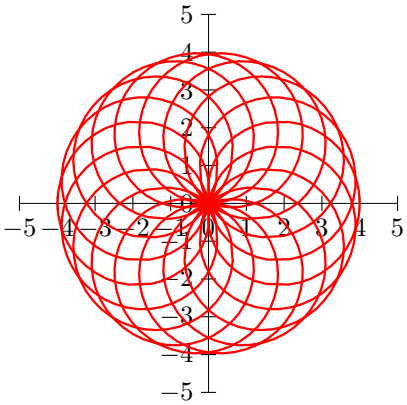
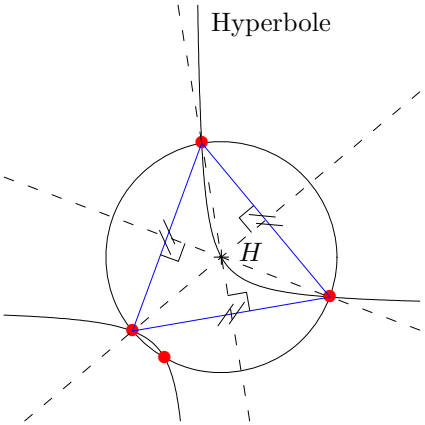
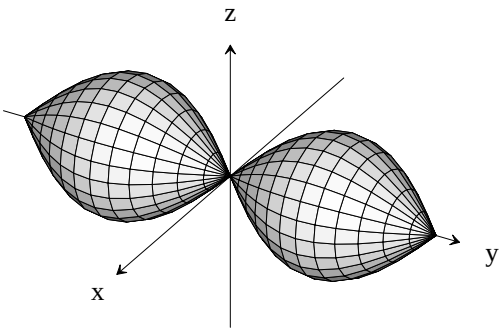
TEXGRAPH 1.9







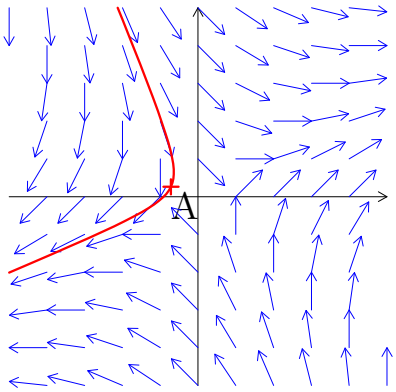


Table des matières

Chapitre 1

Introduction

I) Qu'est ce que TeXgraph ?

1) Présentation

TEXgraph est un programme permettant la réalisation de graphiques mathématiques que l'on peut ensuite exporter dans un fichier texte au format LATEX, au format **PsTricks**, au format **Pgf**, au format **Eps**, ou au format **Pdf**, ce qui permet de les incorporer dans un document TEX[1]. Le logiciel donne un aperçu du graphique à l'écran au fur et à mesure de sa conception, c'est plutôt un « brouillon », notamment en ce qui concerne les labels même s'ils sont correctement placés, car les formules TEX ne sont pas interprétées par TEXgraph mais simplement affichées.

Par contre, après l'inclusion dans un document TEX, avec les formats LATEX, **PsTricks** et **Pgf**, les formules TEX sont interprétées (mais pas en **Eps** ni en **Pdf**, on peut néanmoins utiliser le package **PsFrag** avec les graphiques au format **Eps** (sortie **Psf**).)

Les graphiques peuvent également être enregistrés ou lus par TEXgraph sous forme de fichiers sources (fichiers ***.teg**), ce sont des fichiers texte contenant une série de codes décrivant le graphique. Ces mêmes fichiers peuvent être enregistrés sous forme de **modèle** en mettant l'extension ***.mod**. Un fichier modèle peut être **importé**[2], son contenu vient alors s'ajouter au graphique en cours.

Exemples de graphiques obtenus (sortie LATEX) :

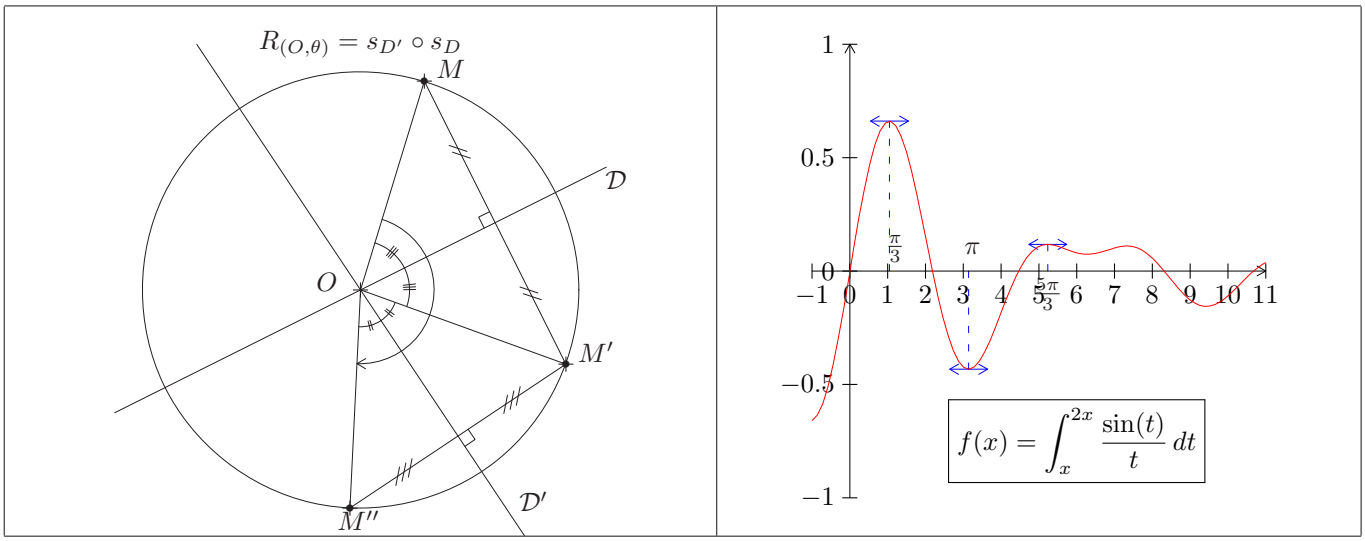


FIG. 1.1 – Exemples

1. Un graphique trop « riche » au format LATEX ou **PsTricks** ou **Pgf**, peut faire déborder la mémoire de TEX !
2. Option `Fichier/Importer un modèle`.

2) Lancement de TeXgraph

Le programme exécutable s'appelle `TeXgraph.exe` et ne nécessite l'installation d'aucune librairie supplémentaire. Au lancement du programme, celui-ci charge le fichier de macros `TeXgraph.mac` qui doit être dans le répertoire courant, si l'absence de ce fichier n'empêche pas le lancement et le fonctionnement du programme, il manquera néanmoins un bon nombre de macros.

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. Les macros ainsi chargées au démarrage sont considérées comme prédéfinies et ne seront supprimées de la mémoire qu'en quittant le programme.

On peut également charger un fichier de macros par le biais du menu avec l'option `Fichier/Charger des macros`. Les macros ainsi chargées sont également considérées comme prédéfinies et ne feront pas partie des graphiques[3], par contre elles seront supprimées de la mémoire au prochain changement de fichier.

II) Composition d'un graphique

1) Les différents éléments

Un graphique est défini par la donnée de :

- **Paramètres** : qui sont les coordonnées (Xmin, Ymin) et (Xmax, Ymax) de la **Fenêtre** contenant le graphique, l'échelle sur les deux axes (en cm), les marges autour du dessin (pour les débordements éventuels) et trois booléens (Bordure, Couleurs et Commentaires), ces trois booléens ont un effet dans les fichiers exportés mais pas à l'écran.
- **Éléments graphiques** : comme les axes, les droites, les courbes, $\cdots$ Le dernier élément du menu (`utilisateur`) permet à l'utilisateur de combiner plusieurs éléments pour en fabriquer un nouveau. Chaque élément graphique est défini par une commande, et a ses propres attributs (couleurs et différents styles).

La commande permettant de définir un élément graphique peut comporter des noms de :

- **Variables Globales** : qui peuvent être créées par le biais du menu ou d'un clic droit de la souris sur la zone de dessin, ou le bouton `Nouv.` de la zone des variables globales (à droite de la fenêtre). 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[4].

La commande permettant de définir un élément graphique peut aussi comporter des noms de :

- **Macros** : ce sont des fonctions créées par l'utilisateur ou bien prédéfinies.

Par exemple, l'utilisateur peut créer une macro appelée `f` définie par la commande :

```
Si(%1=0,1,sin(%1)/%1)
```

%1 représente le premier paramètre de la macro, on définit ainsi la fonction

$$f : t \mapsto \begin{cases} 1 & \text{si } t = 0 \\ \frac{\sin(t)}{t} & \text{sinon} \end{cases}.$$

On peut alors créer un élément graphique *Courbe Paramétrée* (`Ctrl+P`) avec la commande `t + i * f(t)`[5], la courbe sera tracée avec les attributs choisis (ou les attributs par défaut). Si la macro `f` est modifiée alors tous les éléments graphiques (et toutes les variables globales) seront automatiquement recalculés.

