom">, <orientation> [, DrawDot, distance])**.
- Description: L'argument <*coordonnées*> désigne un point de l'espace, il peut être de la forme $M(x,y,z)$ ou bien $[x+iy,z]$. Cette macro crée une variable globale appelée <*"nom"*> et dont la valeur est <*coordonnées*>. Elle crée également un élément graphique affichant le nom de cette variable à coté du point <*coordonnées*>. L'orientation (dans le plan de l'écran) peut être "N" pour nord, "NE" pour nord-est ...etc, ou bien une liste de la forme [longueur, direction] où direction est un complexe, dans ce deuxième cas, la paramètre optionnel <*distance*> est ignoré. Le point est également affiché lorsque <*DrawDot*> vaut 1 (valeur par défaut) et on peut redéfinir la <*distance*> en cm entre le point et le texte (0.25cm par défaut). L'élément graphique créé fait appel à la macro *LabelDot* (p. 95).
- Cette macro est associée à un bouton de la barre d'outils : Supplément 3D.

2.10 Snapshot

- **Snapshot(<export>, <écran ou imprimante (0 ou 1)>, <"nom"> [, montrer(0/1)])**.
- Description: permet de faire une copie d'écran de la zone graphique, le premier argument précise le type d'<*export*>, celui-ci peut-être : *eps*, *epsc*, *pdf*, *pdfc* ou *bmp*. Le deuxième argument précise la résolution de l'image : 0 pour l'écran (96 dpi) et 1 pour l'imprimante (300 dpi), cet argument est ignoré lorsque l'export choisi est *bmp*. Le troisième argument est une chaîne contenant le <*"nom"*> de l'image avec une extension obligatoire : *png* ou *jpg*, et avec le chemin, par défaut ce chemin sera celui du dossier temporaire de TeXgraph. Le quatrième argument est optionnel, il

permet d'indiquer si la capture doit être affichée ou non à l'écran (1 par défaut). Cette macro fait appel à l'utilitaire *convert*.

- Exemple(s): dans la ligne de commande : Snapshot(epsc, 0, "../capture1.png")
- Cette macro est associée à un bouton de la barre d'outils : Standard.

2.11 VarGlob

- **VarGlob(<affixe>)**.
- Description: permet de définir une variable globale initialisée à <*affixe*>. Par défaut, cette macro est associée au clic droit de la souris.

Chapitre X

Représentation en 3D

Pour être tout à fait honnête, TeXgraph n'est pas un logiciel de dessin en 3D, il travaille en complexe. Cependant, il est possible de lui faire faire un minimum de choses dans l'espace :

- Un **point** ou un **vecteur** de coordonnées (x,y,z) est représenté par la liste : [x+i*y,z] ou encore avec la commande *M* (p. 59) : M(x,y,z). Par exemple l'origine est M(0,0,0) ou encore [0,0], il existe aussi la variable *Origin*. Il est possible d'ajouter ou soustraire deux listes, de les multiplier par un nombre, on peut donc faire des combinaisons linéaires. D'autre part une variable locale ou globale peut contenir une liste de complexes, par conséquent une variable A peut très bien contenir une liste comme [x+i*y,z] représentant ainsi ce que nous appelerons un **point3D** ou un **vecteur3D**.
- Un **plan** est représenté par un de ses points et un vecteur normal, c'est à dire une liste : [point3D, vecteur3D].
- Une **droite** est représentée par un de ses points et un vecteur directeur, c'est à dire une liste : [point3D, vecteur3D].
- Une **facette** est représentée par la liste de ses sommets, cette liste se termine par la constante *jump*. L'ordre des sommets est capital, il définit l'orientation de la facette. Exemple : face := [Origin, M(3,0,0), M(0,3,0), jump].
- Une **surface** ou un **polyèdre** est représenté par une liste de facettes.

Il y a deux types de représentations 3D :

1. La représentation d'**objets individuels** : dans ce cas c'est l'utilisateur qui doit gérer la mise en scène, comme l'ordre d'affichage et les éventuelles intersections par exemple. Ce cas correspond aux options que l'on trouve sur la barre *Supplément 3D* de l'interface graphique. Ce cas est adapté lorsqu'il y a un seul objet ou lorsque la gestion de la scène est très simple. L'avantage de cette méthode est de donner une image légère qui reste vectorielle (pour les cercles, les arcs, ...).
2. La représentation **globale d'une scène** : dans ce cas c'est la commande *Build3D()* (p. 139) qui permet de définir la scène et la commande *Display3D()* (p. 140) qui « calcule » la scène et procède à l'affichage. L'ordre d'affichage et les intersections sont donc déterminés automatiquement. L'inconvénient est que le nombre de facettes ou segments peut exploser donnant ainsi une image lourde, d'autre part on perd l'aspect vectoriel pour certains éléments qui sont alors dessinés par segments (arcs, cercles, ...)

Ce chapitre est consacré au premier type, le second fait l'objet du chapitre suivant.

1) Variables prédéfinies

Variables prédéfinies relatives à la représentation en 3D :

- **theta** et **phi** : utilisées pour les calculs de projections sur le plan de l'écran, elles sont initialisées respectivement à $\pi/6$ et $\pi/3$, la première représente la longitude et la deuxième la colatitude. Elles sont modifiables également par l'intermédiaire d'un bouton dans la barre d'outils.
- **sep3D** : constante initialisée à *Re(jump)-i*, sert de délimiteur pour les éléments graphiques dans la commande *Build3D* (p. 139).
- **AngleStep** : représente le pas angulaire lorsque l'on fait tourner un objet 3D à l'aide des boutons représentant les flèches de direction. Celle-ci est initialisée à $\pi/36$ (soit 5 degrés).
- **Origin** : origine, initialisée à *[0,0]*.
- **vecI** : 1er vecteur de base, initialisé à *[1,0]*.
- **vecJ** : 2ième vecteur de base, initialisé à *[i,0]*.
- **vecK** : 3ième vecteur de base, initialisé à *[0,1]*.
- Pour la fenêtre 3D : **Xinf** (=-5), **Xsup** (=5), **Yinf** (=-5), **Ysup** (=5), **Zinf** (=-5) et **Zsup** (=5).
- **HideStyle** : initialisée à dotted, pour le style de tracé des arêtes cachées,
- **HideWidth** : initialisée à *Nil*, pour l'épaisseur du tracé des arêtes cachées,
- **HideColor** : initialisée à *Nil*, pour la couleur du tracé des arêtes cachées.

2) Commandes relatives à la 3D

2.1 Aretes

- **Aretes(<liste de facettes>)**
- Description: cette fonction renvoie la liste des arêtes de l'objet représenté par la <*liste de facettes*>. Une arête est elle même une liste de la forme : [point3D1, point3D2, jump] et la partie imaginaire de la constante *jump* contient soit la valeur 0 pour une arête cachée, soit la valeur 1 pour une arête visible.
- Exemple(s): section d'un tétraèdre :

```
\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}
```

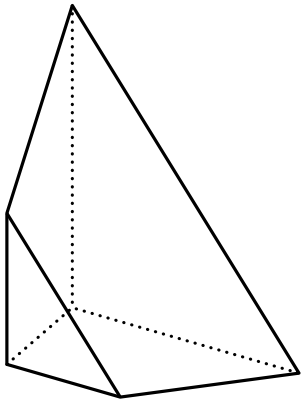


FIGURE 1: *Aretes*

2.2 Bord

- **Bord(<liste de facettes>)**
- Description: cette fonction renvoie la liste des arêtes constituant le bord de l'objet représenté par la <*liste de facettes*>. Une arête est elle même une liste de la forme : [point3D1, point3D2, jump] et la partie imaginaire de la constante *jump* contient soit la valeur 0 pour une arête cachée, soit la valeur 1 pour une arête visible. Une arête est considérée sur le bord lorsqu'elle n'appartient qu'à une seule facette.

2.3 ComposeMatrix3D

- **ComposeMatrix3D(<[vecteur3D1, vecteur3D2, vecteur3D3, vecteur3D4]>)**
- Description: cette fonction compose la matrice <*[vecteur3D1, vecteur3D2, vecteur3D3, vecteur3D4]*> avec la matrice 3D courante (celle-ci affecte la fonction de projection *Proj3D* (p. 110)). Cette matrice représente l'expression analytique d'une application affine de l'espace, c'est une liste de trois vecteurs : *vecteur3D1* qui est le vecteur de translation, *vecteur3D2* qui est le premier vecteur colonne de la matrice de la partie linéaire dans la base canonique, *vecteur3D3* qui est le deuxième vecteur colonne de la matrice de la partie linéaire, et *vecteur3D4* qui est le troisième vecteur colonne de la matrice de la partie linéaire. Par exemple, la matrice de l'identité s'écrit ainsi : [M(0,0,0), M(1,0,0), M(0,1,0], M(0,0,1)] ou encore [Origin, vecI, vecJ, vecK] (c'est la matrice par défaut). (Voir aussi les commandes *GetMatrix3D* (p. 107), *SetMatrix3D* (p. 111), et *IdMatrix3D* (p. 108)).
- Si f est une application affine de l'espace alors sa partie linéaire est Lf(X)=f(X) - f(Origin), le vecteur de translation est f(Origin), et sa matrice s'écrit : [f(Origin), Lf(vecI), Lf(vecJ), Lf(vecK)].

2.4 ConvertToObj

- **ConvertToObj(<liste de facettes>, <sommets>, <facettes>)**
- Description: cette fonction convertit la <*liste de facettes*> au format *obj*, plus précisément les deux derniers arguments doivent être des variables, la variable <*sommets*> reçoit en sortie la liste des sommets (sans doublons) et la variable <*facettes*> reçoit la liste des facettes (séparées par la constante *jump*) comportant non pas les coordonnées des sommets, mais leur numéro d'apparition dans la liste des sommets. La fonction renvoie un complexe $a + ib$ où a est le nombre de sommets et b le nombre de faces. Cette commande est utilisée dans les exports *obj*, *geom* et *jvx*.

Attention : pour un grand nombre de facettes (plusieurs milliers ou plus), cette commande prend un certain temps (compter 2 à 3 mn pour environ 20 000 facettes) !

- La commande *MakePoly* (p. 108) fait l'opération inverse.
- Exemple(s): l'exécution ConvertToObj(Tetra(Origin, 2*vecI, 3*vecJ, vecK), S, F) renvoie la valeur 4+4*i, ce qui signifie 4 sommets et 4 facettes. La variable *S* contient en sortie la liste : [0,0,3*i,0,2,0,0,1], et la variable *F* contient en sortie la liste : [1,2,3,jump,1,3,4,jump,3,2,4,jump,1,4,2,jump].

2.5 ConvertToObjN

- **ConvertToObjN(<liste de facettes>, <sommets>, <facettes>)**
- Description: cette fonction convertit la *<liste de facettes>* au format *obj*, plus précisément les deux derniers arguments doivent être des variables, la variable *<sommets>* reçoit en sortie la liste des sommets (sans doublons) où **chaque sommet est suivi de son vecteur unitaire normal** (ce vecteur est la moyenne des vecteurs normaux aux facettes se partageant le sommet). La variable *<facettes>* reçoit la liste des facettes (séparées par la constante *jump*) comportant non pas les coordonnées des sommets, mais leur numéro d'apparition dans la liste des sommets. La fonction renvoie un complexe $a + ib$ où a est le nombre de sommets et b le nombre de faces. Cette commande est utilisée dans les exports *obj* et *geom*.
  Attention : pour un grand nombre de facettes (plusieurs milliers ou plus), cette commande prend un certain temps !
- Exemple(s): l'exécution de la commande :
  ConvertToObjN(Tetra(Origin,2*vecI,3*vecJ,vecK),S,F)
  renvoie la valeur 4+4*i, ce qui signifie 4 sommets et 4 facettes. La variable *S* contient en sortie la liste :
  [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],
  et la variable *F* contient en sortie la liste : [1, 2, 3, jump, 1, 3, 4, jump, 3, 2, 4, jump, 1, 4, 2, jump].

2.6 Clip3DLine

- **Clip3DLine(<liste de point3D>, <plan>, <fermée(0/1)> [, derrière])**
- Description: cette fonction clippe la liste de points avec le *<plan>*, celui-ci se présente sous la forme d'une liste [point3D, vecteur3D] où le vecteur est normal au plan et point3D un point du plan, la fonction renvoie la partie de la liste contenue dans le demi-espace contenant le vecteur normal (c'est le devant du plan). Le troisième argument précise si la liste doit être fermée ou non. Le dernier argument est facultatif, ce doit être un nom de variable, celle-ci contiendra en sortie la partie de la liste située derrière le plan.
- Exemple(s): couper une hélice :

```
\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}
```

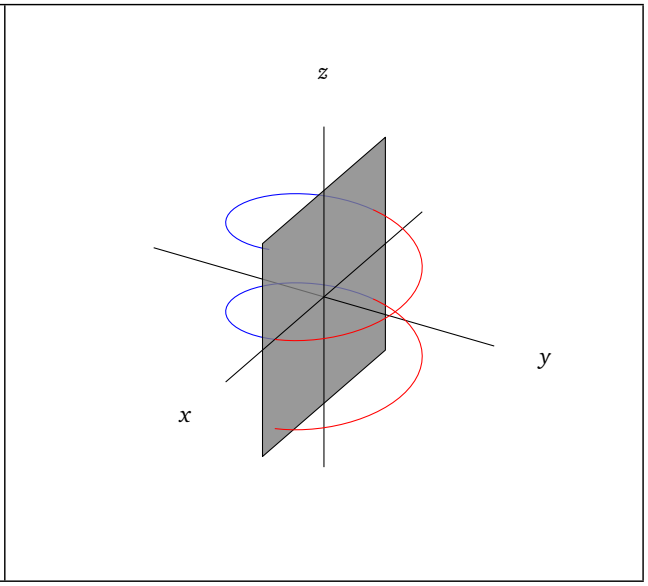


FIGURE 2: *Clip3DLine*

2.7 ClipFacet

- **ClipFacet(<liste de facettes>, <plan>)**

- Description: une facette se présente sous la forme d'une liste de points 3D se terminant par la constante *jump*, ces points sont censés être coplanaires. Exemple : *[Origin, M(0,1,0), M(0,0,3), jump]* est une facette. Les facettes sont orientées par l'ordre d'apparition des sommets.
  Cette fonction coupe toutes les facettes de la liste avec le <*plan*>, celui-ci se présente sous la forme d'une liste du type [A,u] où A est un point3D et u également, cela représente le plan passant par A et normal au vecteur u. Seule la partie des facettes dans le demi-plan contenant u est conservée. La fonction renvoie la liste des facettes coupées.
- Exemple(s): la commande [P :=Tetra(Origin, vecI, vecJ, vecK), ClipFacet(P, [M(0,0,0.5), -vecK])] définit un tétraèdre nommé P et renvoie la partie de P située sous le plan (sous forme de facettes).

```
\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}
```

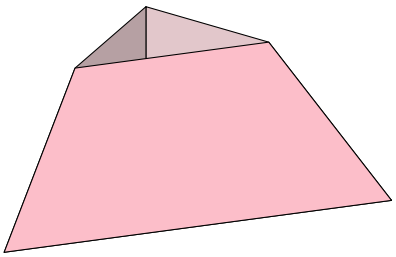


FIGURE 3: *ClipFacet*

2.8 DistCam

- **DistCam(<distance>).**
- Description: permet de changer la position de la caméra en modifiant sa <*distance*> à l'origine. Lorsque cette distance est trop faible, le rendu peut ne pas être correct. Voir aussi *ModelView* (p. 108) et *PosCam* (p. 109).

2.9 Fvisible

- **Fvisible(<facette>)**
- Description: cette fonction renvoie 1 ou 0 suivant que la <*facette*> est visible ou non pour l'observateur. Une facette est visible lorsque son vecteur normal est dirigé vers l'observateur (c'est à dire si le produit scalaire avec le vecteur facette - observateur, est positif). Cette fonction tient compte de la matrice de transformation 3D courante et du type de projection.

2.10 GetMatrix3D

- **GetMatrix3D()**
- Description: cette fonction renvoie la matrice 3D courante. (Voir aussi les commandes *ComposeMatrix3D* (p. 105), *SetMatrix3D* (p. 111), et *IdMatrix3D* (p. 108))

2.11 GetSurface

- **GetSurface(<f(u,v)> [, uMin+i*uMax, vMin+i*vMax, uNbLg+i*vNbLg]).**
- Description: renvoie la liste des facettes de la surface paramétrée par <*f(u,v)*> où f est une fonction de deux variables réelles u et v, et à valeurs dans l'espace. Le deuxième paramètre représente l'intervalle du paramètre u ($[-5,5]$ par défaut), le troisième paramètre représente l'intervalle du paramètre v ($[-5;5]$ par défaut), le quatrième paramètre représente, sous forme complexe, le nombre de lignes pour u et le nombre de lignes pour v (25 lignes par défaut).
- Exemple(s): dessin d'une surface :

```
\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}
```

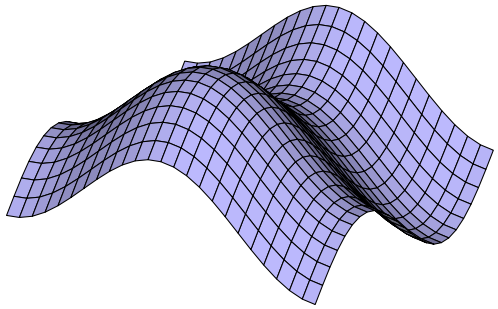


FIGURE 4: *GetSurface*

2.12 IdMatrix3D

- **IdMatrix3D()**
- Description: change la matrice 3D courante en la matrice identité. (Voir aussi les commandes *ComposeMatrix3D* (p. 105), *SetMatrix3D* (p. 111), et *GetMatrix3D* (p. 107))

2.13 Inserer3D

- **Inserer3D(<liste>, <point3D> [, epsilon])**
- Description: le premier argument doit être une variable, la fonction ajoute le <*point3D*> dans la <*liste*> sans qu'il y ait de doublons, et renvoie la position (entier) de ce point dans la variable <*liste*> qui est mise à jour. Le test de comparaison se fait à <*epsilon*> près (0 par défaut).

2.14 MakePoly

- **MakePoly(<liste de points3D>, <liste facettes (format obj)>**
- Description: cette commande prend en entrée une <*liste de points3D*> qui représente des sommets, et une <*liste de facettes*> au format *obj*, c'est à dire que les facettes ne contiennent pas les coordonnées des sommets mais leur numéro d'apparition dans la liste des sommets. La commande renvoie en sortie la liste des facettes construites avec les coordonnées des sommets, cette liste peut alors être dessinée par une des macros *DrawPoly* (p. 137), *DrawFacet* (p. 135).

2.15 ModelView

- **ModelView(<ortho/central>)** ou **ModelView()**.
- Description: permet de modifier le mode de projection *ortho* pour la projection orthographique et *central* pour la projection centrale (voir *Proj3D* (p. 110)). Lorsque l'argument est vide, la fonction renvoie simplement le mode actuel de projection, sinon elle renvoie *Nil*. Voir aussi *PosCam* (p. 109) et *DistCam* (p. 107).