Chaque élément graphique est donc généré à partir d'une commande. Les commandes de TeXgraph sont interprétées comme des **fonctions mathématiques** dont les images sont des **listes de complexes** (interprétés comme des affixes de points ou de vecteurs). Certaines fonctions peuvent donner une image égale à **Nil** (liste vide).

3. Par contre, les 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.

4. On peut désactiver le recalcul automatique d'un élément graphique en décochant l'option adéquate dans les attributs.

5. On peut également utiliser l'expression `t + i * \f(t)`, dans ce cas la macro `f` est immédiatement développée, sinon la macro `f` est appelée à chaque évaluation de l'expression.

Exemple(s) :

- Une liste d'affixes s'écrit entre crochets, ex : `[1+i,exp(i*2*pi/3),bar(z)]`, ou bien : `Liste(1+i,exp(i*2*pi/3),bar(z))`.
- Génération automatique de listes, ex : `Seq(exp(2*i*k*pi/5),k,0,4)` donne la liste des racines cinquièmes de l'unité.
- Certaines fonctions ont un effet graphique, ex : `Ligne([A,B,C],1)`[6] joint les points A, B et C par un segment et renvoie la valeur **Nil**, le second argument égal à 1 signifie que la ligne est fermée. Si aucune valeur n'a été affectée aux variables A, B ou C, la fonction est sans effet.

Les éléments graphiques sont créés à partir du menu (ou des raccoucis), il existe cependant une fonction (ou commande) qui crée un élément graphique `utilisateur`, c'est la commande :

NewGraph(<nom>, <commande>)

Si par exemple dans la ligne de commande au bas de la fenêtre on saisit :

`NewGraph("D1", "[Axes(0,1+i),Droite(1,-1,0)]")`

et que l'on valide, alors un élément graphique `utilisateur` appelé `D1` est créé avec la commande `[Axes(0,1+i),Droite(1,-1,0)]`, cette commande trace les axes avec comme point d'intersection le point d'affixe 0, une graduation en x égale à 1 et une graduation en y égale à 1, puis la droite d'équation $x-y=0$. La fonction **NewGraph** renvoie la valeur **Nil**. Cette fonction est dite **spéciale** car elle n'est pas utilisable partout, mais seulement dans certaines macros (dites macros spéciales[7]) et dans la ligne de commande au bas de la fenêtre.

2) Un exemple

Voici à titre d'exemple, un fichier source `*.teg` suivi du graphique correspondant (en LaTeX) et du fichier exporté. Les lignes commençant par le caractère % sont des commentaires et le caractère # est un séparateur. Mises à part les deux premières lignes, la syntaxe générale est :

<code interne>#<nom>#<commande>##<fin de ligne>

L'utilisateur peut avoir besoin d'éditer ce genre de fichier (dans l'éditeur de son choix) pour saisir plus facilement ses propres macros (dont le code interne est 16) ou des variables globales (dont le code interne est 15).

```
% Fenetre Xmin Xmax Ymin Ymax Xscale Yscale
100#-5#5#-1#3.5#1.000000000000#1.000000000000##
% Marges gauche droite haut bas cadre gestion_couleur
101#0.5#0.5#0.5#0.5#0#1##
% Affectation des Variables theta et phi
18##[Set(theta,0.5236),Set(phi,1.0472)]##
% Déclaration des Macros
16#f#exp(sin(%1))-1##
16#rectangles#[Set(pas, (%3-%2)/%4),
 Set(u, %2),
 Seq( [Assign(%1, t, u+pas/2), Set(y, %1),
          u, u+i*y, Inc(u, pas), u+i*y, u], k, 1, %4)
]##
% Déclaration des Eléments graphiques
%  (Axes)
18##[Set(Arrows,1)]##
1##[0+i*0,1+i*1]#0#0##
% X (Utilisateur)
18##[Set(Arrows,0),Set(FillStyle,2),Set(FillColor,0)]##
```

6. Attention : les fonctions graphiques n'ont d'effet que si elles sont utilisées lors de la création d'un élément graphique `utilisateur`, il est donc inutile de saisir `Ligne([A,B,C],1)` dans la ligne de commande en bas de la fenêtre !

7. Voir le chapitre sur les macros.

```
14#X#[n:= 20, L:= rectangles(\f(t), -5, 5, n),
for k from 1 to n do
aux:= Copy(L, 4*k-3, 4),
Ligne(aux, 1)
od
]##
% Cf (Courbe param.)
18##[Set(Width,8),Set(Color,255),Set(FillStyle,0),Set(FillColor,16777215)]##
3#Cf#t+i*\f(t)#5##
```

La macro `rectangles(f,a,b,n)` construit la liste des sommets de chaque rectangle (qui sont `u, u+i*y, u+pas+i*y, u+pas`) pour la fonction `f` avec la méthode du point médian, sur l'intervalle [a ;b] subdivisé en n intervalles.

L'élément graphique `utilisateur` (appelé `X`) exécute la macro `rectangles`, stocke la liste dans une variable `L`, puis « lit » les éléments de `L` par paquets de 4 points (affixes) et construit une ligne polygonale fermée sur ces 4 points, et ceci n fois (boucle `for`).

```
[n:=20,
L:=rectangles(\f(t),-5,5,n),
for k from 1 to n do
aux:=Copy(L, 4*k-3, 4),
Ligne(aux, 1)
od]
```

(Commande dans un élément graphique Utilisateur)

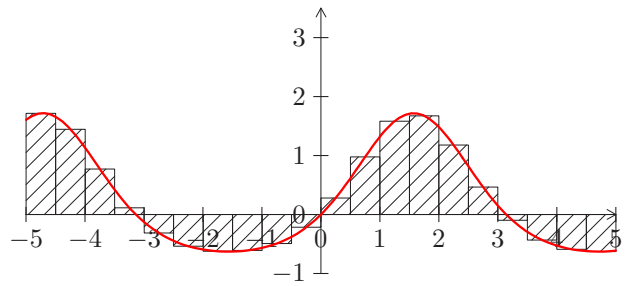


FIG. 1.2 – Méthode du point médian

Extrait du fichier exporté en LaTeX :

```
\unitlength 1cm
\begin{picture}(11,5.5)(-5.5,-1.5)
%  (Axes)
\thinlines
\path(4.8845,0.0667)(5,0)(4.8845,-0.0667)
\path(-0.0667,3.3845)(0,3.5)(0.0667,3.3845)
\path(-5,0)(5,0)
\path(0,-1)(0,3.5)
\multiput(0,-0.1)(1,0){6}{\line(0,1){0.2}}
\put(0,-0.2){\makebox(0,0)[t]{\small $0$}}
\put(1,-0.2){\makebox(0,0)[t]{\small $1$}}
\put(2,-0.2){\makebox(0,0)[t]{\small $2$}}
\put(3,-0.2){\makebox(0,0)[t]{\small $3$}}
\put(4,-0.2){\makebox(0,0)[t]{\small $4$}}
\put(5,-0.2){\makebox(0,0)[t]{\small $5$}}
\multiput(-1,-0.1)(-1,0){5}{\line(0,1){0.2}}
\put(-1,-0.2){\makebox(0,0)[t]{\small $-1$}}
\put(-2,-0.2){\makebox(0,0)[t]{\small $-2$}}
\put(-3,-0.2){\makebox(0,0)[t]{\small $-3$}}
\put(-4,-0.2){\makebox(0,0)[t]{\small $-4$}}
\put(-5,-0.2){\makebox(0,0)[t]{\small $-5$}}
\multiput(-0.1,0)(0,1){4}{\line(1,0){0.2}}
```

```
\put(-0.2,0){\makebox(0,0)[r]{\small $0$}}
\put(-0.2,1){\makebox(0,0)[r]{\small $1$}}
\put(-0.2,2){\makebox(0,0)[r]{\small $2$}}
\put(-0.2,3){\makebox(0,0)[r]{\small $3$}}
\multiput(-0.1,-1)(0,-1){1}{\line(1,0){0.2}}
\put(-0.2,-1){\makebox(0,0)[r]{\small $-1$}}
%X  (Utilisateur)
\path(-4.8745,1.7164)(-5,1.5908)\path(-4.6623,1.7164)(-5,1.3787)
\path(-4.5,1.6665)(-5,1.1665)\path(-4.5,1.4544)(-5,0.9544)
\path(-4.5,1.2423)(-5,0.7423)\path(-4.5,1.0302)(-5,0.5302)
\path(-4.5,0.818)(-5,0.318)\path(-4.5,0.6059)(-5,0.1059)
\path(-4.5,0.3938)(-4.8938,0)\path(-4.5,0.1816)(-4.6816,0)
....etc
```

III) Exportation des graphiques, packages et compilation

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 LaTeX

- 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).
- 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 + Dvipdfm (ou Dvipdfmx)

2) Format PsTricks

- Ces fichiers sont exportés avec l'extension .pst, ils utilisent les macros du package PsTricks.
- 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 package Pgf **version 1.0 minimum**.

- Exemple (minimal) :

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

4) 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 + Dvipdfm (ou Dvipdfmx), à condition que votre installation soit configurée pour que dvipdfm puisse convertir à la volée (avec epstopdf) l'image eps en image pdf.