2.16 Mtransform3D

- **Mtransform3D(<liste de points 3D>, <matrice3d>)**
- Description: cette fonction renvoie la <*liste de points 3D*> transformée par la <*matrice3d*>. Cette matrice représente l'expression analytique d'une application affine de l'espace, c'est une liste de trois vecteurs : *vecteur3D1* qui est le vecteur de translation, *vecteur3D2* qui est le premier vecteur colonne de la matrice de la partie linéaire dans la base canonique, *vecteur3D3* qui est le deuxième vecteur colonne de la matrice de la partie linéaire, et *vecteur3D4* qui est le troisième vecteur colonne de la matrice de la partie linéaire. Par exemple, la matrice de l'identité s'écrit ainsi : [M(0,0,0), M(1,0,0), M(0,1,0), M(0,0,1)] ou encore [Origin, vecI, vecJ, vecK] (c'est la matrice par défaut). (Voir aussi les commandes *GetMatrix3D* (p. 107), *ComposeMatrix3D* (p. 105), et *IdMatrix3D* (p. 108)).

```
\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}
```

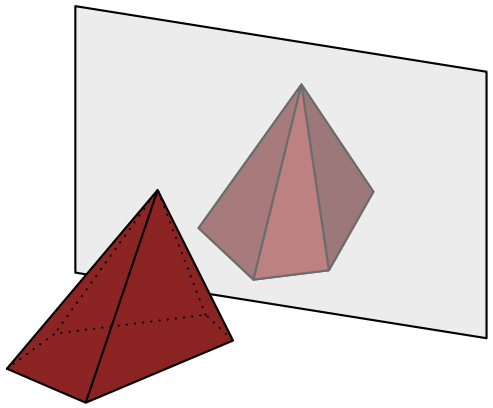


FIGURE 5: *La commande Mtransform3D()*

2.17 Norm

- **Norm(<vecteur3D>)**.
- Description: renvoie la norme du <*vecteur*>.

2.18 Normal

- **Normal()**.
- Description: renvoie le vecteur unitaire normal au plan de projection et dirigé vers l'observateur. Ce vecteur est $M(\sin(\phi)\cos(\theta), \sin(\phi)\sin(\theta), \cos(\phi))$.

2.19 PaintFacet

- **PaintFacet(<liste facettes>, <couleur+i*(non orientées 0/1)>, < (backculling 0/1)+i*contraste>)**.
- Description: cette commande renvoie la <*liste facettes*> après avoir ajouté dans la partie imaginaire de chaque constante *jump* qui sépare les facettes, une <*couleur*> (en réalité c'est la couleur+2). Si l'argument <*non orientées*> vaut 1, alors on ne distingue pas le devant du derrière des facettes. Si l'argument <*backculling*> vaut 1 alors les facettes non visibles sont éliminées. L'argument <*contrast*> est un nombre positif ou nul qui permet d'accentuer ou non le contraste de couleur entre les facettes, avec la valeur 0 la couleur sera unie.
- Cette commande est utilisée par la macro de dessin *DrawFacet* (p. 135).

2.20 PaintVertex

- **PaintVertex(<liste facettes>, <couleur+i*(non orientées 0/1)>, < (backculling 0/1)+i*contraste>)**.
- Description: cette commande renvoie la <*liste facettes*> après avoir ajouté dans la partie imaginaire de la cote de chaque sommet, une <*couleur*> (en réalité c'est la couleur+2). Si l'argument <*non orientées*> vaut 1, alors on ne distingue pas le devant du derrière des facettes. Si l'argument <*backculling*> vaut 1 alors les facettes non visibles sont éliminées. L'argument <*contrast*> est un nombre positif ou nul qui permet d'accentuer ou non le contraste de couleur entre les facettes, avec la valeur 0 la couleur sera unie. L'exécution de cette commande peut être un peu longue pour un grand nombre de facettes.
- Cette commande est utilisée par la macro de dessin *DrawFacet* (p. 135).

2.21 PosCam

- **PostCam(<point3D>)** ou **PostCam()**.
- Description: permet de modifier la position de la caméra. Celle-ci vise toujours l'origine et le plan de projection est le plan passant par l'origine et perpendiculaire à l'axe origine - caméra (c'est le plan de l'écran). Lorsque l'argument est vide, la commande renvoie simplement la position actuelle de la caméra. Voir aussi *ModelView* (p. 108) et *DistCam* (p. 107).

2.22 Prodvec

- **Prodvec(<vecteur3D1>, <vecteur3D2>)**.
- Description: renvoie le résultat du produit vectoriel entre les deux vecteurs.

2.23 Prodscal

- **Prodscal(<vecteur3D1>, <vecteur3D2>)**.
- Description: renvoie le résultat du produit scalaire entre les deux vecteurs.

2.24 Proj3D

- **Proj3D(< liste de point3D >)**.
- Description: cette fonction *Proj3D* calcule et renvoie la liste des projetés des points 3D sur le plan passant par l'origine et normal au vecteur *Normal()* de coordonnées $(sin(\varphi)cos(\theta), sin(\varphi)sin(\theta), cos(\varphi))$ [dirigé vers l'observateur]. La liste de points 3D peut contenir la constante de saut *jump*, elle sera recopiée dans le résultat.

```
\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}
```

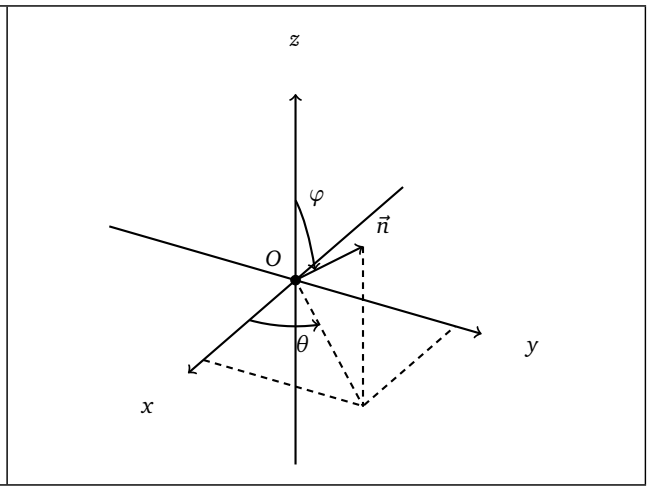


FIGURE 6: *Coordonnées spatiales*

- Il y a deux types de projection : orthographique et centrale. On change de mode avec la commande *ModelView* (p. 108).
  - **projection orthographique** : projection orthogonale sur le plan passant par l'origine et normal au vecteur *Normal()* (ce plan correspond au plan de l'écran). Cela revient à dire que l'observateur est à l'infini. Cette projection a l'avantage d'être linéaire, elle conserve les barycentres, on peut donc dessiner une courbe de BEZIER dans l'espace en utilisant la fonction *Bezier* (p. 80) du plan : si A, B et C sont trois points de l'espace alors on peut créer un élément graphique *Courbe/Bezier* avec la commande Proj3D([A,C,B]) et on verra se dessiner la projection de la courbe de Bézier d'extrémités A et B avec C comme point de contrôle.
  - **projection centrale** : l'observateur est en un certain point C de l'espace (autre que l'origine), le vecteur *Normal()* correspond alors au vecteur $\vec{OC}$ normalisé. La projection se fait toujours sur le plan P passant par l'origine et normal au vecteur *Normal()*, de la manière suivante : le projeté d'un point M est l'intersection de la droite (CM) avec le plan P. Lorsque la distance est trop courte, l'affichage n'est pas toujours correct. Les commandes liées à ce mode de projection sont *PosCam* (p. 109) et *DistCam* (p. 107).
- Exemple(s): représentation dans l'espace d'une courbe plane :

```
\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}
```

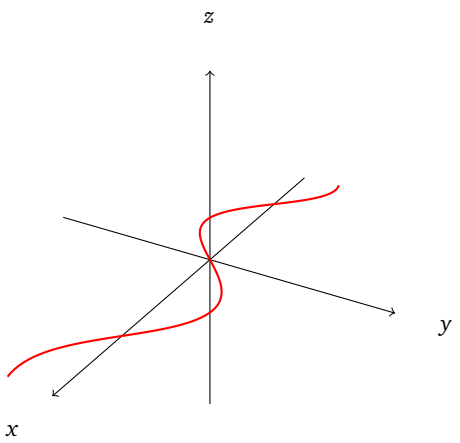


FIGURE 7: *Proj3D*

2.25 ReadObj

- **ReadObj(<"fichier">, <facettes construites>, <lignes construites> [, <sommets>, <facettes obj>, <lignes obj>])**
- Description: cette commande permet de lire un <*"fichier"*> au format *obj* (l'extension est obligatoire). Les arguments suivants doivent être des variables. La variable <*facettes construites*> reçoit la liste des facettes prêtes à être déssinées, de même pour la variable <*lignes construites*>. Les arguments optionnels sont aussi des variables et permettent de récupérer les données du fichier au format *obj* : liste des <*sommets*>, <*facettes obj*> et <*lignes obj*> avec les numéros d'apparition des sommets dans la liste.
- Exemple(s): lecture d'un fichier *triceratops.obj* (chargé à cette adresse :
  `http://www.cs.technion.ac.il/~irit/data/Viewpoint/`
  L'image est obtenue à partir d'une capture (bouton snapshot) avec un export *eps* avant une conversion *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}
```

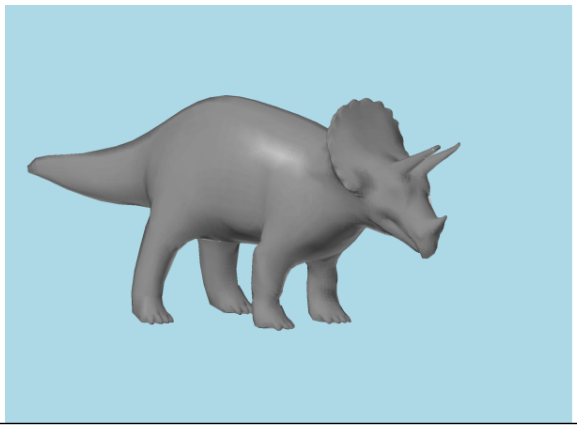


FIGURE 8: *ReadObj*

2.26 SetMatrix3D

- **SetMatrix3D(<[vecteur3D1, vecteur3D2, vecteur3D3, vecteur3D4]>)**
- Description: cette fonction change la matrice courante en <*[vecteur3D1,vecteur3D2,vecteur3D3,vecteur3D4]*> (ceci affecte la fonction de projection *Proj3D* (p. 110)). Cette matrice représente l'expression analytique d'une application affine de l'espace, c'est une liste de trois vecteurs : *vecteur3D1* qui est le vecteur de translation, *vecteur3D2* qui est le premier vecteur colonne de la matrice de la partie linéaire dans la base canonique, *vecteur3D3* qui est le deuxième vecteur colonne de la matrice de la partie linéaire, et *vecteur3D4* qui est le troisième vecteur colonne de la matrice de la partie linéaire. Par exemple, la matrice de l'identité s'écrit ainsi : [M(0,0,0), M(1,0,0), M(0,1,0], M(0,0,1)] ou encore [Origin, vecI, vecJ, vecK] (c'est la matrice par défaut). (Voir aussi les commandes *GetMatrix3D* (p. 107), *ComposeMatrix3D* (p. 105), et *IdMatrix3D* (p. 108)).

- Si f est une application affine de l'espace alors sa partie linéaire est Lf=f-f(Origin), le vecteur de translation est f(Origin), et sa matrice s'écrit : [f(Origin), Lf(vecI), Lf(vecJ), Lf(vecK)].

2.27 Sommets

- **Sommets(< liste de facettes>)**
- Description: cette fonction renvoie la liste des sommets, sans doublons.

2.28 SortFacet

- **SortFacet(<liste de facettes> [, (backculling 0/1)+i*contraste])**
- Description: une facette se présente sous la forme d'une liste de points 3D se terminant par la constante *jump*, ces points sont censés être coplanaires. Exemple : *[Origin, M(0,1,0), M(0,0,3), jump]* est une facette. Les facettes sont orientées par l'ordre d'apparition des sommets.
  Cette fonction classe les facettes de la plus éloignée à la plus proche de l'observateur (c'est la cote du centre de gravité sur l'axe dirigé vers l'observateur qui est pris en compte), et renvoie la liste classée qui en résulte (la liste originale n'est pas modifiée).
  L'argument optionnel est un complexe de la forme (0/1)+i*(0/1).
  Si la partie réelle vaut 1 : les facettes non visibles sont éliminées du tri. Une facette est visible lorsque son vecteur unitaire normal (son sens est déterminé par l'orientation de la facette) est de « même sens » que le vecteur unitaire dirigé vers l'observateur (produit scalaire positif avec le vecteur n()).
  Si la partie réelle vaut 0 : toutes les facettes sont triées.
  Si la partie imaginaire vaut 1 : à chaque facette est attribué un coefficient (produit scalaire entre le vecteur unitaire normal à la facette et *Normal()* qui sert à nuancer la couleur de remplissage lorsque *FillStyle=full*. Ce coefficient est stocké dans la partie imaginaire de la constante *jump* qui termine la facette. La fonction graphique *Ligne* (p. 84) lit ce coefficient, qui est entre 0 et 1 pour une facette visible, et multiplie les composantes rgb de la couleur de remplissage par ce coefficient avant de peindre.
  Si la partie imaginaire vaut 0 : la couleur de remplissage ne sera pas nuancée.
  Par défaut, l'argument optionnel est nul.

3) Les macros mathématiques relatives la 3D

3.1 aire3d

- **aire3d(<liste de facettes convexes>)**.
- Description: renvoie la somme des aires de la <*liste de facettes convexes*>.

3.2 angle3d

- **angle3d(<vecteur3D1>, <vecteur3D2>)**.
- Description: renvoie l'écart angulaire entre les deux vecteurs de l'espace.

3.3 bary3d

- **bary3d(<[point3D1, coef1, point3D2, coef2, ...]>)**.
- Description: renvoie le barycentre du système pondéré <*[(point3D1, coef1), (point3D2, coef2), ...]*>.

3.4 det3d

- **det3d(<vecteur3D1>, <vecteur3D2>, <vecteur3D3>)**.
- Description: renvoie le déterminant des trois vecteurs de l'espace.

3.5 interDD

- **interDD(<droite>, <droite> [, epsilon])**.
- Description: intersection droite-droite. Les droites sont de la forme : [point3D, vecteur directeur]. Si les droites sont coplanaires non parallèles, la macro renvoie un point3D. Par défaut la tolérance <*epsilon*> vaut 1E-10.

3.6 interDP

- **interDP(<droite>, <plan>)**.
- Description: intersection droite-plan. La droite est de la forme : [point3D, vecteur directeur] et le plan de la forme [point3D, vecteur3D normal], la macro renvoie une droite un point3D.

3.7 interLP

- **interLP(<liste de points 3D>, <plan> [, close(0/1)])**.
- Description: cette macro renvoie la liste des points d'intersection entre la ligne polygonale constituée par la <*liste de points 3D*> et le <*plan*>. Le plan est de la forme [point3D, vecteur3D normal]. Le paramètre optionnel <*close*> indique si la ligne doit être refermée ou non (0 par défaut).

3.8 interPP

- **interPP(<plan1>, <plan1>)**.
- Description: intersection plan-plan. Chaque plan est de la forme : [point3D, vecteur3D normal] et la macro renvoie une droite sous la forme d'une liste du type [point3D, vecteur directeur].

3.9 IsAlign3D

- **IsAlign3D(<liste points 3D> [, epsilon])**.
- Description: renvoie 1 si les point3D de la <*liste*> sont alignés, 0 sinon. Par défaut la tolérance <*epsilon*> vaut 1E-10. La <*liste*> ne doit pas contenir la constante *jump*.

3.10 isobar3d

- **isobar3d(<liste point3D>)**.
- Description: renvoie le centre de gravité d'une liste de points de l'espace, la constante *jump* est ignorée.

3.11 IsPlan

- **IsPlan(<liste points 3D> [, epsilon])**.
- Description: renvoie 1 si les point3D de la <*liste*> sont coplanaires, 0 sinon. Par défaut la tolérance <*epsilon*> vaut 1E-10. La <*liste*> ne doit pas contenir la constante *jump*.

3.12 KillDup3D

- **KillDup3D(<liste de points 3D> [, epsilon])**.
- Description: renvoie la <*liste de point3d*> sans doublons, les comparaisons se font à <*epsilon*> près (<*epsilon*> vaut 0 par défaut).

3.13 length3d

- **length3d(<liste point3D> [, fermée(0/1)])**.
- Description: renvoie la longueur de la <*liste point3D*> en unités graphiques, le repère 3D est orthonormé, la <*liste point3D*> peut représenter une liste d'arêtes ou une facette. Par défaut le paramètre <*fermée*> vaut 0.

3.14 Merge3d

- **Merge3d(<liste point3D>)**.
- Description: cette macro permet de recoller des morceaux de listes pour avoir des composantes de longueur maximale, elle renvoie la liste qui en résulte. C'est l'équivalent de la commande *Merge* (p. 45) dans l'espace.

3.15 n

- **n()**.
- Description: macro équivalente à la commande *Normal()* (p. 109). Utilisée en développement immédiat (\n) elle est remplacée par la commande *Normal()*.

3.16 Nops3d

- **Nops3d(<liste point3D>)**.
- Description: renvoie le nombre de point3D de la <*liste*>, en comptant les éventuels *jump*.
- Exemple(s): la commande Nops3d([Origin, jump, 1+i,1, M(1,2,3), jump]) renvoie la valeur 5.

3.17 normalize

- **normalize(<point3D>)**.
- Description: renvoie le vecteur normalisé.

3.18 permute3d

- **permute3d(<liste de point3D>)**.
- Description: modifie la <*liste de point3D*> en plaçant le premier élément 3D (1 point3D = 2 affixes) à la fin, la <*liste de point3D*> doit être une variable. Si le premier élément de cette liste est la constante *jump* alors celle-ci sera déplacée à la fin de la liste (dans ce cas un seul affixe est déplacé).

3.19 planEqn

- **planEqn(<[a,b,c,d]>)**.
- Description: renvoie le plan d'équation $ax + by + cz = d$ sous la forme [point3D, vecteur3D], c'est à dire un point et un vecteur normal.

3.20 Pos3d

- **Pos3d(<point3D>, <liste points 3D> [, epsilon])**.
- Description: renvoie la liste des positions du <*point3D*> dans la <*liste*>, la comparaison se fait à <*epsilon*> près (0 par défaut).
- Exemple(s): la commande Pos3d(M(1,1,0), [Origin, jump, M(1,1,1), M(1,2,3)]) donne la valeur Nil, et Pos3d(M(1,1,1), [Origin, jump, M(1,1,1), M(1,2,3)]) donne la valeur 3.

3.21 purge3d

- **purge3d(<liste point3D> [, epsilon])**.
- Description: renvoie la <*liste point3D*> après avoir supprimer les points consécutifs égaux, supprimer les composantes de cardinal strictement inférieur à 2. Le test d'égalité se fait à <*epsilon*> près, il vaut 1E-10 par défaut.

3.22 px, py, pz, pxy, pxz, pyz

- **px(<point3D>)** : projeté sur Ox.
- **py(<point3D>)** : projeté sur Oy.
- **pz(<point3D>)** : projeté sur Oz.
- **pxy(<point3D>)** : projeté sur xOy.
- **pxz(<point3D>)** : projeté sur xOz.
- **pyz(<point3D>)** : projeté sur yOz.

3.23 replace3d

- **replace3d(<liste de point3D>, <position>, <valeur de remplacement>)**.
- Description: modifie la variable <*liste de points 3D*> en remplaçant l'élément numéro <*position*> par la <*valeur*>, le résultat retourné est *Nil*.
- Exemple(s): si S=[Origin, jump, M(1,1,1), M(1,2,3), jump], alors après la commande replace3d(S,3, [M(1,0,1),M(0,1,1)]), on aura S=[Origin, jump, M(1,0,1),M(0,1,1), M(1,2,3), jump], c'est à dire S=[0,0,jump,1,1,i,1,1+2*i,3,jump].

3.24 reverse3d

- **reverse3d(<liste de point3D>)**.
- Description: renvoie la *<liste de points 3D>* en inversant chacune des composantes de cette *<liste>* (deux composantes sont séparées par un *jump*). Mais la *<liste>* n'est pas modifiée.
- Exemple(s): la commande S :=reverse3d([Origin, M(1,1,0), jump, M(1,1,1), M(1,2,3), jump]) donne S=[M(1,1,0), Origin, jump, M(1,2,3), M(1,1,1), jump], c'est à dire S=[1+i,0,0,0,jump,1+2*i,3,1+i,1,jump].

3.25 viewDir

- **viewDir(<vecteur3D>)** ou **viewDir(<theta>, <phi>)** ou **viewDir(xOy/yOz/xOz)**
- Description: dans la première version, la macro modifie le vecteur normal au plan de projection (voir *n()* (p. 113)) pour qu'il corresponde au *<vecteur3D>* normalisé. Dans la deuxième version, elle modifie les angles de vue *<theta>* et *<phi>*, avec les valeurs fournies, celles-ci doivent être en **degrés**. Dans la troisième version il y a trois arguments possibles : *xOy* ou *yOz* ou *xOz*, ce qui définit le plan de projection.

```
\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}
```

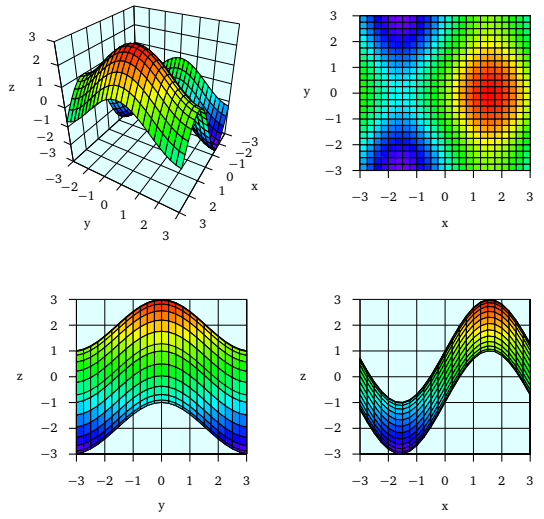


FIGURE 9: *Exemples de vues*

3.26 visible

- **visible(<vecteur3D>)**.
- Description: renvoie 1 si le *<vecteur3D>* est dirigé vers l'observateur (produit scalaire positif).

3.27 Xde, Yde, Zde

- **Xde(<point3D>)** : renvoie l'abscisse.
- **Yde(<point3D>)** : renvoie l'ordonnée.
- **Zde(<point3D>)** : renvoie la cote.

4) Transformations géométriques de l'espace

4.1 antirot3d

- **antirot3d(<liste point3D>, <droite>, <alpha>)**.
- Description: calcule les images de la liste par la rotation d'axe la <*droite*> et d'angle le réel <*alpha*>, composée avec la réflexion par rapport au plan orthogonal à la <*droite*>. La <*droite*> est une liste de la forme : [point3D, vecteur3D directeur], le vecteur directeur oriente la droite, et le plan orthogonal considéré est celui passant par le même point3D.

4.2 defAff3d

- **defAff3d(<nom>, <A>, <A'>, <partie linéaire>)**
- Description: cette fonction permet de créer une macro appelée <*nom*> qui représentera l'application affine qui transforme <*A*> en <*A'*>, et dont la partie linéaire est le dernier argument. Cette partie linéaire se présente sous la forme d'une liste de 3 vecteur3D : [Lf(vecI), Lf(vecJ), Lf(vecK)] où Lf désigne la partie linéaire de la transformation.

4.3 dproj3d

- **dproj3d(<liste point3D>, <droite>)**.
- Description: calcule les images de la liste par la projection orthogonale sur la <*droite*>. La <*droite*> est une liste de la forme : [point3D, vecteur3D directeur].

4.4 dproj3dO

- **dproj3dO(<liste point3D>, <droite>, <vecteur3D normal>)**.
- Description: calcule les images de la liste par la projection oblique sur la <*droite*> et perpendiculairement au <*vecteur normal*>. La <*droite*> est une liste de la forme : [point3D, vecteur3D directeur].

4.5 dsym3d

- **dsym3d(<liste point3D>, <droite>)**.
- Description: calcule les images de la liste par la symétrie orthogonale pa rapport à la <*droite*>. La <*droite*> est une liste de la forme : [point3D, vecteur3D directeur].

4.6 dsym3dO

- **dsym3dO(<liste point3D>, <droite>, <vecteur3D normal>)**.
- Description: calcule les images de la liste par la symétrie oblique par rapport à la <*droite*> et perpendiculairement au <*vecteur normal*>. La <*droite*> est une liste de la forme : [point3D, vecteur3D directeur].

4.7 ftransform3d

- **ftransform3d(<liste point3D>, <f(M)>)**
- Description: renvoie la liste des images des points de <*liste*> par la fonction <*f(M)*>, celle-ci peut-être une expression fonction de M ou une macro d'argument M, M représentant un point3D.

4.8 hom3d

- **hom3d(<liste point3D>, <point3D>, <lambda>)**.
- Description: calcule les images de la liste par l'homothétie de centre <*point3D*> et de rapport le réel <*lambda*>.

4.9 inv3d

- **inv3d(<liste point3D>, <point3D>, <R>)**.
- Description: calcule les images de la liste par l'inversion par rapport à la sphère de centre <*point3D*> et de rayon le réel <*R*>.

4.10 proj3d

- **proj3d(<liste point3D>, <plan>)**.
- Description: calcule la liste des projetés orthogonaux des points de <*liste point3D*> sur le <*plan*>. Le <*plan*> est une liste de la forme : [point3D, vecteur3D normal].

4.11 proj3dO

- **proj3dO(<liste point3D>, <plan>, <vecteur>)**.
- Description: calcule les images de la liste par la projection oblique sur le <*plan*> et parallèlement au <*vecteur*>. Le <*plan*> est une liste de la forme : [point3D, vecteur3D normal].

4.12 rot3d

- **rot3d(<liste point3D>, <droite>, <alpha>)**.
- Description: calcule les images de la liste par la rotation d'axe la <*droite*> et d'angle le réel <*alpha*>. La <*droite*> est une liste de la forme : [point3D, vecteur3D directeur], le vecteur directeur oriente la droite.

4.13 shift3d

- **shift3d(<liste point3D>, <vecteur3D>)**.
- Description: calcule la liste des translatés des points de <*liste point3D*> par le <*vecteur3D*>.

4.14 sym3d

- **sym3d(<liste point3D>, <plan>)**.
- Description: calcule la liste des symétriques orthogonaux des points de <*liste point3D*> par rapport au <*plan*>. Le <*plan*> est une liste de la forme : [point3D, vecteur3D normal].

4.15 sym3dO

- **sym3dO(<liste point3D>, <plan>, <vecteur3D>)**.
- Description: calcule et renvoie la liste des images de la liste par la symétrie oblique par rapport au <*plan*> et parallèlement au <*vecteur3D*>. Le <*plan*> est une liste de la forme : [point3D, vecteur3D normal].

5) Matrices de transformations 3D

Une matrice 3D est une liste de la forme **[vecteur3D1, vecteur3D2, vecteur3D3, vecteur3D4]**. Cette liste représente l'expression analytique d'une application affine de l'espace, c'est une liste de trois vecteurs : **vecteur3D1** qui est le vecteur de translation, **vecteur3D2** qui est le premier vecteur colonne de la matrice de la partie linéaire dans la base canonique, **vecteur3D3** qui est le deuxième vecteur colonne de la matrice de la partie linéaire, et **vecteur3D4** qui est le troisième vecteur colonne de la matrice de la partie linéaire.

Si f est une application affine de l'espace alors sa partie linéaire est *Lf=f-f(Origin)*, le vecteur de translation est *f(Origin)*, et sa matrice s'écrit : *[f(Origin), Lf(vecI), Lf(vecJ), Lf(vecK)]*.

Par exemple, la matrice de l'identité s'écrit ainsi : *[M(0,0,0), M(1,0,0), M(0,1,0), M(0,0,1)]* ou encore *[Origin, vecI, vecJ, vecK]* (c'est la matrice par défaut).

Voir aussi les commandes *ComposeMatrix3D* (p. 105), *GetMatrix3D* (p. 107), *SetMatrix3D* (p. 111) et *IdMatrix3D* (p. 108).

5.1 invmatrix3d

- **invmatrix3d(<[f(0), Lf(vecI), Lf(vecJ), Lf(vecK)]>)**
- Description: renvoie l'inverse de la matrice <*[f(0), Lf(vecI), Lf(vecJ), Lf(vecK)]*>, c'est à dire la matrice :

$$[f^{-1}(0), Lf^{-1}(vecI), Lf^{-1}(vecJ), Lf^{-1}(vecK)]$$

si elle existe.

5.2 matrix3d

- **matrix3d(<fonction affine> [, variable]).**
- Description: renvoie la matrice de la <*fonction affine*>, par défaut la <*variable*> est la lettre *M* (représentant un point3D). Cette matrice se présente sous la forme [f(0), Lf(vecI), Lf(vecJ), L(vecK)], où f désigne l'application affine et Lf sa partie linéaire, (vecI, vecJ, vecK) étant la base canonique.
- Exemple(s): matrix3d(sym3d(M, [Origin,vecK])) renvoie [0,0,1,0,i,0,0,-1], ce qui représente la symétrie orthogonale par rapport au plan xOy.

5.3 mulmatrix3d

- **mulmatrix3d(<matrice3d de f>, <matrice3d de g>)**
- Description: renvoie la matrice de la composée : fog, où f et g sont les deux applications affines de l'espace définies par leur matrice, celle-ci est de la forme [f(0), Lf(vecI), Lf(vecJ), Lf(vecK)] où Lf désigne la partie linéaire.

6) Macros de gestion de la fenêtre 3D

6.1 drawWin3d

- **drawWin3d(<mode>)**
- Description: cette macro dessine la fenêtre 3D courante dans le <*mode*> voulu avec la macro *DrawPoly* (p. 137).

6.2 rectangle3d

- **rectangle3d(<liste point3D>)**
- Description: cette macro détermine le plus petit parallélépipède rectangle contenant la *liste point3D*, cette macro renvoie la grande diagonale de cette boite : *[M(Xinf, Yinf, Zinf), M(Xsup, Ysup, Zsup)]*

6.3 RestoreTphi

- **RestoreTphi()**
- Description: cette macro restaure les valeurs des angles de vue *theta* et *phi* depuis la pile (voir *SaveTphi* (p. 118)).

6.4 RestoreWin3d

- **RestoreWin3d()**
- Description: cette macro restaure la fenêtre 3D et la matrice 3D depuis la pile (voir *SaveWin3d* (p. 118)).

6.5 SaveTphi

- **SaveTphi()**
- Description: cette macro enregistre les valeurs des angles de vue *theta* et *phi*, dans une pile (voir aussi *RestoreTphi* (p. 118)).

6.6 SaveWin3d

- **SaveWin3d()**
- Description: cette macro enregistre la fenêtre 3D actuelle ainsi que la matrice 3D courante dans une pile (voir aussi *RestoreWin3d* (p. 118)).

6.7 transformbox3d

- **transformbox3d(<[M(xinf, yinf, zinf), M(xsup, ysup, zsup)]> [, ortho])**
- Description: cette macro calcule la matrice transformant la boite de grande diagonale <*[M(xinf, yinf, zinf), M(xsup, ysup, zsup)]*> en la boite de grande diagonale $[M(-3,-3,-3), M(3,3,3)]$. Si le paramètre optionnel <*ortho*> vaut 1 (0 par défaut), alors le repère sera orthonormé, cette matrice **est composée avec la matrice 3D courante**, la fenêtre 3D courante est modifiée.

6.8 view3D

- **view3D(<xmin>, <xmax>, <ymin>, <ymax>, <zmin>, <zmax>)** ou **view3D(<[M(xinf, yinf, zinf), M(xsup, ysup, zsup)]>)**
- Description: permet de définir la fenêtre graphique 3D, c'est à dire la valeur des variables *Xinf*, *Xsup*, *Yinf*, *Ysup*, *Zinf* et *Zsup*.

7) Les axes de l'écran et la 3D

L'écran est le plan de projection, il passe par l'origine du repère spatial, le vecteur unitaire normal à ce plan et dirigé vers la caméra est le vecteur désigné par la macro *n()* (p. 113).

7.1 ScreenX

- **ScreenX()**
- Description: cette macro renvoie les coordonnées spatiales du vecteur unitaire de l'axe Ox de l'écran.

7.2 ScreenY

- **ScreenY()**
- Description: cette macro renvoie les coordonnées spatiales du vecteur unitaire de l'axe Oy de l'écran.

7.3 ScreenPos

- **ScreenPos(<affixe> [, distance])**
- Description: cette macro renvoie les coordonnées du point de l'espace se projetant sur l'<*affixe*> donné en argument et à la <*distance*> donnée sur l'axe normal à l'écran (ou l'axe dirigé vers la caméra en projection centrale), cette <*distance*> est facultative et vaut 500 par défaut.

7.4 ScreenCenter

- **ScreenCenter()**
- Description: Cette macro renvoie les coordonnées spatiales du centre de l'écran.

8) Macros de clipping pour la 3D

8.1 Clip3D

- **Clip3D(<liste de facettes>, <polyèdre convexe> [, extérieur(0/1)])**
- Description: cette macro renvoie la <*liste de facettes*> clippées par le <*polyèdre convexe*>. Si le paramètre optionnel *extérieur* vaut 0 (valeur par défaut) c'est la partie intérieure au polyèdre qui est renvoyée, sinon c'est la partie extérieure.

```
\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}
```

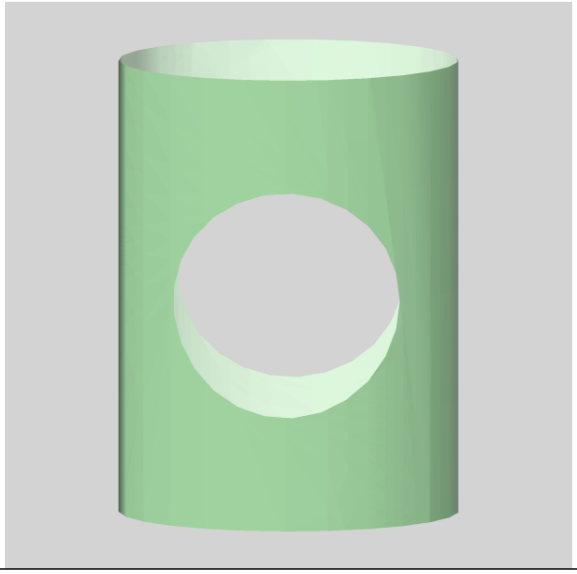


FIGURE 10: *Clip3D*

8.2 clipCurve

- **clipCurve(<liste de point3D> [, fenêtre 3D])**
- Description: cette macro renvoie la *<liste de point3D>* clippées par la *<fenêtre 3D>*, si ce paramètre est absent, c'est la fenêtre 3D courante qui est prise en compte. La *<fenêtre 3D>* est donnée par sa grande diagonale : *[M(xinf, yinf, zinf), M(xsup, ysup, zsup)]*.

8.3 clipPoly

- **clipPoly(<liste de facettes> [, fenêtre 3D])**
- Description: cette macro renvoie la *<liste de facettes>* clippées par la *<fenêtre 3D>*, si ce paramètre est absent, c'est la fenêtre 3D courante qui est prise en compte. La *<fenêtre 3D>* est donnée par sa grande diagonale : *[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}
```

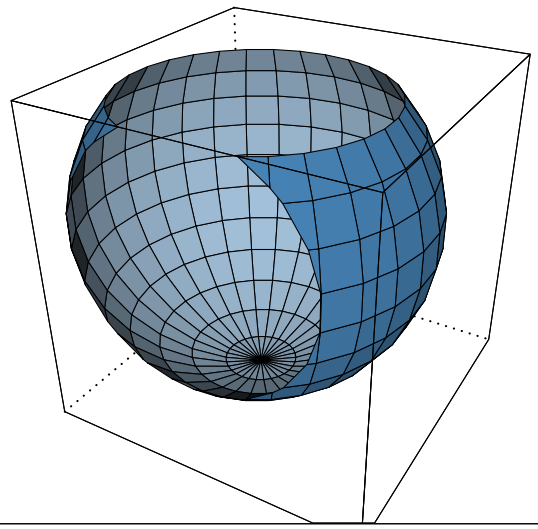


FIGURE 11: *clipPoly*

9) Macros de construction d'objets 3D

9.1 AretesNum

- **AretesNum(<polyèdre>, <liste de numéros>)**

- Description: cette macro renvoie les arêtes du <*polyèdre*> dont les numéros sont dans la <*liste de numéros*>. Les arêtes sont numérotées dans l'ordre de leur apparition.

9.2 Chanfrein

- **Chanfrein(<polyèdre convexe>, <épaisseur> [, épointer(0/1)])**
- Description: renvoie le <*polyèdre convexe*> après l'avoir chanfreiné, pour chaque arête, le solide est sectionné par un plan parallèle au plan bissecteur extérieur aux deux faces adjacentes situé à une distance égale à <*épaisseur*> vers l'intérieur du solide. Le paramètre optionnel <*épointer*> indique si les sommets doivent être épointés ou non (1 par défaut).

```
\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, "non épointé", 1.8-2.6*i, "épointé")
] ;
\end{texgraph}
```

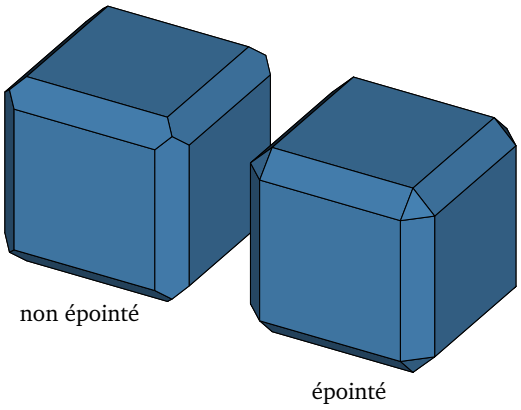


FIGURE 12: *Chanfrein*

9.3 Cone

- **Cone(<point3D>, <vecteur3D>, <rayon> [, nb faces, creux])**
- Description: cette macro renvoie un polyèdre représentant le cône construit à partir d'un <*point3D*> qui est le sommet, d'un <*vecteur3D*> de l'axe qui indique la direction et la hauteur du cône, du <*rayon*> de la face circulaire et du nombre <*nb faces*>, ce dernier vaut 35 par défaut. Le paramètre <*creux*> vaut 0 ou 1 (1 par défaut) et indique si le cône doit est creux ou non, dans la négative la face circulaire est ajoutée aux facettes, c'est la première de la liste.