5) 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 package 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{psfrag,graphicx}
\begin{document}
\input{MonGraph.psf}
\end{document}
```

- Compilations possibles :
  - LaTeX + Dvips
  - LaTeX + Dvips + Ps2pdf

6) Format Pdf

- Ces fichiers sont exportés avec l'extension .pdf. Il y a en réalité deux fichiers générés : TeXgraph crée un fichier eps puis TEXgraph exécute le programme epstopdf. Cela suppose que ce programme soit prévu dans votre distribution TEX et que votre système d'exploitation sait où il se trouve (ainsi que les fichiers utilisés par ce programme). Si vous utilisez un autre programme que epstopdf : il faut éditer le fichier de macros `TeXgraph.mac`, la dernière ligne de ce fichier doit être :

```
16#pdfprog#"epstopdf"##
```

Il suffit de remplacer epstopdf par le nom de votre programme et de sauvegarder le fichier (attention : il faut une fin de ligne après ##).

- 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} %extension non obligatoire
\end{document}
```

- Compilations possibles :
  - PdfLaTeX

IV) Les scripts : Apercu et CompileExporte

Ces deux scripts correspondent à deux boutons de la barre d'outils.

1) Apercu

Cliquer sur ce bouton provoque l'exécution de la macro `Apercu` du fichier TeXgraph.mac, la commande qui définit cette macro est :

```
[Export(pgf,[\InitialPath,"file.pgf"]),
 Exec("apercu.bat","",\InitialPath,0)
]
```

Le graphique en cours est donc exporté au format pgf dans le fichier *file.pgf*, dans le répertoire de TeXgraph, puis **on lance l'exécution du script *apercu.bat***, sans argument, dans le répertoire de TeXgraph. Le contenu du fichier *apercu.bat* est le suivant :

```
pdflatex --src -interaction=nonstopmode apercu.tex
PATH=PATH;c:\program Files\adobe\acrobat 7.0\reader
AcroRd32.EXE apercu.pdf
```

Ce script lance la compilation du fichier *apercu.tex* avec pdfLaTeX, puis ouvre le fichier créé : *apercu.pdf* dans Acrobat. Le contenu du fichier *apercu.tex* est :

```
\documentclass[a4paper]{article}
\usepackage[latin1]{inputenc}
\usepackage{pgf,amssymb,amsmath}
\pagestyle{empty}
\begin{document}
        \begin{figure}
        \centering
        \input{file.pgf}%
        \end{figure}
\end{document}
```

Bien entendu, ces deux fichiers peuvent être modifiés (ainsi que la macro `Apercu`), on les trouvera dans le répertoire de TeXgraph.

2) CompileExporte

Lorsque ce bouton est activé, le programme demande un nom pour le fichier créé au format pdf, appelons-le *Toto.pdf*. Le graphique et alors exporté au format pstricks dans un fichier appelé *file.pst* qui se trouve dans

le répertoire de TeXgraph, puis on lance **l'exécution du script contenu dans la macro** `CompileExporte` du fichier TeXgraph.mac, **avec les deux arguments suivants** :

Toto.pdf Toto.eps

Par défaut, cette macro contient la chaine : `"CompileExporte.bat"`, c'est donc le script :

`CompileExporte.bat Toto.pdf Toto.eps`

qui sera exécuté. Voici le script *CompileExporte.bat* :

```
latex --src -interaction=nonstopmode CompileExporte.tex
dvips -E CompileExporte.dvi
epstopdf CompileExporte.ps
Copy CompileExporte.pdf %1
Copy CompileExporte.ps %2
```

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

La compilation donne le fichier *CompileExporte.dvi* qui contient le graphique en cours, puis il est transformé par dvips en *CompileExporte.ps* qui est en réalité une image eps, cette image est elle-même transformée en *CompileExporte.pdf* par epstopdf. Pour terminer, l'image *CompileExporte.pdf* est renommée en *Toto.pdf* (argument $1), et *CompileExporte.ps* est renommée en *Toto.eps* (argument $2). On dispose ainsi de l'image aux formats eps et pdf avec les formules TeX compilées.

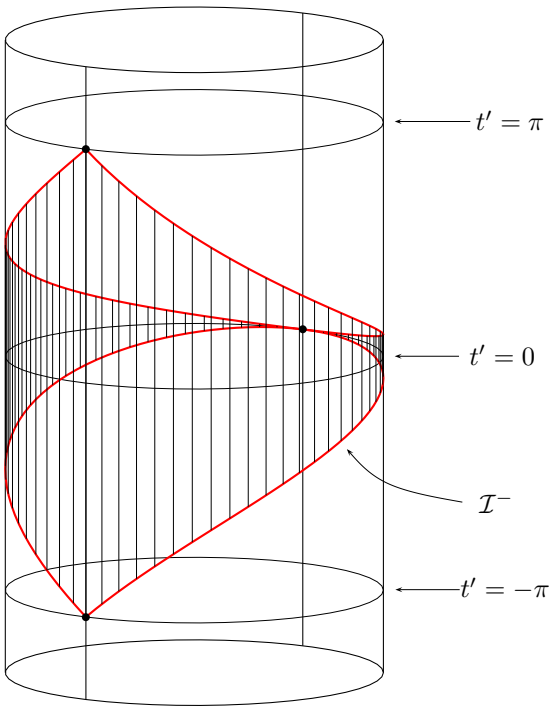


FIG. 1.3 – Un exemple exporté avec `CompileExporte`.

Chapitre 2

Le langage de TeXgraph

I) Généralités

1) Les commandes

Les commandes de TEXgraph sont en fait des **fonctions**[1] (au sens mathématique du terme) d'une ou plusieurs variables et dont les images sont des listes de complexes (éventuellement vides). Par conséquent les expressions manipulées sont des **expressions mathématiques**. La syntaxe générale d'une commande est :

`Liste(argument1, argument2, ···, argumentN)`

ou encore la forme équivalent suivante :

`[argument1, argument2, ···, argumentN]`

S'il y a un seul argument, les crochets sont inutiles. Chaque argument est à son tour une expression mathématique, comme par exemple une expression algèbrique : <arg1> <opération> <arg2>, ou une fonction : <fonction ou macro>(<arg1,···,<argN>), ou un complexe, ou encore une variable[2].

La fonction `Liste` est la « clé » du langage, elle évalue chaque argument et renvoie la liste des résultats[3]. Cette fonction permet également d'écrire des « séquences d'instructions », ce qui fait que l'on parle également de commande.

Exemple(s) :

- Supposons que l'on ait défini 3 variables globales : A, B et C, alors l'expression : `[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[4] :

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

  - il ne reste plus qu'à créer les trois variables A, B, C (si ce n'est déjà fait). Bien sûr on pouvait aussi créer séparément la ligne polygonale et les trois droites.

Les calculs sur les réels se font dans l'ensemble : $[-10^{38}, -10^{-37}] \cup [10^{-37}; 10^{38}]$.

TEXgraph est sensible à la casse, c'est à dire, il fait la distinction entre majuscules et minuscules.

1. y compris les macros.
2. Une variable est définie par un nom et une commande qui représente la « valeur » de la variable
3. Les évaluations donnant la valeur **Nil** ne figurent pas dans la liste des résultats, par exemple : `Liste(-1,1/0,1)` donnera le résultat `[-1,1]`.
4. Les fonctions `Ligne` et `Droite` renvoient la valeur **Nil** mais elles ont un effet graphique dans le contexte `utilisateur`.

2) Les identificateurs

Chaque objet de TEXgraph est identifié à l'aide d'un **identificateur**[5] (ou nom), celui-ci doit respecter les règles suivantes :

- Commencer par une lettre.
- Contenir au plus 10 caractères.
- Chaque caractère doit être : une lettre, ou un chiffre, ou une quote (apostrophe).

3) Interprétation sous forme de chaîne de caractères

TEXgraph ne connaît qu'un seul type de données : les listes de complexes. Mais pour les besoins de certaines fonctions, TEXgraph interprète certains arguments sous forme de chaînes de caractères. 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 cinq cas de figure :

- Les messages : ceux-ci sont délimités par les caractères `"` et `"`[6]. Leur interprétation sous forme de chaîne est directe[7].
- La macro `chaine()` : la commande définissant cette macro prédéfinie est un message[8]. TEXgraph remplace `chaine()` par le message correspondant.
- 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 chaine 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.
- Autre : 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 ", 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**.

II) 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`,
- et la boucle itérative : `for do od`.

Les deux boucles fonctionnent avec deux macros du fichier TeXgraph.mac. Signalons aussi que :

- la commande `Set` (affectation) peut être remplacée par := (par exemple, on peut écrire `x:=2` à la place de `Set(x,2)`).
- 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][9]. On peut aussi écrire `x[n,k]` à la place de `Copy(x, n, k)`, qui renvoie les k éléments de la liste x à partir du n-iéme, si k est nul, alors c'est toute la liste à partir du n-ième élément qui est renvoyée.

5. Celui-ci n'est pas considéré comme une chaîne de caractères et ne doit donc pas être mis entre deux guillemets.
6. Si le message doit contenir le caractère ", alors celui-ci doit être doublé : "".
7. Par contre l'évaluation numérique d'un message donne **Nil**, les variables de TEXgraph ne peuvent donc pas contenir de message.
8. Cette macro est utilisée pour mémoriser des messages, voir les macros spéciales.
9. Attention : `x[n]:=1` ne changera pas le n-ième élément de la liste x, car `x[n]` est une valeur numérique !

1) L'alternative

C'est l'équivalent de la commande `Si`. C'est une fonction qui renvoie la valeur **Nil**.

if <condition1> then <instructions> elif <condition2> then ... else <instructions> fi

<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)`[10].

2) La boucle conditionnelle

C'est une version de la commande `Loop`. C'est une fonction qui renvoie la valeur **Nil**.

while <condition1> do <instructions> od

<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`) 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**.

3) La boucle itérative

C'est une version des commandes `Seq` et `Map`. Il y a deux syntaxes :

for <variable> in <liste valeurs> do <instructions> od

Pour chaque valeur de la variable prise dans la liste, les instructions sont exécutées.

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 : $t \to t^m$ pour m variant dans la liste [-1,-0.25,0.5,2], pour chaque valeur de m on change également la couleur du tracé.

for <variable> from <valeur initiale> to <valeur finale> [step <pas>] do <instructions> od

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.

Exemple(s) : 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 $x = -2\pi$, $x = -2\pi + \frac{\pi}{2}$, ..., $x = 2\pi$.

10. dans cette deuxième version, f(t) est directement remplacée par son expression.

III) Les opérations

1) Opérations usuelles

Ce sont les opérations[11] : `+`, `-`, `*`, `/`. Ces symboles sont obligatoires dans les expressions, par exemple : `2x` [à la place de `2*x`] va générer une erreur. On dispose en plus de l'opération `^` qui correspond à la fonction puissance[12].

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.

Exemple(s) : `1 And 0` donne `0`, `2 Or 1` donne **Nil**.

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 complexes.
- **Inf** (ou encore <) : teste la relation « strictement inférieur à ».
- **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`.

4) Opérations d'intersection

Elles sont au nombre de deux :

- **Inter** : les deux arguments doivent être des listes de deux éléments[13], ils sont alors interprétés comme deux droites [définies par deux points], l'opération **Inter** détermine et renvoie le point d'intersection.
- **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 parcourt » que le premier argument** (et si plusieurs points sont sur le même segment alors ils sont rangés dans le sens de parcourt du deuxième argument).

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**.
- **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**.

Exemple(s) : `[1,2,3,4,5] CutA 3.5` donne `[1,2,3,3.5]`. `[1,2,3,4,5] CutA 6` donne **Nil**.

11. 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]`.
12. L'exposant doit être réel.
13. Il peut y en avoir plus, mais seuls les deux premiers sont pris en compte.

IV) Les fonctions

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

1) Fonctions d'une variable (réelle ou complexe)

i) abs(<argument>)

C'est la fonction module $[z \mapsto |z|]$ des complexes.

ii) arccos(<argument>), arcsin(<argument>), arctan(<argument>)

Fonctions circulaires réciproques usuelles à variable réelle.

iii) Arg(<argument>)

Fonction argument principal (dans l'intervalle $]-\pi;\pi]$).

iv) argch(<argument>), argsh(<argument>), argth(<argument>)

Fonctions hyperboliques réciproques usuelles à variable réelle.

v) bar(<argument>)

Conjugaison $[z \mapsto \overline{z}]$ des complexes.

vi) ch(<argument>) et cos(<argument>)

Cosinus hyperbolique et cosinus trigonométrique à variable réelle.

vii) Ent(<argument>)

Fonction partie entière à variable réelle.

viii) exp(<argument>)

Fonction exponentielle, l'argument est un **complexe**.

ix) Im(<argument>)

Fonction partie imaginaire, l'argument est un **complexe**.

x) ln(<argument>)

Fonction logarithme népérien, l'argument est un réel.

xi) opp(<argument>)

Fonction opposée $[z \mapsto -z]$, l'argument est un **complexe**.

xii) Rand([argument])

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*.

xiii) Re(<argument>)

Fonction partie réelle, l'argument est un **complexe**.

xiv) sh(<argument>) et sin(<argument>)

Ssinus hyperbolique et sinus trigonométrique, à variable réelle.

xv) sqr(<argument>)

Fonction carré [$z \mapsto z^2$], l'argument est un **complexe**.

xvi) sqrt(<argument>)

Fonction racine carrée, l'argument est un réel.

xvii) tan(<argument>), th(<argument>)

Tangente trigonométrique et tangente hyperbolique, à variable réelle.

2) Fonctions de plusieurs variables

i) Assign(<expression>,<variable>,<valeur>)

Cette fonction évalue *valeur* et l'affecte à la variable nommée *variable* dans *expression*[14]. La fonction **Assign** renvoie la valeur **Nil**. Cette fonction est utile dans l'écriture de macros admettant une fonction comme paramètre et qui doit être évaluée.

Exemple(s) : Voici la commande pour définir la macro `moyenne(f(t))` qui calcule $\frac{f(0)+f(1)}{2}$:

```
([Assign(%1,t,0),%1]+[Assign(%1,t,1),%1])/2.
```

L'exécution de `moyenne(t^2)` donne `0.5`, celle de `moyenne(u^2)` donne **Nil**, et l'exécution de `[Set(x,2), moyenne(t*x)]` donne `1`.

ii) Copy(<liste>,<depart>,<nombre>)

Renvoie la liste constituée par les *nombre* éléments de la *liste* à partir de l'élément numéro *depart* [inclus]. La *liste* n'est pas modifiée. Si *nombre* est nul, alors la fonction renvoie tous les éléments de la liste à partir de l'élément numéro *départ*.

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]`.

iii) Del(<liste>,<depart>,<nombre>)

Supprime de la *liste nombre* éléments à partir de l'élément numéro *départ* [inclus]. Si *nombre* est nul, alors la fonction supprime tous les éléments de la liste à partir de l'élément numéro *départ*. Le paramètre *liste* doit être **un nom de variable**, celle-ci est modifiée et la fonction **Del** renvoie **Nil**.

Exemple(s) : `[Set(x,[1,2,3,4]), Del(x,2,2), x]` renvoie `[1,4]`.

iv) Delay(<nombre millisecondes>)

Permet de suspendre l'exécution du programme pendant le laps de temps indiqué (en milli-secondes).

v) Der(<expression>,<variable>,<liste>)

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 suivante.

14. 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.

vi) Diff(<nom>,<expression>,<variable> [,param1,...,paramN])

Cette fonction permet de créer une macro appelée *nom*[15] et dont le corps 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 : `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 \[16]) . 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.

vii) Echange(<variable1>, <variable2>)

Cette fonction échange les deux variables, ce sont en fait les adresses qui sont échangées[17]. La fonction **Echange** renvoie la valeur **Nil**.

viii) Eval(<expression>)

Cette fonction évalue l'*expression* et renvoie le résultat. L'*expression* est interprétée comme une chaîne de caractères.

ix) Fenetre(<A>,<B> [,C])

Cette fonction permet de modifier la fenêtre graphique[18]. Le paramètre A est l'affixe du coin supérieur gauche, le paramètre B est l'affixe du coin inférieur droit, 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[19]. Cette fonction renvoie la valeur **Nil**.

x) Get(<argument>)

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**.

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. Par exemple : `Get(Cercle(0,1))` renvoie la liste des points du cercle de centre 0 et de rayon 1 mais sans créer ce cercle. Attention : même si l'élément graphique n'est pas réellement créé, il est néanmoins clippé par la fenêtre courante.

xi) GetAttr(<argument>)

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**. Si l'*argument* n'est pas un identificateur, alors il est interprété comme une chaîne de caractères et la fonction effectue la même recherche.

15. 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.
16. 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()`.
17. Les contenus ne sont pas dupliqués pour cet échange alors qu'ils le seraient si on utilisait la commande `[aux:=variable1, variable1:=variable2, variable2:=aux)]`.
18. C'est l'équivalent de l'option `Paramètres/Fenêtre`, sauf que les éléments graphiques ne sont pas automatiquement recalculés.
19. Ces deux valeurs doivent être strictement positives.

xii) Inc(<variable>, <expression>)

Cette fonction évalue *expression* et ajoute le résultat à *variable*. Cette fonction est plus avantageuse que : `Set(variable, variable+ expression)`, car dans cette dernière commande la *variable* est évaluée [*i.e.* dupliquée] pour calculer la somme. La fonction **Inc** renvoie la valeur **Nil**.

xiii) InputMac(<nom de fichier>)

Cette fonction permet de charger en mémoire un fichier de macros, celles-ci seront considérées comme **prédéfinies**. 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 de macros est dans le répertoire de TeXgraph, alors il est inutile de préciser le chemin.

xiv) Insert(<liste1>,<liste2> [,position])

Cette fonction insère la *liste2* dans la *liste1* à la position numéro *position*. Lorsque *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**.

xv) Int(<expression>, <variable>, <borne inf.>, <borne sup.>)

Cette fonction fait un calcul approché de l'intégrale de *expression* par rapport à *variable* sur l'intervalle **réel** défini par *borne inf.* et *borne sup.*

Exemple(s) : $\int_t^{2t} e^{\sin(u)}\,du$, s'écrira : `Int(exp(sin(u)), u, t, 2*t)`. 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.

xvi) Liste(<argument1>,...,<argumentn>)

Cette fonction évalue chaque argument et renvoie la liste des résultats.

xvii) Loop(<expression>,<condition>)

Cette fonction est une **boucle** qui construit une liste de la manière suivante : *expression* et *condition* sont évaluées jusqu'à ce que le résultat de *condition* soit égal à 1 [pour True] ou **Nil**, la fonction **Loop** renvoie la liste des résultats de *expression*.

xviii) Map(<expression>,<variable>,<liste>)

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.

xix) Message(<chaîne>)

Cette fonction affiche le paramètre *chaîne* [qui est donc interprété comme une chaîne de caractères] dans une fenêtre. Quand l'utilisateur a cliqué sur `OK`, la fenêtre se referme et la fonction renvoie la valeur **Nil**.

xx) Nops(<liste>)

Cette fonction évalue *liste* et renvoie le nombre d'éléments qui la compose.

xxi) ReadData(<fichier> [,balises])

Cette fonction ouvre un *fichier* texte en lecture, celui-ci est censé contenir des listes de points **du plan** sous forme de coordonnées (numériques), ces listes sont délimitées par des *balises*. La fonction lit le fichier et renvoie la liste des points (sous forme d'affixes) ou bien **Nil** si le fichier n'est pas trouvé ou ne contient pas de point. Le premier paramètre est interprété comme une chaîne de caractères qui contient le nom du fichier (plus éventuellement son chemin), le deuxième paramètre est également interprété comme une chaîne

qui est censée contenir deux caractères : le premier est la balise qui annonce une liste, et le second est la balise qui termine une liste[20], ce paramètre est facultatif et par défaut il n'y a pas de balises (donc une seule liste).