9.4 curve2Cone

- **curve2Cone(<f(t)>, <tmin>, <tmax>, <sommet>, [, rapport, base])**
- Description: cette macro renvoie sous forme de facettes, le cône partant du <*sommet*> et s'appuyant la courbe gauche paramétrée par $f(t) = [x(t) + i * y(t), z(t)]$ ou $f(t) = M(x(t), y(t), z(t))$. Le paramètre <*rapport*> (nul par défaut) permet de construire l'autre partie du cône par homothétie, le dernier paramètre, <*base*>, est une variable qui contiendra en sortie la liste des points du ou des bords du cône.

```
\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}
```

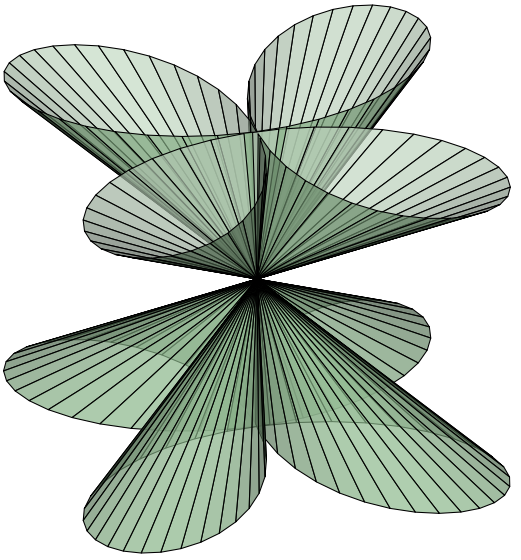


FIGURE 13: *curve2Cone*

9.5 curve2Cylinder

- **curve2Cylinder(<f(t)>, <tmin>, <tmax>, <vecteur3D axe>, [, base])**
- Description: cette macro renvoie sous forme de facettes, le cylindre s'appuyant sur la courbe gauche paramétrée par $f(t) = [x(t) + i * y(t), z(t)]$ ou $f(t) = M(x(t), y(t), z(t))$. Le paramètre <*vecteur3D axe*> détermine de combien la base doit être translatée pour terminer le cylindre. Le dernier paramètre, <*base*>, est une variable qui contiendra la liste des points du ou des bords du cylindre.

```
\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}
```

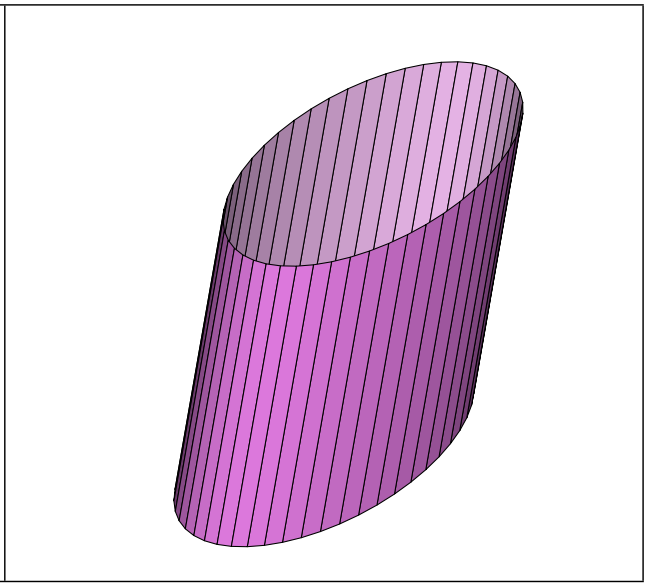


FIGURE 14: *Exemple avec curve2Cylinder*

9.6 curveTube

- **curveTube(<f(t)>, <rayon>, <tmin>, <tmax> [, nb points, nb faces, creux (0/1)])**
- Description: cette macro renvoie sous forme de facettes, un tube centré sur la courbe gauche paramétrée par $f(t) = [x(t) + i * y(t), z(t)]$ ou $f(t) = M(x(t), y(t), z(t))$, de <*rayon*> voulu. Le paramètre <*nb points*> est égal par défaut à la variable globale *NbPoints*. Le paramètre <*nb faces*> vaut 4 par défaut, et le paramètre <*creux*> vaut 1 par défaut.

```
\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}
```

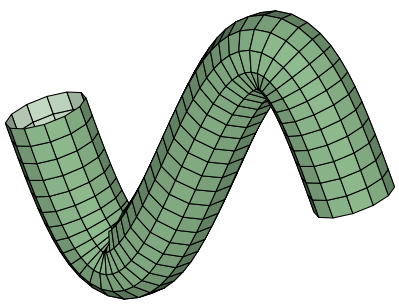


FIGURE 15: *curveTube*

9.7 Cvx3d

- **Cvx3d(<liste de point3D>)**
- Description: renvoie l'enveloppe convexe de la <*liste*> sous forme d'une liste de facettes. La <*liste*> ne doit pas contenir la constante *jump*.

9.8 Cylindre

- **Cylindre(<point3D>, <vecteur3D>, <rayon> [, nb faces, creux])**
- Description: cette macro renvoie un polyèdre représentant le cylindre construit à partir d'un <*point3D*> qui est le centre d'une des deux faces circulaires, d'un <*vecteur3D*> de l'axe qui indique la direction et la hauteur du cylindre, d'un <*rayon*> et du nombre <*nb faces*>, ce dernier vaut 35 par défaut. Le paramètre <*creux*> vaut 0 ou 1 (1 par défaut) et indique si le cylindre est creux ou non, dans la négative les deux faces circulaires sont ajoutées facettes, ce sont les deux premières de la liste.

9.9 FacesNum

- **FacesNum(<polyèdre>, <liste de numéros>)**
- Description: cette macro renvoie les faces du <*polyèdre*> dont les numéros sont dans la <*liste de numéros*>. Les faces sont numérotées dans l'ordre de leur apparition.

9.10 getdroite

- **getdroite(<[point3D,vecteur3D]> [, échelle])**
- Description: cette macro renvoie un segment 3D correspondant à la droite <*[point3D,vecteur3D]*> clippée par la fenêtre 3D courante. La droite est définie sous la forme d'un de ses points et un vecteur directeur. Le paramètre optionnel <*échelle*>, qui vaut 1 par défaut, permet d'agrandir ou diminuer la taille de ce segment (par rapport à son milieu).

9.11 getplan

- **getplan(<[point3D,vecteur3D]> [, échelle])**
- Description: cette macro renvoie une facette correspondant au plan <*[point3D,vecteur3D]*> clippé par la fenêtre 3D courante. Le plan est défini sous la forme d'un de ses points et un vecteur normal. Le paramètre optionnel <*échelle*>, qui vaut 1 par défaut, permet d'agrandir ou diminuer la taille de cette facette.

9.12 getplanEqn

- **getplanEqn(<[a,b,c,d]> [, échelle])**
- Description: cette macro renvoie une facette correspondant au plan <*[a,b,c,d]*> clippé par la fenêtre 3D courante. Le plan est défini sous la forme d'une équation cartésienne $ax + by + cz = d$. Le paramètre optionnel <*échelle*>, qui vaut 1 par défaut, permet d'agrandir ou diminuer la taille de cette facette.

9.13 grille3d

- **grille3d(<x ou y ou z>, <valeur> [, <pas>])**
- Description: cette macro renvoie le plan <*x ou y ou z*> = <*valeur*> sous forme d'une grille (liste de segments). Par défaut le <*pas*> est de 1, mais il peut être de la forme `pas1+i*pas2` si on veut un pas différent sur les deux côtés de la grille ; lorsque `pas2` est nul, on considère qu'il est égal à `pas1`.

```
\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}
```

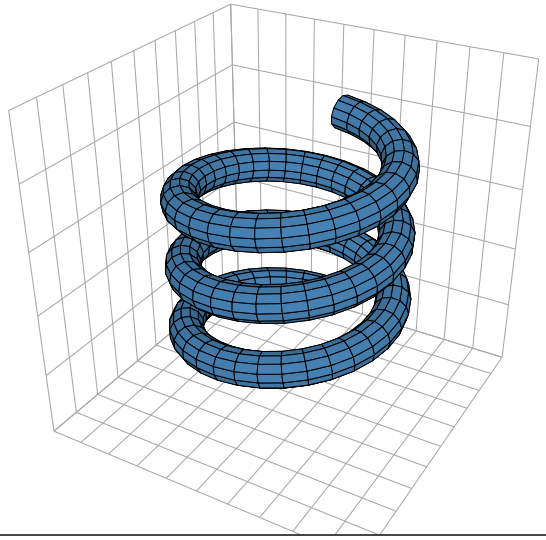


FIGURE 16: *grille3d*

9.14 HollowFacet

- **HollowFacet(<polyèdre> [, épaisseur+i*(mode 0/1), intérieur])**
- Description: cette macro creuse chaque facette du <*polyèdre*> en laissant une <*épaisseur*> au bord (0.25 par défaut), lorsque <*mode*> est nul (valeur par défaut) le découpage est parallèle au bord, lorsque <*mode*> vaut 1, le découpage est . Les morceaux de facettes enlevés sont restitués dans la variable <*intérieur*> si elle est présente.

```
\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}
```

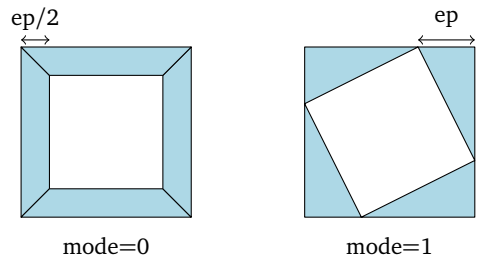


FIGURE 17: *Valeurs de mode (HollowFacet)*

```
\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}
```

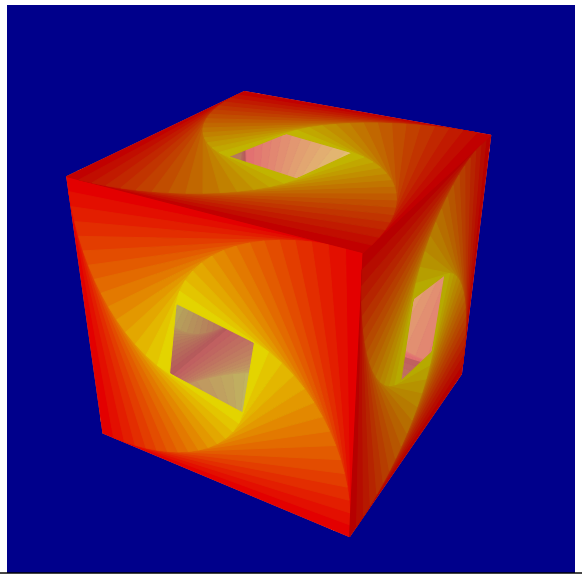


FIGURE 18: *HollowFacet : exemple*

9.15 Intersection

- **Intersection(<plan>, <polyedre>) [, facette])**
- Description: le plan doit être de la forme : $[S,u]$ (plan passant par le point S et normal au vecteur u). La macro détermine l'intersection du <*polyèdre*> avec ce <*plan*> et renvoie celle-ci sous forme d'une *liste d'arêtes* (que l'on peut dessiner avec la macro *DrawAretes* (p. 105)). Il est possible de récupérer l'intersection sous forme d'une <*facette*> en mettant une variable en troisième paramètre.

9.16 line2Cone

- **line2Cone(<ligne 3D>, <sommet>, [, fermée(0/1), rapport, base])**
- Description: cette macro renvoie sous forme de facettes, le cône partant du <*sommet*> et s'appuyant la <*ligne 3D*>, celle-ci ne doit pas contenir la constante *jump*. Le paramètre <*rapport*> (nul par défaut) permet de construire l'autre partie du cône par homothétie, le dernier paramètre, <*base*>, est une variable qui contiendra en sortie la liste des points du ou des bords du cône. L'argument <*fermée*> précise si la ligne doit être refermée ou non (0 par défaut).

9.17 line2Cylinder

- **line2Cylinder(<ligne 3D>, <vecteur3D axe>, [, fermée(0/1), base])**
- Description: cette macro renvoie sous forme de facettes, le cylindre s'appuyant sur la <*ligne 3D*>, celle-ci ne doit pas contenir la constante *jump*. Le paramètre <*vecteur3D axe*> détermine de combien la base doit être translatée pour terminer le cylindre. Le dernier paramètre, <*base*>, est une variable qui contiendra la liste des points du ou des bords du cylindre. L'argument <*fermée*> précise si la ligne doit être refermée ou non (0 par défaut).

9.18 lineTube

- **lineTube(<liste points 3D>, <rayon>, <nb faces> [, fermé, creux])**
- Description: cette macro renvoie sous forme de facettes, un tube centré sur la <*liste points 3D*>, de <*rayon*> indiqué avec le nombre <*nb faces*> voulu. Le paramètre <*fermé*> vaut 0 ou 1 et indique si la ligne doit être fermée (0 par défaut). Le paramètre <*creux*> vaut 0 ou 1 et indique si le tube est creux ou doit être fermé au bout (1 par défaut), ce paramètre n'est pas pris en compte lorsque la ligne est fermée.

```
\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}
```

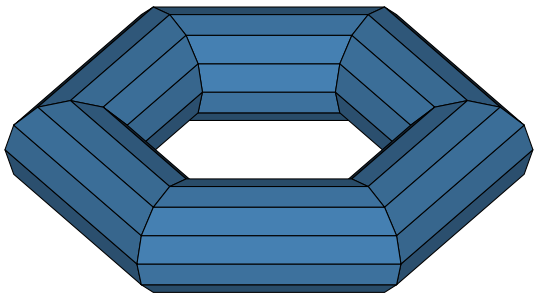


FIGURE 19: *lineTube*

9.19 Parallelep

- **Parallelep(<sommet>, <vecteur3D1>, <vecteur3D2>, <vecteur3D3>)**
- Description: cette macro construit et renvoie la liste des facettes d'un parallélépipède à partir d'un <*sommet*> et de trois vecteurs, supposés dans le sens direct.

9.20 pqGoneReg3D

- **pqGoneReg3D(<axe>, <sommet>, <[p,q]>)**
- Description: cette macro construit et renvoie la liste des points d'un <*p/q*>-gone régulier de l'espace, à partir de son <*axe*> et d'un <*sommet*>. L'axe est une droite de l'espace c'est à dire une liste de la forme : [point3D, vecteur3D], et le sommet est un point3D.

9.21 Prisme

- **Prisme(<base>, <vecteur3D>)**
- Description: cette macro renvoie la liste des facettes d'un prisme à partir d'une <*base*> et d'un <*vecteur3D*> qui représente le vecteur de translation de la base à la face opposée. La base est une liste de point3D coplanaires, cette liste doit être dans le sens direct, le plan étant orienté par le vecteur de translation.

9.22 Pyramide

- **Pyramide(<base>, <sommet>)**
- Description: cette macro construit et renvoie la liste des facettes d'une pyramide construite à partir de sa <*base*> et du <*sommet*>. La base est une liste de point3D coplanaires, cette liste doit être dans le sens direct, le plan étant orienté par le sommet.

9.23 rotCurve

- **rotCurve(<f(t)>, <Axe>, <tmin>, <tmax> [, angleMin, angleMax , tNbpoints, angleNbpoints])**
- Description: cette macro renvoie sous forme de facettes, la surface obtenue en faisant tourner autour de l'<*Axe*>, la courbe gauche paramétrée par $f(t) = [x(t) + i * y(t), z(t)]$ ou $f(t) = M(x(t), y(t), z(t))$. L'argument <*Axe*> est une droite de l'espace déterminée par une liste [point 3D, vecteur3D directeur]. Par défaut on a <*angleMin*>=$-\pi$, <*angleMax*>=π, <*tNbpoints*>=25, et <*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}
```

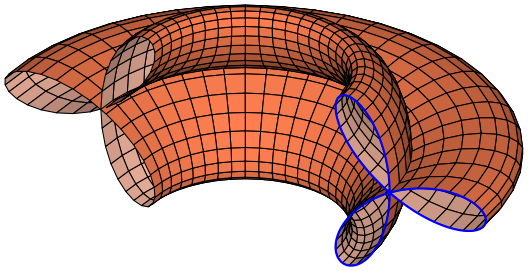


FIGURE 20: *rotCurve*

9.24 rotLine

- **rotLine(<ligne 3D>, <Axe>, [, fermée(0/1), angleMin, angleMax, angleNbpoints])**
- Description: cette macro renvoie sous forme de facettes, la surface obtenue en faisant tourner autour de l'*<Axe>*, la *<ligne 3D>*, celle-ci ne doit pas contenir la constante *jump*. L'argument *<Axe>* est une droite de l'espace déterminée par une liste [point 3D, vecteur3D directeur]. Par défaut on a *<angleMin>*=$-\pi$, *<angleMax>*=π, et *<angleNbpoints>*=25. L'argument *<fermée>* précise si la *<ligne 3D>* doit être referme ou non (0 par défaut).

```
\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}
```

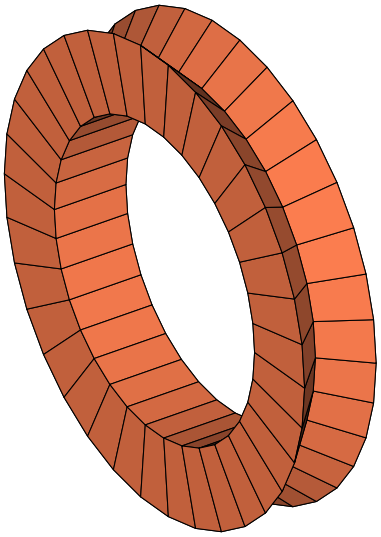


FIGURE 21: *rotLine*

9.25 Section

- **Section(<plan>, <polyèdre>)**
- Description: cette macro permet de découper un *<polyèdre>* avec un *<plan>*. Le plan doit être de la forme : $[S,u]$, cela représente le plan passant par le point S et normal au vecteur u. La macro détermine la section du polyèdre avec ce plan, et la partie du polyèdre qui est dans le demi-espace contenant le vecteur u, est conservée et renvoyée par la macro sous forme d'un polyèdre (liste de facettes).
- Exemple(s): section d'un 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}
```

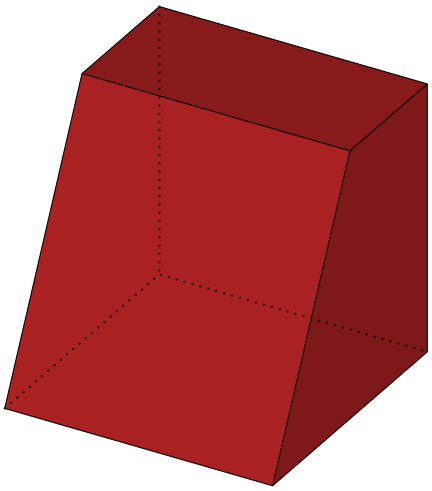


FIGURE 22: *Section*

9.26 Sphere

- **Sphere(<centre>, <rayon> [, nb méridiens, nb parallèles))**
- Description: Cette macro renvoie un polyèdre représentant la sphère construite à partir de son <*centre*> et de son <*rayon*>. Les deux autres paramètres optionnels déterminent le nombre de faces, par défaut de nombre <*nb méridiens*> vaut 40 et le nombre <*nb parallèles*>vaut 25.

9.27 Tetra

- **Tetra(<sommet>, <vecteur3D1>, <vecteur3D2>, <vecteur3D3>)**
- Description: cette macro construit et renvoie la liste des facettes d'un tétraèdre à partir d'un <*sommet*> et trois vecteurs, supposés dans le sens direct.

9.28 trianguler

- **trianguler(<liste de facettes convexes>)**
- Description: cette macro renvoie la <*liste de facettes convexes*> après les avoir triangulées.

10) Les macros de dessin de lignes pour la 3D

10.1 Arc3D

- **Arc3D(<B>, <A>, <C>, <rayon>, <sens>).**
- Description: dessine l'arc de cercle de centre <*A*>, de rayon <*rayon*>, qui joint la droite (AB) et la droite (AC) en restant dans le plan (ABC), dans le sens direct si <*sens*> est strictement positif.

10.2 Axes3D

- **Axes3D(<Ox>, <Oy>, <Oz>, <gradx>, <grady>, <gradz>).**
- Description: trace les axes du repère spatial, on donne les coordonnées de l'origine et le pas des graduations sur les axes (0= aucune graduation).

10.3 AxeX3D

- **AxeX3D(<option1>, <option2>, ...).**
- Description: trace l'axe Ox du repère spatial, cet axe est dirigé par le vecteur *vecI* et passe par un point qui est l'origine par défaut. Les options sont :
  - axeOrigin := ⟨ point3D ⟩ : permet de donner un point de l'axe, par défaut ce point est l'origine : M(0,0,0).