Attention : le retour chariot est considéré comme un séparateur de valeurs numériques.

xxii) RestoreAttr()

Restaure l'ensemble des attributs sauvegardés dans une pile par la commande *SaveAttr*.

xxiii) Rgb(<rouge>, <vert>, <bleu>)

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.

Exemple(s) : `Set(Color, Rgb(0.5,0.5,0.5))` sélectionne le gris.

xxiv) SaveAttr()

Sauvegarde sur une pile l'ensemble des attributs courants.

xxv) Seq(<expression>,<variable>,<départ>,<fin> [,pas])

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*[21]. Lorsque *variable* sort de l'intervalle, la boucle s'arrête et la fonction **Seq** renvoie la liste des résultats.

xxvi) Set(<variable>,<valeur>)

Cette fonction permet d'affecter à *variable*[22] la *valeur* spécifiée. La fonction **Set** renvoie la valeur **Nil**.

xxvii) Si(<condition1>,<expr1>,...,<conditionN>,<exprN> [,sinon])

Cette fonction évalue *condition1*[23], si celle-ci donne la valeur 1 alors la fonction évalue *expr1* et renvoie le résultat, sinon elle évalue *condition2*, si celle-ci donne la valeur 1 alors la fonction évalue *expr2*, 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**.

xxviii) Solve(<expression>,<variable>,<borne inf.>,<borne sup.>[,<n>])

Cette fonction fait une résolution approchée de 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 $\int_0^x e^{u^2}\,du = 1$: `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`.

20. Entre deux listes la constante `jump` est insérée.
21. Le pas peut être négatif mais il doit être non nul. Lorsqu'il n'est pas spécifié, sa valeur par défaut est 1.
22. 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, ...).
23. 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**.

xxix) Sort(< variable liste>)

Cette fonction trie la liste (de complexes) passée en argument dans l'ordre lexicographique, cette liste doit être une variable, et celle-ci sera modifiée. La fonction renvoie la valeur **Nil**.

Exemple(s) : si la variable `L` contient la liste `[-2,-3+i,1,1-2*i]` alors après l'exécution de `Sort(L)`, la variable contiendra la liste `[-3+i,-2,1-2*i,1]`. La méthode utilisée est le Quick Sort.

xxx) Str(<nom de macro>)

Lorsque l'expression contenant cette commande est interprétée comme une chaîne de caractères, 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)`.

xxxi) StrComp(<chaine1>,<chaine2>)

Les deux arguments sont interprétés comme une chaîne de caractères, puis les deux chaînes sont comparées, si elles sont égales alors la fonction renvoie la valeur 1, sinon la valeur 0.

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
```

xxxii) String(<expression>)

Lorsque l'expression contenant cette commande est interprétée comme une chaîne de caractères, cette fonction renvoie *expression* sous forme d'une chaîne. En dehors de ce contexte, la fonction **String** renvoie **Nil**.

xxxiii) Timer(<milli-secondes>)

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.

xxxiv) TimerMac(<corps de la macro à exécuter>)

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 revoie 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)].
```

3) Les macros mathématiques de TeXgraph.mac

Ce fichier est chargé automatiquement lors du lancement du programme, à condition qu'il soit présent dans le répertoire courant. Les variables et les macros de ce fichier sont donc considérées comme prédéfinies.

i) Opérations arithmétiques

- **div(<x>,<y>)** : renvoie l'unique entier k tel que $x - ky$ soit dans l'intervalle $[0; |y|[$.
```
if %2>0 then Ent(%1/%2)
elif %2<0 then -Ent(-%1/%2)
fi
```
- **mod(<x>,<y>)** : renvoie l'unique réel r de l'intervalle $[0; |y|[$ tel que $x = ky + r$ avec k entier.
```
%1-\div(%1,%2)*%2
```

ii) Opérations sur les variables

- **Abs(<x>)** : calcule la norme de x en cm.
```
abs(Re(%1)*Xscale+i*Im(%1)*Yscale)
```
- **free(<x>)** : libère la variable x en la mettant à **Nil**.
```
Set(%1,1/0)
```
- **nil(<x>)** : renvoie 1 si la variable x est à **Nil**, 0 sinon.
```
Nops([%1])=0
```
- **VarGlob(<affixe>)** : permet de définir une variable globale initialisée à *affixe*. Il est également possible d'afficher cette variable à l'écran en même temps. Par défaut, cette macro est associée au clic droit de la souris.
```
if Input("Entrez le nom de la variable, puis si voulez créer le label,
   indiquez la position du texte (N=nord, NO=nord-ouest...),
   puis la distance en cm [facultative, 0.25cm par défaut].
   La valeur de la variable sera l'affixe du point cliqué.
(et choisissez les attributs)
Exemple: A, NE", "Créer une variable",chaine())
then
 Eval([ "NewPoint(",%1,",",chaine(), ")" ])
fi
```

iii) Opérations sur les listes

- **length(<liste>)** : calcule la longueur de *liste* en cm.
```
[$long:=0, $first:=1/0,
 for $z in %1 do
  if z=jump then first:=1/0 else Inc(long, \Abs(z-first)), first:=z fi
 od,
 long
]
```
- **permute(<liste>)** : modifie la *liste* en plaçant le premier élément à la fin, *liste* doit être une variable.
```
[Append(%1,Copy(%1,1,1)),Del(%1,1,1)]
```
- **rectangle(<liste>)** : détermine le plus petit rectangle contenant la liste, cette macro renvoie le coin sup gauche suivi du coin inf droit.
```
[Set($Ok,0),
Map( Si(($z=jump)=0,Si(Ok=0,[Set($Xmin,Re(z)),Set($Xmax,Re(z)),
                             Set($Ymin,Im(z)),Set($Ymax,Im(z)),Set(Ok,1)],
                            [Si(Re(z)<Xmin, Set(Xmin,Re(z)),
                                Re(z)>Xmax, Set(Xmax,Re(z))),
                             Si(Im(z)<Ymin, Set(Ymin,Im(z)),
                                Im(z)>Ymax, Set(Ymax,Im(z)))]
                        )
        ), z, %1),
Xmin+i*Ymax,Xmax+i*Ymin]
```

- **replace(<liste>,<position>,<valeur>)** : modifie la variable *liste* en remplaçant l'élément numéro *position* par la *valeur*, cette macro renvoie **Nil**.
```
[Set($num,1),
 Set($retour, Map( [Si(num=%2, %3, $z), Inc(num, 1)], z, %1)),
 Echange(%1, retour)]
```
- **reverse(<liste>)** : renvoie la *liste* à l'envers.
```
Seq(Copy(%1,$k,1),k,Nops(%1),1,-1)
```