- xlimits := ⟨ [xinf,xsup] ⟩ : définit l'étendue de l'axe, par défaut, c'est l'intervalle [Xinf, Xsup].
- xgradlimits := ⟨ [x1,x2 ⟩ : définit l'étendue des graduations, par défaut c'est la même étendue que *xlimits*.
- xstep := ⟨ nombre ⟩ : définit le pas des graduations : 1 par défaut. Si cette valeur est nulle, alors il n'y aura pas de graduations (ni de labels).
- tickdir := ⟨ vecteur3D ⟩ : indique la direction des graduations, par défaut ce vecteur est *-vecK*.
- tickpos := ⟨ 0..1 ⟩ : indique la position des graduations par rapport à l'axe, par défaut la valeur est 0.5 ce qui signifie que l'axe passe au milieu des graduations.
- labels := ⟨ 0/1 ⟩ : indique si les labels des graduations sont affichés ou non (1 par défaut).
- originlabel := ⟨ 0/1 ⟩ : indique si le label de l'origine est affiché ou non (0 par défaut).
- nbdeci := ⟨ entier ⟩ : nombre de décimales affichées (2 par défaut). Lorsque la variable prédéfinie *usecomma* vaut 1, le point décimal est remplacé par une virgule. Lorsque la variable *dollar* vaut 1, les graduations sont encadrées par le caractère $.
- xlabelstyle := ⟨ left/right/... ⟩ : définit le style de label, la valeur par défaut est celle de *LabelStyle*. Le style ne s'applique pas à la légende.
- xlabelsep := ⟨ distance en cm ⟩ : définit la distance entre l'extrêmité des graduations et les labels (0.25 par défaut).
- newxlegend(<"texte">) : macro qui définit la légende pour l'axe Ox, par défaut le texte est `"$x$"`. Si la chaîne est vide, alors il n'y aura pas de légende.
- xlegendsep := ⟨ distance en cm ⟩ définit la distance entre l'extrémité des graduations et la légende ou l'extrêmité de l'axe suivant la position. Cette distance vaut 0.5 par défaut et s'ajoute à *xlabelsep* quand la légende n'est pas à une extrêmité.
- legendpos := ⟨ 0..1 ⟩ : définit la position de la légende, s'il y en a une. Avec la valeur 0 la légende est à l'extrêmité « inférieure » de l'axe, avec la valeur 1 la légende est à l'extrémité « supérieure » de l'axe, sinon elle est le long de l'axe. par défaut cette valeur est 0.5 (milieu de l'axe).

10.4 AxeY3D

- **AxeY3D(<option1>, <option2>, ...).**
- Description: trace l'axe Oy du repère spatial, cet axe est dirigé par le vecteur *vecJ* et passe par un point qui est l'origine par défaut. Les options sont :
  - axeOrigin := ⟨ point3D ⟩ : permet de donner un point de l'axe, par défaut ce point est l'origine : M(0,0,0).
  - ylimits := ⟨ [yinf,ysup] ⟩ : définit l'étendue de l'axe, par défaut, c'est l'intervalle [Yinf, Ysup].
  - ygradlimits := ⟨ [y1,y2 ⟩ : définit l'étendue des graduations, par défaut c'est la même étendue que *ylimits*.
  - ystep := ⟨ nombre ⟩ : définit le pas des graduations : 1 par défaut. Si cette valeur est nulle, alors il n'y aura pas de graduations (ni de labels).
  - tickdir := ⟨ vecteur3D ⟩ : indique la direction des graduations, par défaut ce vecteur est *-vecK*.
  - tickpos := ⟨ 0..1 ⟩ : indique la position des graduations par rapport à l'axe, par défaut la valeur est 0.5 ce qui signifie que l'axe passe au milieu des graduations.
  - labels := ⟨ 0/1 ⟩ : indique si les labels des graduations sont affichés ou non (1 par défaut).
  - originlabel := ⟨ 0/1 ⟩ : indique si le label de l'origine est affiché ou non (0 par défaut).
  - nbdeci := ⟨ entier ⟩ : nombre de décimales affichées (2 par défaut). Lorsque la variable prédéfinie *usecomma* vaut 1, le point décimal est remplacé par une virgule. Lorsque la variable *dollar* vaut 1, les graduations sont encadrées par le caractère $.
  - ylabelstyle := ⟨ left/right/... ⟩ : définit le style de label, la valeur par défaut est celle de *LabelStyle*. Le style ne s'applique pas à la légende.
  - ylabelsep := ⟨ distance en cm ⟩ : définit la distance entre l'extrémité des graduations et les labels (0.25 par défaut).
  - newylegend(<"texte">) : macro qui définit la légende pour l'axe Oy, par défaut le texte est `"$y$"`. Si la chaîne est vide, alors il n'y aura pas de légende.
  - ylegendsep := ⟨ distance en cm ⟩ définit la distance entre l'extrémité des graduations et la légende ou l'extrêmité de l'axe suivant la position. Cette distance vaut 0.5 par défaut et s'ajoute à *ylabelsep* quand la légende n'est pas à une extrêmité.
  - legendpos := ⟨ 0..1 ⟩ : définit la position de la légende, s'il y en a une. Avec la valeur 0 la légende est à l'extrêmité « inférieure » de l'axe, avec la valeur 1 la légende est à l'extrémité « supérieure » de l'axe, sinon elle est le long de l'axe. par défaut cette valeur est 0.5 (milieu de l'axe).

10.5 AxeZ3D

- **AxeZ3D(<option1>, <option2>, ...).**
- Description: trace l'axe *Oz* du repère spatial, cet axe est dirigé par le vecteur *vecK* et passe par un point qui est l'origine par défaut. Les options sont :
  - axeOrigin := ⟨ point3D ⟩ : permet de donner un point de l'axe, par défaut ce point est l'origine : M(0,0,0).
  - zlimits := ⟨ [zinf,zsup] ⟩ : définit l'étendue de l'axe, par défaut, c'est l'intervalle [Zinf, Zsup].
  - zgradlimits := ⟨ [z1,z2 ⟩ : définit l'étendue des graduations, par défaut c'est la même étendue que *zlimits*.
  - zstep := ⟨ nombre ⟩ : définit le pas des graduations : 1 par défaut. Si cette valeur est nulle, alors il n'y aura pas de graduations (ni de labels).
  - tickdir := ⟨ vecteur3D ⟩ : indique la direction des graduations, par défaut ce vecteur est *-vecJ*.
  - tickpos := ⟨ 0..1 ⟩ : indique la position des graduations par rapport à l'axe, par défaut la valeur est 0.5 ce qui signifie que l'axe passe au milieu des graduations.
  - labels := ⟨ 0/1 ⟩ : indique si les labels des graduations sont affichés ou non (1 par défaut).
  - originlabel := ⟨ 0/1 ⟩ : indique si le label de l'origine est affiché ou non (0 par défaut).
  - nbdeci := ⟨ entier ⟩ : nombre de décimales affichées (2 par défaut). Lorsque la variable prédéfinie *usecomma* vaut 1, le point décimal est remplacé par une virgule. Lorsque la variable *dollar* vaut 1, les graduations sont encadrées par le caractère $.
  - zlabelstyle := ⟨ left/right/... ⟩ : définit le style de label, la valeur par défaut est celle de *LabelStyle*. Le style ne s'applique pas à la légende.
  - zlabelsep := ⟨ distance en cm ⟩ : définit la distance entre l'extrémité des graduations et les labels (0.25 par défaut).
  - newzlegend(<"texte">) : macro qui définit la légende pour l'axe *Oz*, par défaut le texte est `"$z$"`. Si la chaîne est vide, alors il n'y aura pas de légende.
  - zlegendsep := ⟨ distance en cm ⟩ définit la distance entre l'extrémité des graduations et la légende ou l'extrêmité de l'axe suivant la position. Cette distance vaut 0.5 par défaut et s'ajoute à *zlabelsep* quand la légende n'est pas à une extrémité.
  - legendpos := ⟨ 0..1 ⟩ : définit la position de la légende, s'il y en a une. Avec la valeur 0 la légende est à l'extrêmité « inférieure » de l'axe, avec la valeur 1 la légende est à l'extrémité « supérieure » de l'axe, sinon elle est le long de l'axe. par défaut cette valeur est 0.5 (milieu de l'axe).

```
\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}
```

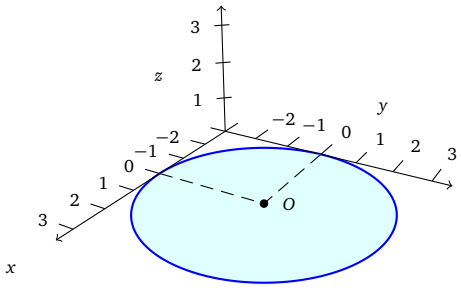


FIGURE 23: *Exemples d'axes*

10.6 BoxAxes3D

- **BoxAxes3D(<option1>, <option2>, ...).**
- Description: trace les trois axes Ox, Oy et Oz du repère spatial sur trois des arêtes de la boite correspondant à la fenêtre 3d courante. Les options sont :
  - labels := ⟨ 0/1 ⟩ : indique si les labels des graduations sont affichés ou non (1 par défaut).

- nbdeci := ⟨ entier ⟩ : nombre de décimales affichées (2 par défaut). Lorsque la variable prédéfinie *usecomma* vaut 1, le point décimal est remplacé par une virgule. Lorsque la variable *dollar* vaut 1, les graduations sont encadrées par le caractère $.
- drawbox := ⟨ 0/1 ⟩ : indique si toutes les arêtes de la boite doivent être dessinées (0 par défaut).
- grid := ⟨ 0/1 ⟩ : indique si une grille doit être dessinée (0 par défaut). Lorsque cette option vaut 1, alors les trois grilles du fond de la boîte sont dessinées. Si la variable *FillStyle* vaut *full* alors elles sont peintes dans la couleur définie par *FillColor*.
- gridcolor := ⟨ couleur ⟩ : couleur de la grille si celle-ci est dessinée (noir par défaut).
- gridwidth := ⟨ épaisseur ⟩ : épaisseur des traits de la grille (2 par défaut).
- xaxe := ⟨ 0/1 ⟩ : indique si l'axe Ox doit être affiché (1 par défaut).
- xlimits := ⟨ [xinf,xsup] ⟩ : définit l'étendue de l'axe, par défaut, c'est l'intervalle [Xinf, Xsup].
- xgradlimits := ⟨ [x1,x2] ⟩ : définit l'étendue des graduations, par défaut c'est la même étendue que *xlimits*.
- xstep := ⟨ nombre ⟩ : définit le pas des graduations : 1 par défaut. Si cette valeur est nulle, alors il n'y aura pas de graduations (ni de labels).
- xlabelstyle := ⟨ left/right/... ⟩ : définit le style de label pour l'axe Ox, la valeur par défaut est celle de *LabelStyle*. Le style ne s'applique pas à la légende.
- xlabelsep := ⟨ distance en cm ⟩ : définit la distance entre l'extrémité des graduations et les labels (0.25 par défaut).
- newxlegend(<"texte">) : macro qui définit la légende pour l'axe Ox, par défaut le texte est `"$x$"`. Si la chaîne est vide, alors il n'y aura pas de légende.
- xlegendsep := ⟨ distance en cm ⟩ : définit la distance entre l'extrémité des graduations et la légende. Cette distance vaut 0.5 par défaut et s'ajoute à *xlabelsep*.
- yaxe := ⟨ 0/1 ⟩ : indique si l'axe Oy doit être affiché (1 par défaut).
- ylimits := ⟨ [yinf,ysup] ⟩ : définit l'étendue de l'axe, par défaut, c'est l'intervalle [Yinf, Ysup].
- ygradlimits := ⟨ [y1,y2] ⟩ : définit l'étendue des graduations, par défaut c'est la même étendue que *ylimits*.
- ystep := ⟨ nombre ⟩ : définit le pas des graduations : 1 par défaut. Si cette valeur est nulle, alors il n'y aura pas de graduations (ni de labels).
- ylabelstyle := ⟨ left/right/... ⟩ : définit le style de label pour l'axe Oy, la valeur par défaut est celle de *LabelStyle*. Le style ne s'applique pas à la légende.
- ylabelsep := ⟨ distance en cm ⟩ : définit la distance entre l'extrémité des graduations et les labels (0.25 par défaut).
- newylegend(<"texte">) : macro qui définit la légende pour l'axe Oy, par défaut le texte est `"$y$"`. Si la chaîne est vide, alors il n'y aura pas de légende.
- ylegendsep := ⟨ distance en cm ⟩ : définit la distance entre l'extrémité des graduations et la légende. Cette distance vaut 0.5 par défaut et s'ajoute à *ylabelsep*.
- zaxe := ⟨ 0/1 ⟩ : indique si l'axe Oz doit être affiché (1 par défaut).
- zlimits := ⟨ [zinf,zsup] ⟩ : définit l'étendue de l'axe, par défaut, c'est l'intervalle [Zinf, Zsup].
- zgradlimits := ⟨ [z1,z2] ⟩ : définit l'étendue des graduations, par défaut c'est la même étendue que *zlimits*.
- zstep := ⟨ nombre ⟩ : définit le pas des graduations : 1 par défaut. Si cette valeur est nulle, alors il n'y aura pas de graduations (ni de labels).
- zlabelstyle := ⟨ left/right/... ⟩ : définit le style de label pour l'axe Oz, la valeur par défaut est celle de *LabelStyle*. Le style ne s'applique pas à la légende.
- zlabelsep := ⟨ distance en cm ⟩ : définit la distance entre l'extrémité des graduations et les labels (0.25 par défaut).
- newzlegend(<"texte">) : macro qui définit la légende pour l'axe Oz, par défaut le texte est `"$z$"`. Si la chaîne est vide, alors il n'y aura pas de légende.
- zlegendsep := ⟨ distance en cm ⟩ : définit la distance entre l'extrémité des graduations et la légende. Cette distance vaut 0.5 par défaut et s'ajoute à *zlabelsep*.

- Exemple(s): voir *ici* (p. 12).

10.7 Cercle3D

- **Cercle3D(<point3D>, <rayon>, <vecteur3D normal>).**
- Description: Dessine un cercle dans l'espace, de centre <*point3D*>, le <*vecteur3D normal*> est normal au plan du cercle et non nul.

10.8 Courbe3D

- **Courbe3D(<f(t)> [, divisions, discontinuités])**
- Description: dessine une courbe gauche paramétrée par <*f(t)*> avec $f(t) = [x(t) + iy(t), z(t]$ ou encore $f((t) = M(x(t), y(t), z(t))$. On peut indiquer le nombre de <*divisions*> par 2 autorisé entre 2 points consécutifs, et la prise en compte des <*discontinuités*> (0 ou 1) comme dans la fonction *Courbe* (p. 81).

10.9 Dcone

- **Dcone(<point3D>, <vecteur3D>, <rayon>, <mode>)**
- Description: dessine un cône à partir de son sommet <*point3D*>, d'un <*vecteur3D*> de l'axe qui indique la direction et la hauteur du cône, et du <*rayon*> de la face circulaire. Le <*mode*> peut valoir :
  - 0 : fil de fer, avec parties cachées,
  - 1 : contour visible uniquement, on peut utiliser le style : FillStyle :=full pour avoir un remplissage.
  - 2 : contour visible (on peut utiliser le style : FillStyle :=full pour avoir un remplissage), auquel on superpose les parties cachées.

  Le tracé des parties cachées utilise les variables *HideStyle*, *HideColor*, *HideWith*.

10.10 Dcylindre

- **Dcylindre(<point3D>, <vecteur3D>, <rayon>, <mode>)**
- Description: dessine un cylindre à partir d'un <*point3D*> qui est le centre d'une des deux faces circulaires, d'un <*vecteur3D*> de l'axe qui indique la direction et la hauteur du cylindre, et d'un rayon r. Le <*mode*> peut valoir :
  - 0 : fil de fer, avec parties cachées,
  - 1 : contour visible uniquement, on peut utiliser le style : FillStyle :=full pour avoir un remplissage.
  - 2 : contour visible (on peut utiliser le style : FillStyle :=full pour avoir un remplissage), auquel on superpose les parties cachées.

  Le tracé des parties cachées utilise les variables *HideStyle*, *HideColor*, *HideWith*.

10.11 DpqGoneReg3D

- **DpqGoneReg3D(<axe>, <sommet>, <[p,q]>)**
- Description: cette macro dessine un <*p/q*>-gone régulier de l'espace, à partir de son <*axe*> et d'un <*sommet*>. L'axe est une droite de l'espace c'est à dire une liste de la forme : [point3D, vecteur3D], et le sommet est un point3D.

10.12 DrawAretes

- **DrawAretes(<liste arêtes> , mode (0/1))**
- Description: dessine une <*liste d'arêtes*>. Une arête est une liste de deux point3D qui se termine par la constante *jump*, la partie imaginaire de celle-ci contient la valeur 0 pour une arête cachée et 1 pour une arête visible (voir la commande *Aretes* (p. 105)). Le <*mode*> peut valoir :
  - 0 : toutes les arêtes sont dessinées,
  - 1 : les arêtes visibles seulement sont dessinées.

  Le tracé des arêtes cachées utilise les variables *HideStyle*, *HideColor*, *HideWith*.

10.13 DrawDdroite

- **DrawDdroite(<droite> [, longueur L])**
- Description: trace une demi-droite [A,A+u) de l'espace, celle-ci est de la forme *[A=point3D, u=vecteur3D directeur]*. S'il n'y a pas d'autre argument, alors la demi-droite est entièrement dessinée. S'il y a le paramètre <*L*>, alors c'est le segment qui relie A à A+L*u/norm(u) qui est dessiné.

10.14 DrawDroite

- **DrawDroite(<droite> [, longueur L1, longueur L2])**
- Description: trace une droite de l'espace, celle-ci est de la forme *[point3D, vecteur3D directeur]*. S'il n'y a pas d'autre argument, alors la droite est entièrement dessinée. S'il y a deux autres paramètres : <*L1*> et <*L2*>, alors si on appelle *A* le point et *u* le vecteur directeur, c'est le segment qui relie A-L1*u/norm(u) à A+L2*u/norm(u) qui est dessiné.

10.15 DrawPlan

- **DrawPlan(<plan>, <vecteur3D>, <longueur1>, <longueur2> [, type])**
- Description: permet de représenter un plan de l'espace, le paramètre <*plan*> est de la forme *[point3D, vecteur3D normal]*, notons A le point et u le vecteur3D normal, le paramètre suivant est un vecteur du plan (notons le v), la macro calcule le produit vectoriel $w = u \wedge v$ et détermine le parallélogramme suivant :

```
\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}
```

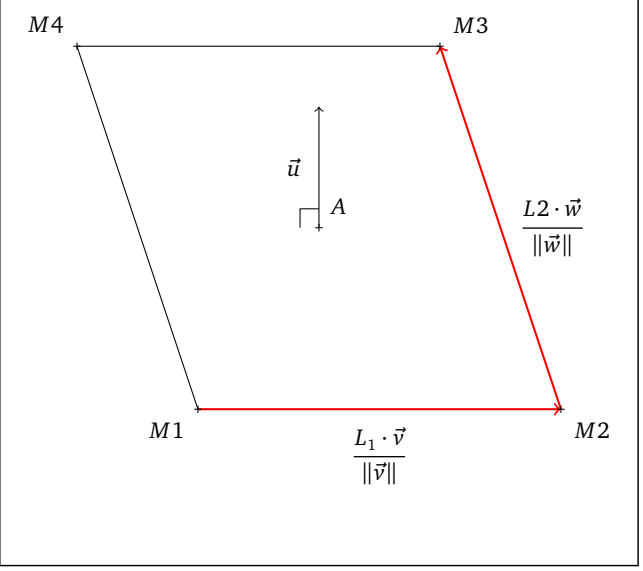


FIGURE 24: *La macro drawplan*

où L1 est le paramètre <*longueur1*> et L2 le paramètre <*longueur2*>. Si le dernier paramètre <*type*> est absent, alors c'est le parallélogramme qui est dessiné, les différentes valeurs possibles sont -1, -2, -3, -4, 1, 2, 3, 4. Ce qui donne (le point A, le vecteur u et l'angle droit ont été ajoutés) :

```
\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(), Model-
View(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, "pas de type")
      ];
\end{texgraph}
```

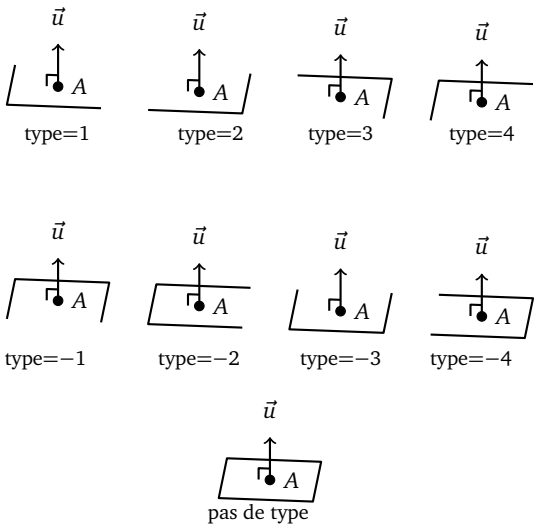


FIGURE 25: *Types de plans*

10.16 Dsphere

labelDsphere

- **Dsphere(<point3D>, <rayon>, <mode>)**
- Description: dessine une sphère à partir de son centre <*point3D*> et de son <*rayon*>. Le <*mode*> peut valoir :
  - 0 : fil de fer, avec parties cachées,
  - 1 : contour visible uniquement, on peut utiliser le style : FillStyle :=full pour avoir un remplissage.
  - 2 : contour visible (on peut utiliser le style : FillStyle :=full pour avoir un remplissage), auquel on superpose les parties cachées.

  Le tracé des parties cachées utilise les variables *HideStyle*, *HideColor*, *HideWith*.

10.17 LabelDot3D

- **LabelDot3D(<point3D>, <"texte">, <orientation> [, DrawDot, distance]).**
- Description: cette macro affiche un texte à coté du point <*point3D*>. Les trois paramètres suivants s'appliquent à la projection du point sur le plan de l'écran. L'orientation peut être "N" pour nord, "NE" pour nord-est ...etc, ou bien une liste de la forme [longueur, direction] où direction est un complexe, dans ce deuxième cas, le paramètre optionnel <*distance*> est ignoré. Le point est également affiché lorsque <*DrawDot*> vaut 1 (0 par défaut) et on peut redéfinir la <*distance*> en cm entre le point et le texte (0.25cm par défaut).

10.18 Ligne3D

- **Ligne3D(<liste de point3D>, <fermée>)**
- Description: dessine une ligne polygonale dans l'espace, la <*liste de point3D*> peut contenir la constante *jump*. Le paramètre <*fermée*> vaut 0 ou 1 et indique si la courbe doit être fermée (1=fermée).

10.19 markseg3d

- **markseg3d(<point3D1>, <point3D2>, <n>, <espacement>, <longueur> [, angle])**.
- Description: marque le segment défini par <*point3D1*> et <*point3D2*> avec <*n*> petits traits, l'<*espacement*> est en unité graphique, et la <*longueur*> en cm. Le paramètre optionnel <*angle*> permet de définir en degrés l'angle que feront les marques par rapport au segment (45 degrés par défaut).

10.20 Point3D

- **Point3D(<liste de point3D>)**
- Description: identique à la commande *Point* (p. 85), mais avec des points de l'espace.

11) Les macros de dessin de facettes pour la 3D

Ces macros se chargent de l'affichage d'objets à facettes basé sur un tri en fonction de l'éloignement du centre de gravité des facettes à l'observateur. Cette méthode ne donne pas toujours le résultat escompté, notamment en le cas de « grandes » facettes.

11.1 Dparallelep

- **Dparallelep(<sommet>, <vecteur3D1>, <vecteur3D2>, <vecteur3D3> [, mode, contrast])**
- Description: cette macro dessine un parallélépipède à partir d'un <*sommet*> et de trois vecteurs, supposés dans le sens direct. Cette macro utilise *DrawPoly* (p. 137) pour dessiner dans le <*mode*> et avec le <*contrast*> voulus.

11.2 Dprisme

- **Dprisme(<base>, <vecteur3D> [, mode, contraste])**
- Description: cette macro dessine un prisme à partir d'une <*base*> et d'un <*vecteur3D*> qui représente le vecteur de translation de la base à la face opposée. La base est une liste de point3D coplanaires, cette liste doit être dans le sens direct, le plan étant orienté par le vecteur de translation. Cette macro utilise *DrawPoly* (p. 137) pour dessiner dans le <*mode*> et avec le <*contraste*> voulus.

11.3 Dpyramide

- **Dpyramide(<base>, <sommet> [, mode, contraste])**
- Description: cette macro dessine une pyramide à partir de sa <*base*> et du <*sommet*>. La base est une liste de point3D coplanaires, cette liste doit être dans le sens direct, le plan étant orienté par le sommet. Cette macro utilise *DrawPoly* (p. 137) pour dessiner dans le <*mode*> et avec le <*contraste*> voulus.

11.4 DrawFacet

- **DrawFacet(facettes1, [options1], facettes2, [options2], ...)**
- Description: cette macro trie l'ensemble de toutes les facettes et les affiche en fonction de leurs options avec la possibilité de faire un lissage (de GOURAUD) ou non, mais **les éventuelles intersections de sont pas gérées**. Les options possibles sont :
  - backculling := ⟨ 0/1 ⟩. Indique si les facettes non visibles doivent être éliminées ou non (0 par défaut).
  - color := ⟨ couleur ⟩. Choix de la couleur (white par défaut).
  - contrast := ⟨ nombre positif ⟩. Le contraste normal a la valeur 1 (valeur par défaut), un contraste nul signifie que la couleur est unie. Ce nombre permet de faire varier le contraste entre les couleurs des facettes d'une même liste.
  - smooth := ⟨ 0/1 ⟩. Indique si l'algorithme de GOURAUD (lissage des facettes) doit être utilisé ou non lors de l'exportation *pstricks* ou *eps* (0 par défaut).

- Les options par défaut ne sont pas réinitialisées entre <*facettes1*> et <*facettes2*> (idem pour les suivantes), ainsi par défaut, les options de <*facettes2*> sont les mêmes que celles de <*facettes1*>. Si les options sont identiques, on peut remplacer <*facettes1*> par <*[facettes1,facettes2]*>, ou bien mettre une liste vide ([]) pour <*options2*>.
- Lorsqu'il n'y a pas de lissage du tout, la macro *DrawFlatFacet* (p. 136) est un peu plus performante. S'il y a beaucoup de lissages (ou bien que des lissages) à effectuer sur grand nombre de facettes, le rendu écran peut-être très long et la commande *draw("SmoothFacet",...)* (p. 137) est alors préférable, car cette dernière n'effectue le lissage qu'au moment de l'export et non pas dès son exécution.

```
\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}
```

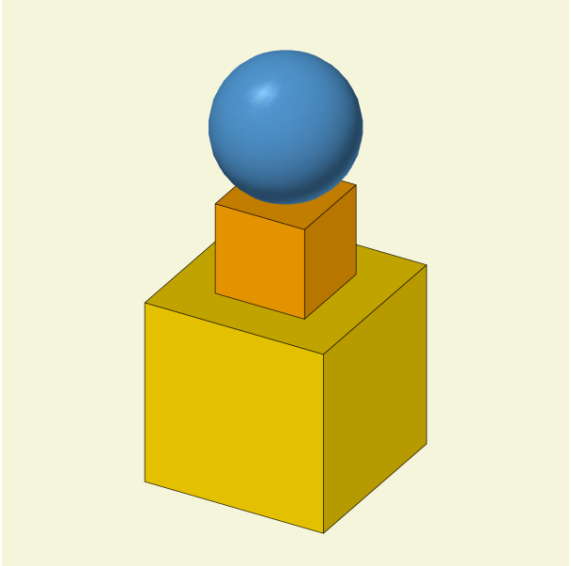


FIGURE 26: *DrawFacet*

11.5 DrawFlatFacet

- **DrawFlatFacet(facettes1, [options1], facettes2, [options2], ...)**
- Description: cette macro trie l'ensemble de toutes les facettes et les affiche en fonction de leurs options, mais **les éventuelles intersection de sont pas gérées et il n'y a pas de lissage de GOURAUD**. Les options possibles sont :
  - backculling := ⟨ 0/1 ⟩. Indique si les facettes non visibles doivent être éliminées ou non (0 par défaut).
  - color := ⟨ couleur ⟩. Choix de la couleur (white par défaut).
  - contrast := ⟨ nombre positif ⟩. Le contraste normal a la valeur 1 (valeur par défaut), un contraste nul signifie que la couleur est unie.

```
\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}
```

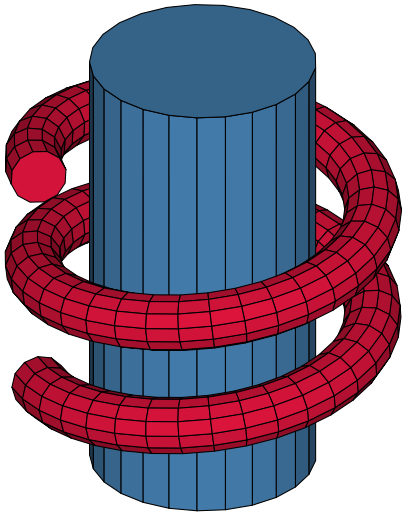


FIGURE 27: *DrawFlatFacet*

11.6 DrawPoly

- **DrawPoly(<polyedre convexe> [, mode, contraste]])**
- Description: elle permet de dessiner un <*polyèdre convexe*> dans le <*mode*> voulu. Ce mode, qui a la valeur 0 par défaut, peut prendre les valeurs suivantes :
  - mode 0 : le dessin se fait arête par arête, y compris les arêtes cachées (qui seront dessinées dans le style HideStyle), pas de remplissage,
  - mode 1 : le dessin se fait par face visible, celles-ci peuvent être remplies en fonction de l'attribut *FillStyle*, toutes les facettes ont alors la même couleur (*FillColor*),
  - mode 2 : le dessin est fait comme dans le mode 1 (faces visibles), puis on rajoute les arêtes cachées,
  - mode=3 : comme le mode 1 mais la couleur de remplissage est nuancée en fonction de l'exposition des facettes et en fonction de la valeur de <*contraste*>,
  - mode=4 : le dessin se fait par face visible mais la couleur de remplissage des facettes est nuancée en fonction de l'exposition des facettes et en fonction de la valeur de <*contraste*>, puis on rajoute les arêtes cachées.
- Le paramètre <*contraste*> est un nombre positif qui vaut 1 par défaut, il permet d'accentuer ou non le contraste des couleurs des différentes facettes, la valeur 0 donnera une couleur unie comme les modes 1 et 2.
- L'avantage de cette macro est la gestion des arêtes, ce qui n'est pas le cas de la macro *DrawFacet* (p. 135).

11.7 DrawSmoothFacet

- **draw("SmoothFacet", facettes1, [options1], facettes2, [options2], ...)**
- Description: cette macro trie l'ensemble de toutes les facettes et les affiche en fonction de leurs options mais **les éventuelles intersections de sont pas gérées**. Les exportations en *pstrick* ou *eps*, et donc *epsc* et *pdf* aussi (mais pas *pdfc*), l'algorithme de GOURAUD est utilisé pour le remplissage des facettes (après triangulation de celles-ci) ce qui donne un effet de lissage, ce lissage n'est pas visible à l'écran. **Avec cette macro les arêtes ne sont pas dessinées**. Les options sont :
  - backculling := ⟨ 0/1 ⟩. Indique si les facettes cachées doivent être éliminées ou non (0 par défaut).
  - color := ⟨ couleur ⟩. Choix de la couleur (white par défaut).
  - contrast := ⟨ nombre positif ⟩. Le contraste normal a la valeur 1 (valeur par défaut), un contraste nul signifie que la couleur est unie.
  - Cette macro utilise un export personnalisé et donc être utilisée sous la forme **draw("SmoothFacet", facettes1, [options1], facettes2, [options2], ...)**, sous cette forme l'export provoquera automatiquement l'exécution de la macro **ExportSmoothFacet()** qui est définie dans le fichier *scene3d.mac*. Alors que sous la forme **DrawSmoothFacet(facettes1, [options1], facettes2, [options2], ...)** l'export sera l'export classique, c'est à dire ce que l'on voit à l'écran (facettes sans lissage).

```
\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}
```

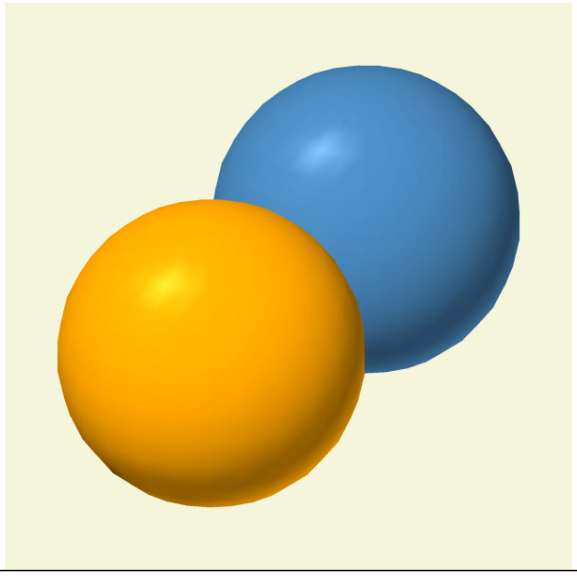


FIGURE 28: *Exemple avec DrawSmoothFacet*

Avertissement : l'exemple ci-dessus illustre la macro *DrawSmoothFacet* qui permet de lisser les facettes avec l'algorithme de GOURAUD. Mais celui-ci n'est vraiment connu que de ghostscript ce qui explique que le rendu en pdf est parfois long (voire très long) et peu intéressant pour de grosses images, dans ces cas là on préférera une image jpeg haute résolution (ou un export *eps* si le document doit rester au format ps).

11.8 Dsurface

- **Dsurface(<f(u,v)> [, uMin+i*uMax, vMin+i*vMax, uNbLg+i*vNbLg, (smooth 0/1)+i*contraste])**
- Description: cette macro dessine une surface paramétrée par <*f(u,v)*> où f est une fonction de deux variables réelles u et v, et à valeurs dans l'espace. Le deuxième paramètre représente l'intervalle de la variable u ($[-5,5]$ par défaut), le troisième paramètre représente l'intervalle de la variable v ($[-5;5]$ par défaut), le quatrième paramètre représente, sous forme complexe, le nombre de lignes pour u et le nombre de lignes pour v (25 lignes par défaut). C'est la macro *DrawFacet* (p. 135) qui fait le rendu avec la couleur correspondant à la variable *FillColor* et avec le <*contraste*> demandé (1 par défaut) et un lissage lorsque <*smooth*> vaut 1 (0 par défaut).

11.9 Dtetraedre

- **Dtetraedre(<sommet>, <vecteur3D1>, <vecteur3D2>, <vecteur3D3> [, mode, contraste])**
- Description: cette macro dessine un tétraèdre à partir d'un <*sommet*> et trois vecteurs, supposés dans le sens direct. Cette macro utilise *DrawPoly* (p. 137) pour dessiner dans le <*mode*> et avec le <*contraste*> voulus.

Chapitre XI

Commande Build3D : représentation d'une scène 3D

Il est désormais possible de mélanger plusieurs objets 3D pour constituer une scène en gérant les intersections. Cette scène est construite à partir de l'algorithme des BSP-trees sous forme d'un arbre par la commande **Build3D**, et la commande **Display3D** permet d'afficher cette scène à l'écran.

Mise en garde : cette technique donne en vectoriel des images qui peuvent rapidement devenir très lourdes pour des scènes un peu complexes (c'est à dire avec un grand nombre de facettes).

1) Les deux commandes de base

1.1 Build3D

Cette commande sert à définir la liste des éléments 3D qui composent la scène. Cette commande ne fait pas de dessin ; comme on peut le voir dans le fichier d'exemple *display3d.teg*, les différentes scènes sont construites dans des macros, et un seul élément graphique suffit, il contient simplement l'instruction *Display3D()* (p. 140). C'est cette commande qui calcule la scène (plus précisément qui construit un arbre d'affichage), et qui affiche la scène. Lorsque par exemple l'angle de vue change, seule la commande Display3D() doit être réévaluée mais pas la commande Build3D().

La syntaxe générale de Build3D est la suivante :

- **Build3D(<objet1>, <objet2>,...)**
- Description: cette fonction détruit la scène existante et en crée une nouvelle avec les objets cités en argument, elle renvoie *Nil*. Chaque objet peut à son tour être une liste d'objets 3D différents, ils sont alors séparés par la constante : *sep3D*. On trouvera plus loin les *macros de construction pour Build3D* (p. 140), mais nous présentons ici les objets « atomiques », ils sont de quatre types, et codés en interne de la façon suivante :
  - **les facettes** : dans ce cas l'objet doit être de la forme :
    [<±1+i*nuance>, <couleur±i*opacité>, <liste facettes>]
    La valeur <*-1*> signifie que le lissage de GOURAUD doit être utilisé dans les exports qui le prennent en compte. Avec la valeur <*1*> il n'y a pas de lissage. La <*nuance*> est facultative et vaut 0 par défaut. L'<*opacité*> est facultative et vaut 1 par défaut, sinon ce doit être un nombre entre 0 et 1, lorsque l'opacité est multipliée par -i, cela signifie par convention, qu'on ne distingue par le devant du derrière de la face, alors qu'avec +i les deux côtés n'ont pas exactement la même couleur. La couleur des facettes est nuancée en fonction de leur exposition, le paramètre <*nuance*> permet de modifier ceci, sa valeur doit supérieure ou égale à -1 :
    - nuance=-1 : pas de nuance, toutes les facettes de l'objet auront la même couleur,
    - nuance=0 : c'est la valeur par défaut,
    - plus on augmente la valeur de nuance, plus le contraste augmente.
  - **les lignes** : dans ce cas l'objet doit être de la forme :
    [<2>, <couleur+i*opacité>, <épaisseur+i*style ligne>, <liste point3D>]
  - **les points** : dans ce cas l'objet doit être de la forme :
    [<3>, <couleur+i*opacité>, <width+i*linestyle, <liste point3D>]
  - **les labels** : dans ce cas l'objet doit être de la forme :
    [<3+i>, <couleur+i*numéro>, <labelsize+i*labelstyle>, <[pos,dir]>]
  - **les labels compilés** :
    [<3-i>, <couleur+i*numéro>, <labelsize+i*labelstyle>, <[pos,dir]>]

- Un certain nombre de macros du fichier *scene3d.mac* (chargé au démarrage) simplifient la définition des éléments d'une scène 3D et peuvent donc être utilisées comme arguments de la commande *Build3D*. Toutes ces macros comportent une liste d'options dans leur dernier argument et une option se déclare ainsi : *<nom> := <valeur>*.
- Exemple(s): on dessine une sphère coupée, un plan, un cylindre, puis les axes avec les traits cachés.

```
\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}
```

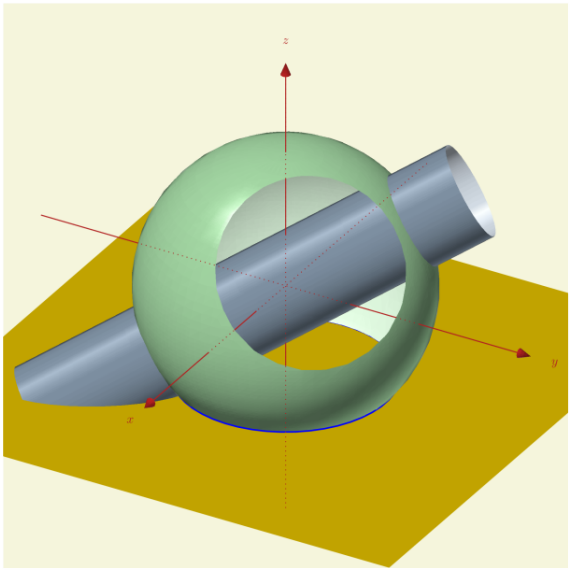


FIGURE 1: *Build3D*

1.2 Display3D

- **Display3D()**
- Description: cette fonction dessine à l'écran la scène créée avec *Build3D* (p. 139). Cette fonction s'utilise sans argument.