iv) Transformations géométriques

- **RealCoord(<affixe écran>)** : renvoie l'affixe réelle compte tenu des échelles.
```
 Re(%1)*Xscale+i*Im(%1)*Yscale}
```
- **ScrCoord(<affixe réelle>)** : renvoie l'affixe écran.
```
 Re(%1)/Xscale+i*Im(%1)/Yscale
```
- **TeXCoord(<screen affixe>)** : renvoie l'affixe exportée en tex, pst et pgf. Pour l'eps il y a la commande EpsCoord.
```
 (Re(%1)-Xmin)*Xscale+i*(Im(%1)-Ymin)*Yscale
```
- **sym(<liste>,<A>,<B>)** ou **sym(<liste>,[<A>,<B>])** : renvoie la liste des symétriques des points de *liste*, par rapport à la droite (AB).
```
[$A:=%2, if nil(%3) then $B:=A[2], A:=A[1] else B:=%3 fi,
 $U:=ScrCoord(i*RealCoord(B-A)),
 for $z in %1 do 2*([z,z+U] Inter [A, B])- z od]
```
- **proj(<liste>,<A>,<B>)** ou **proj(<liste>,[<A>,<B>])** : renvoie la liste des projetés orthogonaux des points de *liste* sur la droite (AB).
```
[$A:=%2, if nil(%3) then $B:=A[2], Del(A,2,1) else B:=%3 fi,
$U:=ScrCoord(i*RealCoord(B-A)),
for $z in %1 do [z,z+U] Inter [A, B] od]
```
- **med(<A>,<B>)** : renvoie une liste de deux points de la médiatrice de [A,B].
```
 [(%1+%2)/2, (%1+%2)/2+ScrCoord(i*RealCoord(%2-%1))]
```
- **bissec(<B>,<A>,<C>,<1 ou 2>)**, renvoie une liste de deux points de la bissectrice, 1=intérieure.
```
 [$u:=%1-%2, $v:=%3-%2,
 if %4=1 then %2, %2+u/Abs(u)+v/Abs(v)
 else %2, %2+u/Abs(u)-v/Abs(v)
fi]
```
- **parallel([<A>,<B>], <C>)** : renvoie une liste de deux points de la parallèle à (AB) passant par C.
```
 [$x:=%1, %2, %2+x[2]-x[1]]
```
- **perp([<A>,<B>], <C>)** : renvoie une liste de deux points de la perpendiculaire à (AB) passant par C.
```
 [$x:=%1, $u:=x[2]-x[1], %2, %2+ScrCoord(i*RealCoord(u))]
```
- **parallelo(<A>,<B>,<C>)** : renvoie la liste des sommets du parallèlogramme de sommets consécutifs A,B,C.
```
 [%1, %2, %3, %3+%1-%2]
```
- **rect(<A>,<B>,<C>)** : renvoie la liste des sommets du rectangle de sommets consécutifs A,B, le côté opposé passant par C.
```
 [$x:=proj(%3,%2,ScrCoord(i*RealCoord(%1-%2))+%2),%1,%2,x,x+%1-%2]
```
- **carre(<A>,<B>,<1 ou-1>)** : renvoie la liste des sommets du carré de sommets consécutifs A et B, 1=sens direct.
```
 [$U:=ScrCoord(i*RealCoord(%2-%1)), if %3<0 then U:=-U fi, %1,%2,U+%2,U+%1]
```
- **polyreg(<A>, <B>, <nombre de cotés>)** : renvoie la liste des sommets du polygône régulier de centre a, passant par B et avec le nombre de côtés indiqué.

```
 if %3>0 And Ent(%3)=%3 then
 $R:=RealCoord(%2-%1), $C:=RealCoord(%1),
 for $k from 1 to %3 do  ScrCoord(C+R*exp(i*k*2*pi/%3)) od
fi
```

V) Variables et constantes

1) Les constantes prédéfinies

- Les constantes mathématiques : **i**, π, **e**.
- La constante de saut : *jump*. Cette constante est utilisée pour séparer les différentes composantes connexes d'une ligne polygonale[24].

  Exemple(s) : la courbe d'équation $y = \frac{1}{x}$ 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)]`.
- Les constantes d'exportation : **tex**, **teg**, **pst**, **pgf**, **eps**, ce sont les valeurs possibles que peut prendre la « constante » **ExportMode** (qui est déterminée par TEXgraph au moment de l'exportation).
- 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**.
- Les constantes graphiques :
  - Les couleurs :
    - **black** [=0],
    - **white** [=1],
    - **red** [=2],
    - **green** [=3],
    - **blue** [=4],
    - **yellow** [=5],
    - **cyan** [=6],
    - **magenta** [=7],
    - **gray** [=8],
  - Styles de trait :
    - **noline** [=-1],
    - **solid** [=0],
    - **dashed** [=1],
    - **dotted** [=2],
  - Épaisseur du trait (en **nombre entier de dixième de point** de TEX) :
    - **thinlines** [=2],
    - **thicklines** [=8],
    - **Thicklines** [=14],
  - Styles de point :
    - **dot** [=0],

24. Signalons au passage que les lignes polygonales sont automatiquement « clippées » par TEXgraph avec le rectangle correspondant à la fenêtre courante.

  - **bigdot** [=1],
  - **cross** [=2].

- 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,
  - **framed** : le texte est centré et 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 est centré et peut contenir des sauts de paragraphes (\\),

  Exemple(s) : `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[25] :
  - **none** [=0] : pas de remplissage,
  - **full** [=1] : le polygone est rempli avec la couleur désignée par **FillColor**, ceci est sans effet en LaTeX.
  - **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**.

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 :

25. Le remplissage est « calculé » par TeXgraph pour la sortie LaTeX.

- **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()** peut forcer le recalcul de cet élément[26].
- 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'extrêmité 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**,
- **DotStyle** : style de point, initialisée à **dot**,
- **FillColor** : couleur du remplissage, initialisée à **white**,
- **FillStyle** : style de remplissage, initialisée à **none**,
- **LabelAngle** : orientation des labels par rappot à l'horizontale, c'est un angle en degré initialisé à 0,
- **LabelSize** : taille des labels, initialisée à **small**,
- **LabelStyle** : style de label, initialisée à **centered**,
- **LineStyle** : style de lignes, initialisée à **solid**,
- **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,
- **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[27]. Il est cependant toujours possible d'accéder à la liste des points composant un élément graphique avec la fonction **Get**. Mais cela nécessite que l'élément graphique dont on utilise le nom soit **déjà créé**, sinon la fonction **Get** renverra la valeur **Nil**.

Exemple(s) : dans la FIG 2.1 nous avons créé une `ligne polygonale` nommée `L`, définie par la commande `Seq(t+i/t,t,0,2*pi,0.1)` (une branche de l'hyperbole $y = \frac{1}{x}$) et la `courbe paramétrée C` définie par la commande `t+i*sin(2*t)*3/2`. Puis nous avons créé un élément graphique `utilisateur` dont la commande est à gauche du graphique.

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[28] (même si elle n'existe pas encore).

26. Ou bien le bouton **R** de la barre d'outils.
27. Ceci qui était le cas dans les versions 1.0 et 1.1.
28. Une macro sans paramètre s'utilise quand même avec deux parenthèses : `toto()`.

```
[Set($ligne1,Get(L)),
Set($ligne2, Get(C)),
Set($P, ligne1 InterL ligne2),
Set(DotStyle, bigdot), Point(P)]
```

(Commande dans un élément graphique Utilisateur)

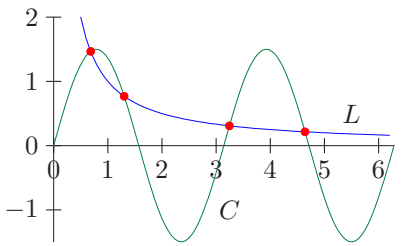


FIG. 2.1 – Utilisation du nom d'un élément graphique

- 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[29] 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 $[30] devant son nom : `$x1`.

Les variables globales se déclarent par l'intermédiaire du Menu, elles portent un nom et sont définies à partir d'une commande. 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...

4) 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`[31].

5) Les variables de TeXgraph.mac

- **stock** (=1/0) : Variable 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.
- **tailleB** (=92+i*20) : longueur d'un bouton et hauteur en pixels.
- **DeltaB** (=22*i) : écart entre deux boutons + hauteur d'un bouton.
- **RefPoint** (=18+29*i) : point de référence pour le premier bouton.
- **NbBoutons** (=0) : compteur de boutons.

29. 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.
30. Il suffit en fait de le mettre uniquement devant la première occurrence.
31. La modification de la fenêtre par le menu entraîne aussi le recalcul automatique.

Chapitre 3

Fonctions et macros graphiques

Ces fonctions créent un élément graphique au moment de leur évaluation et renvoient un résultat égal à **Nil**, elles ne sont utilisables **que dans le contexte « Utilisateur »**[1]

Dans les figures ci-dessous, nous n'avons indiqué que la commande définissant l'objet `Utilisateur`.

I) Fonctions prédéfinies

Voici la liste :

1) Arc(<B>,<A>,<C>,<rayon> [,sens])

Trace un arc de cercle de centre A, de rayon *rayon* partant de la droite (AB) jusqu'à la droite (AC), l'argument facultatif *sens* indique : le sens trigonométrique si sa valeur est 1 [c'est sa valeur par défaut], le sens contraire si valeur est -1. Voir Fig 3.1.

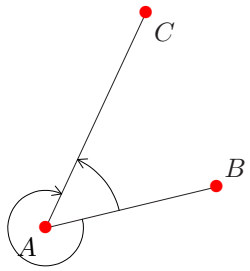
`[Set(Arrows,1), Arc(B,A,C,1,-1), Arc(B,A,C,2,1)]` (Commande dans un élément graphique Utilisateur)	

Fig. 3.1 – La fonction **Arc**

2) Axes(<Ox+i*Oy>,<Gradx+i*GradY>)

Dessine les axes, l'affixe du point d'intersection des axes est *Ox+i*Oy* et la graduation des axes est donnée par *Gradx+i*Grady*. Les axes occupent toute la fenêtre en cours[2].

3) Bezier(<liste de points>)

Trace une succession de courbes de Bezier (avec éventuellement des segments de droite). Il y a plusieurs possibilités pour la liste de points :

1. Option `Elément graphique/Créer/Utilisateur` du menu. Ces fonctions graphiques peuvent aussi être utilisées dans des macros, mais elles ne seront évaluées que si ces macros sont exécutées dans un contexte « utilisateur ».
2. En utilisant la commande `Fenetre` on peut faire varier la taille des axes.

a) 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 : $t[tA+(1-t)C]+(1-t)[tC+(1-t)B]$

b) une liste de 4 points ou plus : [A1,C1,C2,A2,C3,C4,A3...] : il s'agit alors d'une succession de courbes de Bezier à 2 points de contrôles, la première va de A1 à A2, elle est contrôlée par C1, C2 (paramétrée par $t\big(t[tA1+(1-t)C1]+(1-t)[tC1+(1-t)C2]\big)+(1-t)[tC2+(1-t)A2]$), 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.

`Bezier(A,C1,C2,B,jump,C,jump,D,jump,A)`

(Commande dans un élément graphique Utilisateur)

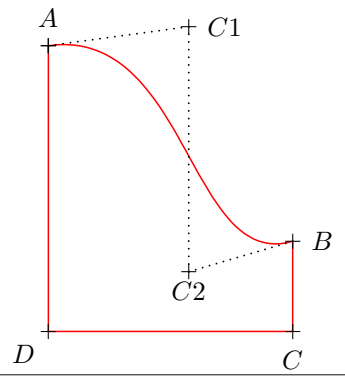


FIG. 3.2 – La fonction **Bezier**

4) Cercle(<A>,<r> [,B])

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 fonctions **Arc** et **Cercle**, on peut s'attendre à des surprises dans le résultat final si le repère n'est pas orthonormé !*[3]

5) Courbe(<expression> [,n,[,1]])

Trace la courbe paramétrée par *expression* qui est une **fonction de t**. Pour une courbe cartésienne du type $y=f(x)$, *expression* peut être égale à `t+i*f(t)`, pour une courbe paramétrée par $x(t)$ et $y(t)$, *expression* peut être égale à `x(t)+i*y(t)` et pour une courbe polaire paramétrée par $r(t)$, *expression* peut être égale à `r(t)*exp(i*t)`.

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 seuil égal à $\frac{\text{tMax}-\text{tMin}}{\text{NbPoints}-1}$ 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.

Exemple(s) : courbe cartésienne d'équation $y=x\text{E}(\frac{1}{x})$ (E désigne la partie entière) avec les attributs suivants : `tMin=-2, tMax=2, NbPoints=50` : voir FIG 3.3.

6) Droite(<A>,<B> [,C])

Trace la droite (AB) lorsque le troisième argument C est omis, sinon c'est la droite d'équation cartésienne $Ax+By=C$.

3. Le repère est orthonormé lorsque les paramètres `Xscale` et `Yscale` sont égaux, voir option `Paramètres/Fenêtre`.

```
Courbe(t+i*t*Ent(1/t),5,1)
```

(Commande dans un élément graphique Utilisateur)

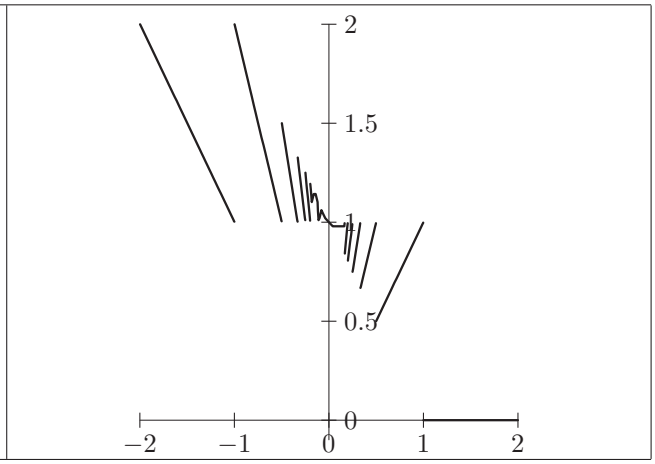


FIG. 3.3 – La fonction **Courbe**

Exemple(s) : développée d'une ellipse (ou enveloppe des normales, la normale au point M est la bissectrice intérieure de FMF'), `f` est la macro définie par `4*cos(%1)+3*i*sin(%1)`, `F` une variable égale $\sqrt{7}$ et `F'` une variable égale à `-F` (foyers de l'ellipse paramétrée par $f(t)$) : voir FIG 3.4.

```
Seq(
[Set(M, f(t)),
Droite(M,M+(F-M)/abs(F-M)+
(F'-M)/abs(F'-M))
],t,-pi,pi,0.1)
```

(Commande dans un élément graphique Utilisateur)

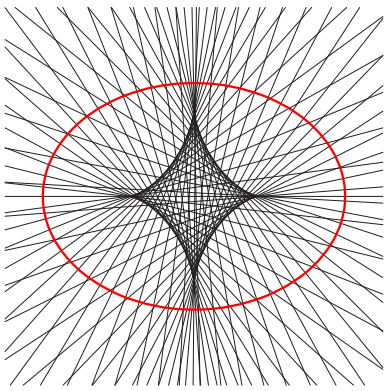


FIG. 3.4 – La fonction **Droite**

7) EquaDif(<f(t,x,y)>,<t0>,<x0+i*y0> [,mode])

Trace une solution approchée de l'équation différentielle : $x'(t) + iy'(t) = f(t, x, y)$ avec la condition initiale $x(t0) = x0$ et $y(t0) = y0$. 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 $x'' + tx' + x = \sin(t)$ avec la condition initiale $x(0) = -1$ et $x'(0) = \frac{1}{2}$, 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 $Y = x'$: voir FIG 3.5.

8) Grille(<Ox+i*Oy>,<Gradx+i*GradY>)

Dessine une grille de repérage, l'affixe du point de référence de la grille est *Ox+i*Oy* et les divisions en x et y sont données par *Gradx+i*Grady*. La grille est dessinée en gris clair, elle occupe toute la fenêtre en cours.

`EquaDif(y+i*(t*x+y+sin(t)),0,i/2,1)`

(Commande dans un élément graphique Utilisateur)

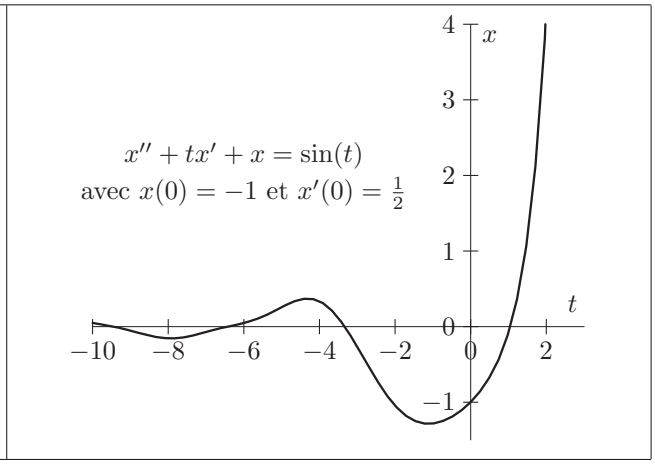


FIG. 3.5 – La fonction **EquaDif**

9) Implicit(<f(x,y)> [,n,m])

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 $n = m = 25$. Sur chaque pavé ainsi obtenu on teste la méthode de NEWTON et dans l'affirmative, on trace l'intersection de la tangente[4] et du pavé[5].

Exemple(s), courbe implicite d'équation $\sin(xy) = 0$: voir FIG 3.6.

`Implicit(sin(x*y))`

(Commande dans un élément graphique Utilisateur)

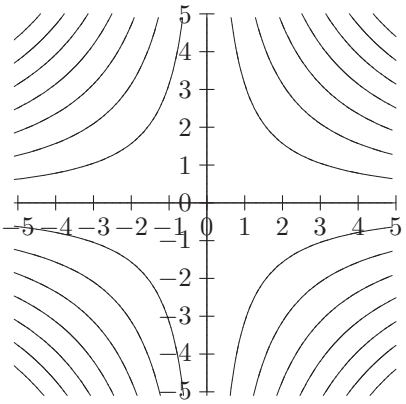


FIG. 3.6 – La fonction **Implicit**

10) Label(<affixe1>,<texte1>,···,<affixeN>,<texteN>)

Place la chaîne de caractères *texte1* à la position *affixe1* etc... Les paramètres *texte1*, ..., *texteN* sont donc interprétés comme des chaînes de caractères. Voir FIG 3.7.

11) Ligne(<liste>,<fermée> [,rayon])

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 [il vaut 0 par défaut], alors les « angles » de la ligne polygonale sont arrondis avec un arc de cercle dont le rayon correspond à l'argument. Voir FIG 3.8.