2) Les macros pour Build3D()

2.1 Les options globales

- hiddenLines := ⟨ 0/1 ⟩ : cette option est prise en compte par la macro *bdLine* (p. 143). C'est la valeur par défaut de l'option hidden. Lorsque la valeur ce celle-ci est 1, la ligne est dessinée une deuxième fois mais par dessus la scène, dans la même couleur, avec le style *HideStyle* et l'épaisseur *HideWidth* (ou 0.8pt si cette variable est à *Nil*). Cette superposition ne se voit donc pas sur les parties visibles du trait mais seulement sur les parties cachées.
- TeXifyLabels := ⟨ 0/1 ⟩ : cette option est prise en compte par la macro *bdLabel* (p. 143). C'est la valeur par défaut de l'option TeXify, celle-ci indique si le label est une formule mathématique qui doit être compilée par TEX, TeXgraph lance une compilation pdflaTeX en arrière-plan puis appelle l'utilitaire *pstoedit* (`http://www.pstoedit.net/`) qui traduit le fichier pdf en flattened postscript que TeXgraph peut ensuite parser pour récupérer la formule sous forme de chemins. Cela suppose donc qu'une distribution TEX est installée ainsi que le programme pstoedit. Le fichier compilé s'appelle *tex2FlatPs.tex*, et se trouve dans le dossier `$HOME/.TeXgraph` de l'utilisateur sous linux, et dans `c:\tmp` sous windows. On en trouve également une copie dans le dossier d'installation de TeXgraph, par défaut ce fichier utilise la police *fourier* en 12pt, lorsque la variable *dollar* vaut 1, la formule est insérée entre les deux délimiteurs : `\[..\]`, sinon elle est laissée telle quelle, puis elle est composée avec la taille `\large`. Par défaut cette option vaut 0.

2.2 bdArc

- **bdArc(<B>, <A>, <C>, <R>, <sens>, [options])**
- Description: définit un arc de cercle dans l'espace de rayon *<R>*, allant de (AB) vers (AC). Le plan (BAC) est orienté par la base $(\vec{AB},\vec{AC})$ et le *<sens>* doit valoir 1 s'il est direct ou -1 sinon. Options de bdArc :
  - labelarc(<"texte">). C'est une macro qui permet de placer un label sur l'arc.
  - normal := ⟨ vecteur3D non nul ⟩. Vecteur qui sera considéré comme le vecteur normal au plan si l'angle est plat (*Nil* par défaut).

– radscale := ⟨ nombre ⟩. Nombre qui, multiplié par le rayon de l'arc, donnera la distance du label au centre de l'arc (1.25 par défaut).

- C'est la macro bdCurve qui est appelée pour dessiner l'arc, on peut donc utiliser les options de *bdCurve* (p. 142).

2.3 bdAngleD

- **bdAngleD(<B>, <A>, <C>, <longueur>, [options])**
- Description: crée « l'angle droit » de l'espace défini par les deux droites (*AB*) et (*AC*), où *A*, *B* et *C* sont des points 3D.
- Cette macro appelle bdLine, on peut donc utiliser les options de *bdLine* (p. 143).
- Exemple(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}
```

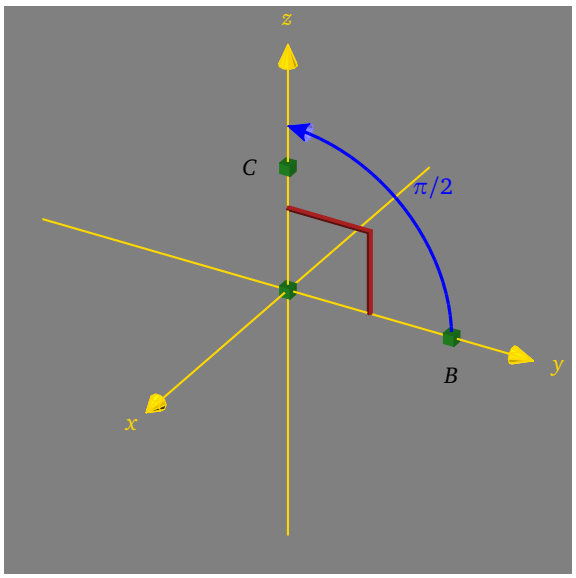


FIGURE 2: *bdAngleD*

2.4 bdAxes

- **bdAxes(<point3D>, [options])**
- Description: définit les axes, <*point3D*> est le point de concours des trois axes. Options de bdAxes :
  – labels := ⟨ 0/1 ⟩. Indique la présence ou non des lettres *x*, *y* et *z* au bout des trois axes (1 par défaut).
  – newxlegend(<"texte">), newylegend(<"texte">), newzlegend(<"texte">) : macros qui permettent de définir la légende sur les axes, par défaut il s'agit de : `$x$`, `$y$` et `$z$`.
- Cette macro appelle bdLine, on peut donc utiliser les options de *bdLine* (p. 143).

2.5 bdCercle

- **bdCercle(<point3D>, <rayon R>, <vecteur3D normal>, [options])**
- Description: définit un cercle dans l'espace de centre <*point3D*> et de <*rayon R*>, le plan du cercle est orthogonal au <*vecteur3D normal*>.
- Cette macro appelle bdCurve, on peut donc utiliser les options de *bdCurve* (p. 142).
- Exemple(s): les cercles de *Villarceau* (p. 146).

2.6 bdCone

- **bdCone(<point3D>, <vecteur3D>, <rayon>, [options])**
- Description: définit le cône construit à partir d'un <*point3D*> qui est le sommet, d'un <*vecteur3D*> de l'axe qui indique la direction et la hauteur du cône, et du <*rayon*> de la face circulaire. Les options de bdCone sont celles de *bdFacet* (p. 142), plus :
  – hollow := ⟨ 0/1 ⟩. Indique si le cône est creux ou non (1 par défaut).
  – nbfacet := ⟨ nombre de facettes ⟩. Définit le nombre de facettes (35 par défaut).
  – border := ⟨ 0/1 ⟩. Indique si le contour doit être dessiné ou non (0 par défaut).
  – bordercolor := ⟨ couleur ⟩. Indique la couleur du contour (identique à *color* par défaut).

2.7 bdCurve

- **bdCurve(<f(t)>, [options])**
- Description: définit une courbe dans l'espace, celle-ci est paramétrée par $f(t) = [x(t) + i * y(t), z(t)]$ ou $f(t) = M(x(t), y(t), z(t))$, où $x(t)$, $y(t)$ et $z(t)$ sont des fonctions d'une variable t. Options de bdCurve :
  - t := ⟨ [tmin, tmax] ⟩. Intervalle pour le paramètre t, [-5,5] par défaut.
  - nbdot := ⟨ entier positif ⟩. Définit le nombre de points, celui-ci est de 25 par défaut.
- Cette macro appelle bdLine, on peut donc utiliser les options de *bdLine* (p. 143).

2.8 bdCylinder

- **bdCylinder(<point3D>, <vecteur3D>, <rayon>, [options])**
- Description: définit le cylindre construit à partir d'un <*point3D*> qui est le centre d'une des deux faces circulaires, d'un <*vecteur3D*> de l'axe qui indique la direction et la hauteur du cylindre, et du <*rayon*>. Les options de bdCylinder sont celles de *bdFacet* (p. 142), plus :
  - hollow := ⟨ 0/1 ⟩. Indique si le cylindre est creux ou non (1 par défaut).
  - nbfacet := ⟨ nombre de facettes ⟩. Définit le nombre de facettes (35 par défaut).
  - border := ⟨ 0/1 ⟩. Indique si le contour doit être dessiné ou non (0 par défaut).
  - bordercolor := ⟨ couleur ⟩. Indique la couleur du contour (identique à *color* par défaut).

2.9 bdDot

- **bdDot(<liste point3D>, [options])**
- Description: définit une liste de points de l'espace. Options de bdDot :
  - color := ⟨ couleur ⟩. Définit la couleur des points (black par défaut).
  - dir := ⟨ vecteur3D1 ou [vecteur3D1,vecteur3D2] ⟩. Lorsque dotstyle=line (un trait), l'option dir doit contenir un vecteur directeur du trait à tracer (dans l'espace). Lorsque dotstyle=cross (croix), l'option dir doit contenir une liste de deux vecteurs directeurs pour les traits à tracer (dans l'espace). Par défaut dir vaut *Nil*.
  - dotscale := ⟨ nombre positif ⟩. Définit un facteur d'échelle (1 par défaut).
  - dotstyle := ⟨ disc/cube/line/cross ⟩. Définit le style de points (disc par défaut).
- Lorsque dostsyle=cube la macro bdFacet est appelée, on peut dans ce cas utiliser les options de *bdFacet* (p. 142), lorsque dotstyle=line ou cross la macro bdLine est appelée, on peut alors utiliser les options de *bdLine* (p. 143).

2.10 bdDroite

- **bdDroite(<[point3D, vecteur3D]>, [options])**
- Description: définit une droite, celle-ci est représentée par la liste <*[point 3D, vecteur3D directeur]*>. Options de bdDroite :
  - scale := ⟨ nombre strictement positif ⟩. La droite est intersectée par la fenêtre 3D courante ce qui donne un segment, celui-ci peut être agrandi ou diminué.
- Cette macro appelle bdLine, on peut donc utiliser les options de *bdLine* (p. 143).

2.11 bdFacet

- **bdFacet(<liste facettes>, [options])**
- Description: définit une liste de facettes. Options de bdFacet :
  - backculling := ⟨ 0/1 ⟩. Indique si les facettes non visibles doivent être éliminées ou non (0 par défaut). Une facette est non visible lorsque son vecteur normal n'est pas dans la direction de l'observateur.
  - clip := ⟨ 0/1 ⟩. Indique si les facettes doivent être clippées par la fenêtre définie par l'option clipwin lorsque clip vaut 1, ou bien par le plan défini par l'option clipwin lorsque clip vaut -1 (clip=0 par défaut).
  - clipwin := ⟨ [M(xinf,yinf,zinf), M(xsup,yup,zsup)] ⟩. Définit la fenêtre 3D pour un éventuel clipping lorsque clip=1, la fenêtre est alors donnée par sa grande diagonale : [M(xinf,yinf,zinf), M(xsup,yup,zsup)] (c'est la fenêtre courante par défaut). Mais lorsque clip=-1 l'option clipwin est interprétée comme un plan : [point3D, vecteur3D normal].
  - color := ⟨ couleur ⟩. Choix de la couleur (white par défaut).
  - contrast := ⟨ nombre positif ⟩. Le contraste normal a la valeur 1 (valeur par défaut), un contraste nul signifie que la couleur est unie.

- smooth := ⟨ 0/1 ⟩. Indique si l'algorithme de GOURAUD (lissage des facettes) doit être utilisé ou non lors de l'exportation pstricks ou eps (0 par défaut). Attention, les visualisateurs pdf sont lents pour afficher ce type d'images !
- opacity := ⟨ nombre entre 0 et 1 ⟩. Valeur de l'opacité (1 par défaut), permet d'introduire la transparence lorsque l'opacité est strictement inférieure à 1.
- matrix := ⟨ matrice 3d ⟩. Permet de définir une matrice de transformation qui sera appliquée aux facettes (l'identité par défaut). La transformation s'effectue avant l'éventuel clipping.
- twoside := ⟨ 0/1 ⟩. Indique si on doit distinguer ou non le devant-derrière des facettes. Dans l'affirmative, les deux côtés n'ont pas la même couleur (1 par défaut).
- above := ⟨ nombre positif ou nul ⟩. Permet de placer les facettes par dessus la scène, elles sont translatées avec le vecteur `above*500*\n` (0 par défaut).
- border := ⟨ 0/1 ⟩. Indique si on doit dessiner ou non les arêtes des facettes (0 par défaut).
- bordercolor := ⟨ couleur ⟩. Couleur des arêtes lorsque border=1 (black par défaut).
- hidden := ⟨ 0/1 ⟩. Indique si les arêtes cachées doivent être dessinées lorsque border=1, si c'est le cas alors les variables *HideStyle* et *HideWidth* sont utilisées. Par défaut, cette option a la valeur de l'option générale *hiddenLines*.

2.12 bdLabel

- **bdLabel(<point3D>, <"texte">, [options])**
- Description: définit un label dans l'espace, le <*point3D*> est le point d'ancrage. Le label est dessiné sur le plan de projection et non pas réellement dessiné dans l'espace, mais son point d'ancrage est géré dans la scène pour déterminer l'ordre d'affichage. Options de bdLabel :
  - TeXify := ⟨ 0/1 ⟩ : indique si le label doit être compilé par TEX, voir l'option générale *TeXifyLabels* (p. 140).
  - scale := ⟨ nombre>0 ⟩. Lorsque l'option *TeXify* vaut 1, la taille du label peut être modifiée avec cette option.
  - color := ⟨ couleur ⟩. Définit la couleur du label (black par défaut).
  - dotcolor := ⟨ couleur ⟩. Définit la couleur du point d'ancrage si celui-ci doit être affiché (égale à color par défaut).
  - labelpos := ⟨ [distance cm, affixe direction] ⟩. Indique la position du label par rapport au point d'ancrage **sur le plan de projection** (*Nil* par défaut, dans ce cas la distance est considérée comme nulle).
  - labelsize := ⟨ small/... ⟩. Définit la taille du label comme LabelSize (égal à *LabelSize* par défaut) lorsque l'option *TeXify* vaut 0.
  - labelstyle := ⟨ type de label ⟩. Définit le style de label comme LabelStyle (égal à *LabelStyle* par défaut).
  - showdot := ⟨ 0/1 ⟩. Indique si le point d'ancrage doit être affiché (0 par défaut).
- Lorsque *showdot* vaut 1, on peut utiliser les options de *bdDot* (p. 142) car celle-ci sera appelée.
- Exemple(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}
```

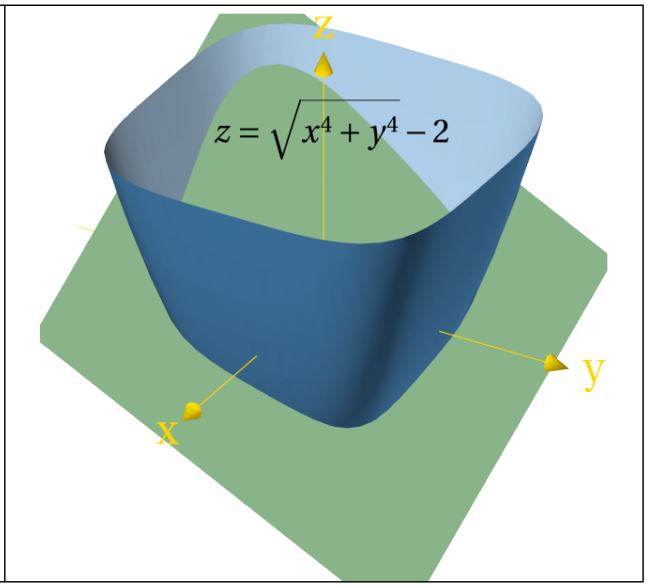


FIGURE 3: *Utilisation de l'option TeXify*

2.13 bdLine

- **bdLine(<liste point3D>, [options])**

- Description: définit une ligne polygonale dans l'espace. Options de bdLine :
  - arrows := ⟨ 0/1/2 ⟩. Indique la présence ou non de flèche (aucune, une ou deux, aucune par défaut). Cette option suppose que la ligne ne contient pas la constante *jump*.
  - arrowscale := ⟨ nombre positif ⟩. Facteur d'échelle pour les flèches (1 par défaut).
  - clip := ⟨ -1/0/1 ⟩. Indique si la ligne doit être clippée par la fenêtre définie par l'option *clipwin* lorsque *clip* vaut 1, ou bien par le plan défini par l'option *clipwin* lorsque *clip* vaut -1 (clip=0 par défaut).
  - clipwin := ⟨ [M(xinf,yinf,zinf), M(xsup,yup,zsup)] ⟩. Définit la fenêtre 3D pour un éventuel clipping lorsque *clip* vaut 1, la fenêtre est alors donnée par sa grande diagonale : [M(xinf,yinf,zinf), M(xsup,yup,zsup)] (c'est la fenêtre courante par défaut). Mais lorsque *clip* vaut -1 l'option *clipwin* est interprétée comme un plan : [point3D, vecteur normal].
  - close := ⟨ 0/1 ⟩. Indique s'il faut refermer la ligne ou non, (0 par défaut).
  - color := ⟨ couleur ⟩. Choix de la couleur (black par défaut).
  - hollow := ⟨ 0/1 ⟩. Lorsque l'option *tube* vaut 1, la ligne est remplacée par un tube à facettes. Celui-ci peut être creux (hollow :=1) ou non (hollow :=0) (0 par défaut).
  - linestyle := ⟨ style de ligne ⟩. Définit le style de tracé de ligne (solid par défaut).
  - nbfacet := ⟨ nombre de facettes ⟩. Définit le nombre de facettes lorsque *tube* vaut 1 (4 facettes par défaut).
  - opacity := ⟨ nombre entre 0 et 1 ⟩. Valeur de l'opacité (1 par défaut), permet d'introduire la transparence lorsque l'opacité est strictement inférieure à 1.
  - radius := ⟨ rayon du tube ⟩. Rayon du tube lorsque *tube* vaut 1 (0.01 par défaut).
  - radiusscale := ⟨ nombre>0 ⟩. Facteur d'échelle pour le rayon du tube lorsque *tube* vaut 1 (1 par défaut).
  - tube := ⟨ 0/1 ⟩. Indique s'il faut construire un tube (à facettes) à partir de la ligne (0 par défaut).
  - width := ⟨ épaisseur du trait ⟩ (8 par défaut).
  - matrix := ⟨ matrice 3d ⟩. Permet de définir une matrice de transformation qui sera appliquée aux points de la ligne (l'identité par défaut). La transformation s'effectue avant l'éventuel clipping.
  - above := ⟨ nombre positif ou nul ⟩. Permet de placer la ligne par dessus la scène, elle est translatée avec le vecteur `above*500*\n` (0 par défaut).
  - hidden := ⟨ 0/1 ⟩. Indique si les traits cachés doivent être dessinés lorsque border=1, si c'est le cas alors les variables *HideStyle* et *HideWidth* sont utilisées. Par défaut, cette option a la valeur de l'option générale *hiddenLines*.
- Lorsque l'option *tube* vaut 1, la macro bdFacet est appelée, on peut donc utiliser dans ce cas les options de *bdFacet* (p. 142).

2.14 bdPlan

- **bdPlan(<plan>, [options])**
- Description: définit un plan, ce <*plan*> est représenté par une liste du type : [point 3D, vecteur3D normal]. Options de bdPlan :
  - scale := ⟨ nombre strictement positif ⟩. Le plan est intersecté par la fenêtre 3D courante ce qui donne une facette, celle-ci peut être agrandie ou diminuée.
- Cette macro appelle bdFacet, on peut donc utiliser les options de *bdFacet* (p. 142). Par défaut l'option *twoside* vaut 0 (on ne distingue pas le devant-derrière de la facette).

```
\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}
```

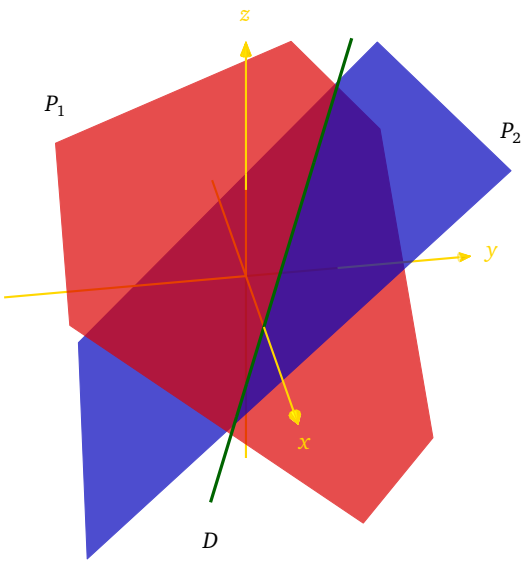


FIGURE 4: *Intersection de 2 plans*

2.15 bdPlanEqn

- **bdPlanEqn(<[a,b,c,d]>, [options])**
- Description: définit le plan d'équation $ax + by + cz = d$, celui-ci est représenté par la liste : *<[a,b,c,d]>*. Options de bdPlanEqn :
  - scale := ⟨ nombre strictement positif ⟩. Le plan est intersecté par la fenêtre 3D courante ce qui donne une facette, celle-ci peut être agrandie ou diminuée.
- Cette macro appelle bdFacet, on peut donc utiliser les options de *bdFacet* (p. 142). Par défaut l'option *twoside* vaut 0 (on ne distingue pas le devant-derrière de la facette).

2.16 bdPrism

- **bdPrism(<liste de point3D>, <vecteur3D>, [options])**
- Description: définit le prisme construit à partir d'une *<liste de point3D>* qui forme la base (supposée plane), et un *<vecteur3D>* de translation pour calculer l'autre base. Les options de bdPrism sont celles de *bdFacet* (p. 142), plus :
  - hollow := ⟨ 0/1 ⟩. Indique si le prisme est creux ou non (1 par défaut).

  Lorsque l'option border vaut 1, la macro *bdLine* (p. 143) est appellée, on peut donc utiliser les options de celles-ci.

2.17 bdPyramid

- **bdPyramid(<liste de point3D>, <point3D>, [options])**
- Description: définit la pyramide construite à partir d'une *<liste de point3D>* qui forme la base (supposée plane), et un *<point3D>* qui est le sommet. Les options de bdPyramid sont celles de *bdFacet* (p. 142), plus :
  - hollow := ⟨ 0/1 ⟩. Indique si la pyramide est creuse ou non (1 par défaut).

  Lorsque l'option border vaut 1, la macro *bdLine* (p. 143) est appellée, on peut donc utiliser les options de celles-ci.

2.18 bdSphere

- **bdSphere(<point3D>, <rayon R>, [options])**
- Description: définit une sphère de centre *<point3D>*, et de *<rayon R>*. Les options sont celles de *bdFacet* (p. 142) plus :

- grid := ⟨ [nb méridiens, nb parallèles] ⟩. Nombre de méridiens et de parallèles pour définir les facettes, [40,25] par défaut.
- border := ⟨ 0/1 ⟩. Indique si le contour doit être dessiné ou non (0 par défaut).
- bordercolor := ⟨ couleur ⟩. Indique la couleur du contour (identique à *color* par défaut).

2.19 bdSurf

- **bdSurf(<f(u,v)>, [options])**
- Description: définit une surface paramétrée, par $f(u,v) = [x(u,v) + i * y(u,v), z(u,v)] = M(x(u,v), y(u,v), z(u,v))$, où x, y et z sont des fonctions des deux variables u et v. Options de bdSurf :
  - u := ⟨ [umin, umax] ⟩. Intervalle pour la variable u, [-5,5] par défaut.
  - v := ⟨ [vmin, vmax] ⟩. Intervalle pour la variable v, [-5,5] par défaut.
  - grid := ⟨ [unbdot, vnbdot] ⟩. Définit la grille, c'est à dire le nombre de points pour u et pour v, celle-ci est de [25,25] par défaut.
- Cette macro appelle bdFacet, on peut donc utiliser les options de *bdFacet* (p. 142).

2.20 bdTorus

- **bdTorus(<point3D>, <rayon R>, <rayon r>, <vecteur3D normal>, [options])**
- Description: définit un tore de centre *<point3D>*, de grand *<rayon R>*, de petit *<rayon r>*, le *<vecteur3D normal>* permet de définir le « plan du tore ». Les options de bdTorus sont celles de *bdFacet* (p. 142), plus :
  - grid := ⟨ [nb méridiens, nb parallèles] ⟩. Nombre de méridiens et de parallèles pour définir les facettes, [40,25] par défaut.

```
\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}
```

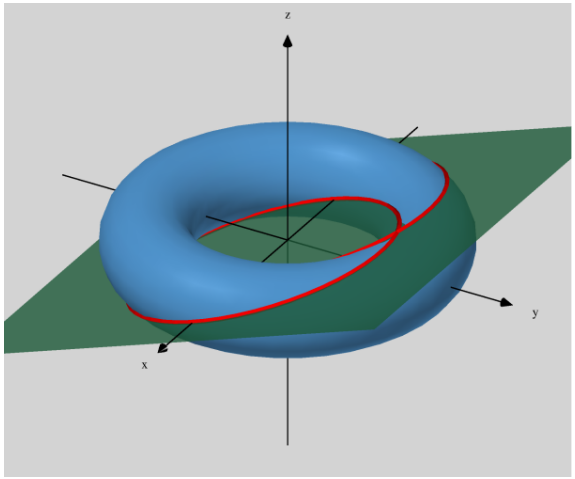


FIGURE 5: *Cercles de Villarceau*

3) Exportations en obj, geom et jvx

3.1 Scène construite avec Build3D

Trois nouvelles exportations sont apparues en bas du menu *Fichier*, celles-ci ne s'appliquent qu'à la scène construite avec la commande *Build3D()*. Ces exports sont :

1. format **obj** : les fichiers *obj* peuvent être lus par la plupart des grands logiciels de 3D, comme *Blender* (`http://www.blender.org/`) par exemple.
2. format **geom** : les fichiers geom sont destinés uniquement au logiciel *geomview* (`http://www.geomview.org/`) qui permet en particulier une manipulation à la souris de la figure dans l'espace.

3. format **jvx** : les fichiers *jvx* sont destinés uniquement à l'applet *javaview* (`http://www.javaview.de/`) qui permet une manipulation à la souris de la figure dans l'espace, plus de nombreuses autres options permettant de contrôler la scène (comme cacher certains éléments, ou exporter la scène...) grâce à un panneau de contrôle. L'affichage peut se faire dans une page web, ou bien en local dans une fenêtre java.

Ces trois exportations peuvent aussi être activées par les commandes :
Export(obj, <nom de fichier>) ou Export(geom, <nom de fichier>) ou Export(jvx, <nom de fichier>).
où <nom de fichier> désigne le nom complet du fichier avec extension.

3.2 Scène construite sans Build3D

Il est également possible d'exporter une scène aux formats *obj*, *geom* et *jvx* sans passer par la commande Build3D :

- **SceneToObj(<nom de fichier>, <élément1>, <élément2>, ...)**
- **SceneToGeom(<nom de fichier>, <élément1>, <élément2>, ...)**
- **SceneToJvx(<nom de fichier>, <élément1>, <élément2>, ...)**
- Description: l'argument <*nom de fichier*> désigne le nom complet du fichier sans l'extension, celle-ci étant automatiquement ajoutée. Les arguments suivants sont les éléments qui composent la scène, **ce sont les mêmes arguments que l'on passerait à la commande Build3D**, on peut en particulier utiliser les macros prévues initialement pour *Build3D* (p. 140) (bdAxes, bdArc, ...).

3.3 Export d'un élément isolé

Il y a deux autres macros d'export qui sont :

- **WriteObj(<nom de fichier>, <liste des sommets>, <liste des facettes> [, liste des lignes])**,
- **WriteOff(<nom de fichier>, <liste des sommets>, <liste des facettes> [, liste des lignes])**,
- Description: l'argument <*nom de fichier*> désigne le nom complet du fichier sans l'extension, celle-ci étant automatiquement ajoutée. L'argument suivant est la liste des points 3D qui sont les sommets des facettes et/ou des lignes qui suivent. Le troisième argument est la liste des facettes où **chaque sommet est remplacé par son numéro de position dans la liste des sommets** (de même pour le dernier argument). C'est le format naturel pour les fichiers *obj*. La commande *ConvertToObj* (p. 105) peut être utilisée pour faire cette conversion.

Le format *off* est un format du logiciel *geomview*.

Chapitre XII

Du code TeXgraph dans un fichier LaTeX

1) Installation

Sous windows, il vous faudra copier le fichier *texgraph.sty* dans votre arborescence TEX et mettre la base à jour. Sous linux, l'installation du paquet *texgraph.sty* est faite automatiquement lors de l'exécution du script *install.sh*.

IMPORTANT : la compilation d'un document LATEX utilisant ce paquet, doit se faire avec l'option –shell-escape (ou –enable-write18 suivant la distribution).

2) Utilisation

Une fois déclaré le paquet avec : `\usepackage{texgraph}`, vous pouvez utiliser l'environnement :

```
\begin{texgraph}[<options>]
 <code TeXgraph>
\end{texgraph}
```

Lors de la compilation le code est copié dans un fichier *<nom>.teg* (fichier source de TeXgraph) en tant qu'élément graphique Utilisateur (par défaut), puis le programme *TeXgraphCmd* est appelé, il charge le fichier *<nom>.teg*, exporte le résultat dans le format demandé, enfin, le compilateur LATEX reprend la main et le fichier résultant est chargé avec `\input` ou bien `\includegraphics` suivant l'export demandé.

Pour être tout à fait exact, c'est un script qui est appelé : *CmdTeXgraph*, le contenu de celui-ci sous linux est :

```
#!/bin/sh
export TeXgraphDir= <accès au dossier TeXgraph>
export PATH=<accès au dossier TeXgraph>:$PATH
$TeXgraphDir/TeXgraphCmd $1 $2 $3 > $3.log
```

Le champ *<accès au dossier TeXgraph>* est complété lors de l'installation. Le premier paramètre est le type d'export demandé (pst, pgf, tkz,...), le deuxième indique si le source doit être exporté (option *src=true*), et le troisième est le nom du fichier source sans extension.

Les options possibles sont :

- name = < nom > : permet de donner un nom à l'image (sans extension), par défaut ce nom est le nom du fichier courant suivi du numéro d'apparition de l'environnement (fichier1, fichier2, ...). Ce paramètre doit être indiqué en premier lorsqu'il n'est pas omis.
- export = < none/pst/pgf/tkz/eps/psf/pdf/epsc/pdfc/teg/texsrc > : ce paramètre peut prendre les valeurs suivantes : *none* (aucun fichier n'est exporté), *pst* (pstricks, option par défaut), *pgf*, *tkz* (pgf en fait mais dans un environnement tikzpicture ce qui permet d'ajouter des instructions tikz), *eps*, *psf* (eps+psfrag), *pdf*, *epsc* (eps compilé), *pdfc* (pdf compilé), *teg* (fichier source texgraph) ou *texsrc* (fichier source texgraph colorisé pour TEX). Il détermine automatiquement le type d'export ainsi que le mode d'inclusion (input ou includegraphics ou rien).
- call = < true/false > : ce booléen vaut *true* par défaut, il indique si on appelle réellement TeXgraph, dans la négative le code TeXgraph est ignoré, ce qui permet d'éviter les appels inutiles en cas de compilations multiples, le fichier image est cependant inclus, en fonction du paramètre *auto*. Lorsque *call* a la valeur *true*, un *<fichier>.teg* est créé, il est compilé par *TeXgraphCmd* qui exporte ensuite un fichier image et un fichier log.
- auto = < true/false > : ce booléen vaut *true* par défaut, il indique si le fichier image doit être inclus automatiquement à l'aide des macros input ou includegraphics. Dans la négative le fichier image n'est pas chargé. Lorsqu'elle n'est pas omise, cette option doit être indiquée après l'option export.

- commandchars = < true/false > : ce booléen vaut *false* par défaut, lorsqu'il a la valeur *true*, l'environnement peut contenir des appels à des commandes TEX à condition de remplacer \ par # devant le nom des commandes, ex : `#commande{...}`. Si cette commande contient des macros qui ne doivent pas être développées, elles devront être précédées de `\noexpand`.
- src = < true/false > : ce booléen vaut *false* par défaut, lorsqu'il a la valeur *true*, TEXgraph exportera en plus du graphique, le fichier source colorisé en TEX (fichier avec l'extension *src*), et c'est ce fichier source qui est inclus à la place de l'environnement, comme dans tous les exemples que l'on peut voir dans ce document. Les différentes couleurs sont prédéfinies dans le fichier *texgraph.sty* et peuvent être redéfinies par l'utilisateur dans son document. Voici les définitions :

```
\newcommand*{\TegSrcFontSize}{small}%taille des caractères
\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 > : ce booléen vaut *false* par défaut, il indique si le contenu de l'environnement est un fichier source TeXgraph entier (*file=true*), ou bien seulement un élément graphique Utilisateur (*file=false*).
- preload = < {"<fichier1>";"<fichier2>";...} > : permet de charger un ou plusieurs paquets avant de créer le graphique, ex : *preload={"papiers.mod";"draw2d.mod"}*.
- cmdi = < commande > : permet d'importer le graphique à l'intérieur la commande, ex : `cmdi={\raisebox{-2cm}}`
- cmdii = < commande > : applique une deuxième commande par dessus la première (*cmdi*).

Le paquet possède trois options globales qui sont :

- nocall : cette option permet de redéfinir la valeur par défaut de l'option *call* à la valeur *false*, par conséquent les environnements texgraph n'appelleront le programme *TeXgraphCmd* que si l'option *call* (ou *call=true*) est mentionnée.
- src : cette option permet de redéfinir la valeur par défaut de l'option *src* à la valeur *true* pour tous les environnements texgraph.
- export = < pst/pgf/tkz/eps/psf/pdf/epsc/pdfc > : cette option permet de redéfinir l'export par défaut.

Exemples : `\usepackage[nocall]{texgraph}` ou encore `\usepackage[export=pgf]{texgraph}`.

Mises en garde

- les commandes et macros relatives à l'interface graphique (la souris, le menu, les boutons les items, le timer, ...) sont ignorées.
- débuter une ligne par un commentaire entre accolades provoque une erreur lorsque l'option *commandchars* est activée, par contre on peut commenter en début de ligne de la manière suivante : `//blablabla` (toute la ligne est alors en commentaire).

3) La commande Special

La commande *Special(<chaine>)* écrit la chaîne telle quelle dans le fichier d'exportation. Si la chaîne contient les balises \[et \], alors le texte compris entre celles-ci est interprété et évalué par TeXgraph avant l'écriture. Par exemple, la présence de :

```
Special("\gdef\a{\[Int(exp(-u*u),u,0,1)\]}")
```

définira dans un export interprété par TEX (comme tex, ou pst ou pgf ou tkz), une macro globale (utilisable en dehors de l'environnement texgraph), appelée *a*, et contenant une valeur approchée de :

$$\int_0^1 e^{-u^2}\,du$$

4) Exemples

Avec l'option *file=false* (option par défaut), le code TeXgraph est inclus dans un élément graphique utilisateur avant d'être envoyé au programme *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}
```

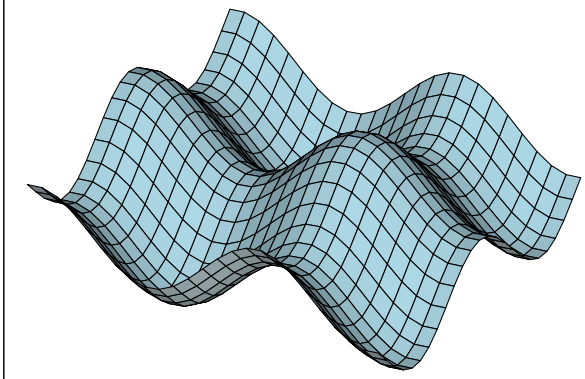


FIGURE 1: Un exemple avec file=false

Dans ce premier exemple, le fichier véritablement envoyé au programme est :

```
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)
   ];
```

Avec l'option *file=true*, le code TeXgraph est considéré comme un fichier source pour le programme *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}
```

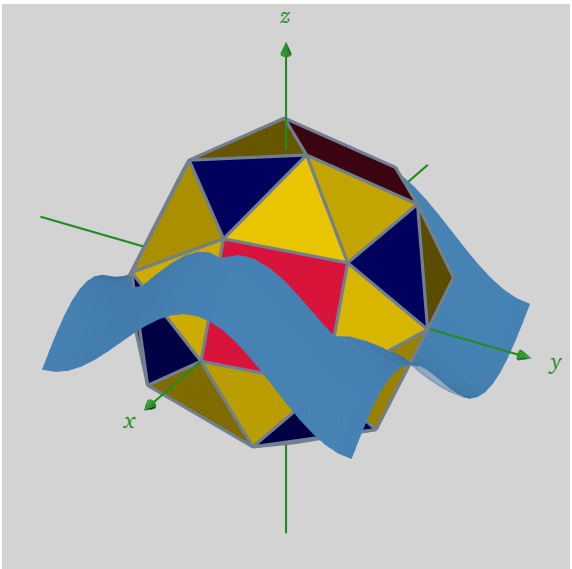


FIGURE 2: Un exemple avec file=true

5) Syntaxe d'un fichier source

Dans ce deuxième exemple, le véritable fichier envoyé au programme est :

```
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()];
```

- La première ligne (`TeXgraph#`) est automatiquement ajoutée, elle annonce un fichier source (pour la version 1.95 et les suivantes, les sources des anciennes versions sont néanmoins compatibles).
- La rubrique **Cmd** annonce des commandes, chaque commande se termine par un point-virgule, les commandes sont exécutées au fur et à mesure de la lecture du fichier.
- La rubrique **Include** annonce des fichiers à charger, chaque nom de fichier est une chaîne suivie par un point-virgule, les fichiers sont chargés au fur et à mesure de la lecture.
- La rubrique **Var** annonce la déclaration des variables globales, la syntaxe de cette déclaration est :
  <nom> = <expression> ;
  L'<*expression*> est évaluée numériquement avant d'être affectée à la variable globale <*nom*>. Les déclarations sont exécutées au fur et à mesure de la lecture du fichier.
- La rubrique **Mac** annonce la déclaration des macros, la syntaxe de cette déclaration est :
  <nom> = <expression> ;
  L'<*expression*> est analysée et s'il n'y a pas d'erreur une macro appelée <*nom*> est créée avec cette <*expression*>. Les déclarations sont exécutées au fur et à mesure de la lecture du fichier.
- La rubrique **Graph** annonce la déclaration des éléments graphiques Utilisateurs, la syntaxe de cette déclaration est :
  <nom> = <expression> ;
  L'<*expression*> est analysée et s'il n'y a pas d'erreur un élément graphique appelé <*nom*> est créé avec cette <*expression*>. Les éléments graphiques sont créés au fur et à mesure de la lecture du fichier.

Quelques règles :

1. La première ligne est obligatoire.
2. Il peut y avoir plusieurs rubriques *Cmd*, *Include*, *Var*, *Mac* et *Graph*.
3. Il n'y a pas d'ordre impératif à respecter sur les rubriques, il faut simplement se souvenir qu'une variable globale (ou une macro), n'existe qu'après avoir été déclarée.

Index