4. Sauf pour les points critiques.

5. La méthode a le mérite d'être élémentaire mais elle est plutôt longue, pour tracer la courbe d'équation $\int_x^y e^{u^2}\,du = 1$ par exemple, il faut être très patient, on a plutôt intérêt à passer par l'équation différentielle $x'(t) + iy'(t) = 1 + i\exp(x^2 - y^2)$ avec la condition initiale $x(0) = 0$ et $y(0) =$`Solve(Int(exp(u^2),u,0,t)-1,t,0,1)`.

```
[Ligne(
Seq(3*exp(2*i*k*pi/5),k,0,4),1),
Seq(
Label(3.35*exp(2*i*k*pi/5),
["$A_{",k+1,"}$"]),
k,0,4)]
```

(Commande dans un élément graphique Utilisateur)

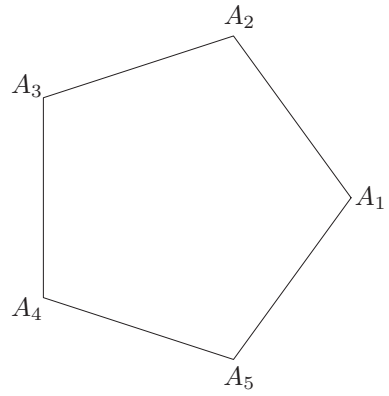


FIG. 3.7 – La fonction **Label**

```
[L:=[0,1,i],
for k from 1 to 5 do
L:=[Map(z/2,z,L), Map((1+z)/2,z,L),
Map((i+z)/2,z,L)]
od,
for k from 1 to 3^5 do
Aux:=Copy(L,3*k-2,3), Ligne(Aux,1)
od]
```

(Commande dans un élément graphique Utilisateur)

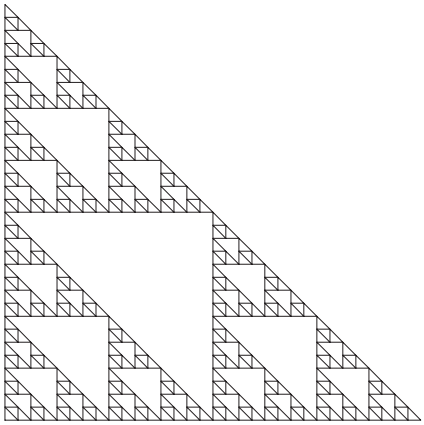


FIG. 3.8 – La fonction **Ligne** (triangle de SIERPINSKI à l'ordre 5)

12) Point(<A1>,...,<An>)

Représente le nuage de points *A1* ... *An*. Voir FIG 3.9.

```
Map([Set(M,t-sin(t)+i*(1-cos(t))),
Set(I,t+i),Set(DotStyle,cross),
Point(I),Set(DotStyle,bigdot),
Point(M), Cercle(I,1), Seg(M,I),
Set(Arrows,1),Arc(M,I,t,0.5,t),
Set(Arrows,0),Set(LineStyle,dashed),
Seg(I,t),Set(LineStyle,solid)],
t,[-4,-1.85,0,1.85,3])
```

(Commande dans un élément graphique Utilisateur)

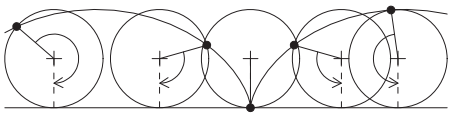


FIG. 3.9 – La cycloïde

13) Spline(<V0>,<A0>,$\cdots$,<An>,<Vn>)

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].

II) Macros graphiques de TeXgraph.mac

Ces macros n'ont d'effet que dans un contexte `Utilisateur`.

1) angleD(<B>,<A>,<C>,<r>)

Dessine l'angle $\widehat{BAC}$ avec un parallèlogramme de coté r.

```
[Set($angleDu,(%1-%2)/abs(%1-%2)),Set($angleDv,(%3-%2)/abs(%3-%2)),
 Ligne([%2+%4*angleDu,%2+%4*(angleDu+angleDv),%2+%4*angleDv],0)]
```

2) arc(<B>,<A>,<C>,<r> [,sens])

A le même effet que la fonction graphique **Arc** mais le fichier exporté en LaTeX n'utilisera pas la macro `\arc` car celle-ci n'est pas toujours correctement traduite en Pdf par le traducteur de MiKTeX [dviPdfm].

```
Si(nil(%5), Ligne(Get(Arc(%1,%2,%3,%4,1)),0),
            Ligne(Get(Arc(%1,%2,%3,%4,%5)),0))
```

3) Clip(<liste>

Permet de clipper les éléments graphiques déjà dessinés avec la *liste* qui doit être une courbe fermée et simple.

```
[ $Ok:=0, $S:=%1,
  for $z in S do {on cherche le point le plus à gauche}
    if ($z=jump)=0 then
if Ok=0 then $xmin:=Re(z), $y:=Im(z), Ok:=1
        else if Re(z)<xmin then xmin:=Re(z), y:=Im(z) fi
                 fi
   fi
 od,
$X:=xmin+i*y, $L:=[ S CutB X, S CutA X], $Y:=L[3],
$oldfillstyle:= FillStyle, $oldlinestyle:= LineStyle,
FillStyle:=full, LineStyle:=noline,
$coinS:=Xmin+i*Ymax, $coinI:=Xmax+i*Ymin,
Fenetre(coinS-margeG/2+i*margeH/2, coinI+margeD/2-i*margeB/2),
if Re(bar(i)*(Y-X)) <=0  {sens direct?}
then
   Ligne( [L, Xmin+i*y, Xmin+i*Ymax, Xmax+i*Ymax, Xmax+i*Ymin,Xmin+i*Ymin,Xmin+i*y,X],1)
else
   Ligne( [L, Xmin+i*y, Xmin+i*Ymin, Xmax+i*Ymin, Xmax+i*Ymax,Xmin+i*Ymax,Xmin+i*y, X],1)
fi,
Fenetre(coinS,coinI),
FillStyle:=oldfillstyle, LineStyle:= oldlinestyle
]
```

4) Ddroite(<A>,<B>)

Dessine la demi-droite [A,B).

```
[Set($L, Get(Droite(%1,%2))), Set($u, Copy(L,1,1)), Set($v, Copy(L,2,1)),
 Si( Re( bar(%2-%1)*(v-u))>=0, Ligne([%1,v],0), Ligne([%1,u],0))
]
```

5) domaine1(<f(t)> [,a,b [,divisions]])

Délimite la partie du plan comprise entre la courbe C_f, l'axe Ox et les droites $x = a$, $x = b$ si a et b sont précisés, sinon $x =$`tMin` et $x =$`tMax`. ATTENTION : les courbes sont clippées par la fenêtre !

```
Ligne([Set(FillStyle,0),Set($nbdiv,Si(nil(%4),0,%4)),
       Si(nil(%2)=0 And nil(%3)=0,
          [Set(tMin,%2),Set(tMax,%3)]),
           tMin, Map(Si((z=jump)=0,z),z,Get(Courbe(t+i*%1,nbdiv))),tMax
      ],1)
```

6) domaine2(<f(t)>,<g(t)> [,a,b [,divisions]])

Délimite la partie du plan comprise entre les courbes C_f, C_g et les droites $x = a$, $x = b$ si a et b sont précisés, sinon $x =$`tMin` et $x =$`tMax`. ATTENTION : les courbes sont clippées par la fenêtre !

```
Ligne([Set(FillStyle,0),Set($nbdiv,Si(nil(%4),0,%4)),
       Si(nil(%3)=0 And nil(%4)=0,[Set(tMin,%3),Set(tMax,%4)]),
       Map(Si((z=jump)=0,z),z,Get(Courbe(t+i*%1,nbdiv))),
       reverse(Map(Si((z=jump)=0,z),z,Get(Courbe(t+i*%2,nbdiv))))
       ],1)
```

7) domaine3(<f(t)>,<g(t)> [,divisions])

Délimite la partie du plan comprise entre les courbes C_f et C_g avec t dans l'intervalle [`tMin`, `tMax`], en recherchant les points d'intersection. ATTENTION : les courbes sont clippées par la fenêtre !

```
Ligne(
[Set(FillStyle,0),Set($nbdiv,Si(nil(%3),0,%3)),
 Set($C1 Map(Si(($z=jump)=0,z),z,Get(Courbe($t+i*%1,nbdiv)))),
 Set($C2 Map(Si((z=jump)=0,z),z,Get(Courbe(t+i*%2,nbdiv)))),
 Set($P,C1 InterL C2),
 Set($A,Copy(P,1,1)),Set($B,Copy(P,Nops(P),1)),
 A,Map(Si(Re(z)>=Re(A) And Re(z)<=Re(B),z),z C1),B,
 reverse(Map(Si(Re(z)>=Re(A) And Re(z)<=Re(B),z),z,C2))
],1)
```

8) flecher(<liste>,<pos1>,...,<posN>)

Dessine des flèches le long de la ligne polygonale *liste*, la position de chaque flèche (pos1, ...) est un nombre entre 0 et 1 (0= début de la ligne et 1= 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.

```
[ $liste:=%1, $L:=\length( liste),
  $aux:= %2, Sort(aux), $long:=0,  $first:=Copy(liste,1,1), $k:=1, Arrows:=1,
  for $x in aux do
    if 0<=Re(x) And Re(x)<=1 then
   $L':=Re(x)*L,
if L'>0 then
   while long<L'  do
Inc(k,1), $z:=Copy(liste,k,1),
if z=jump then first:=1/0 else Inc(long, \Abs(z-first)), first:=z
fi
od,
u:=z-Copy(liste,k-1,1), u:=u/\Abs(u)
     else u:=Copy(liste,k+1,1)-first, u:=u/\Abs(u)
fi,
```

```
   $P:= first+(L'-long)*u,
   Ligne( [ P+(1-2*(Im(x)=0))*u/100, P], 0)
   fi
  od
]
```

9) GradDroite(<A>,<u>,<hautDiv>,<nbSubdiv> [,<pos.>,<orient.> [,<facteur> [,<texte>]]])

Gradue la droite passant par A et dirigée par u, hautdiv est la hauteur des graduations (cm), nbSubdiv est le nombre de subdivisions, pos indique la position des labels (1=dessous, 2=dessus), si *orient*=i les labels sont orthogonaux à la droite, sinon *orient* représente le LabelStyle, chaque abscisse est multipliée par facteur et accompagnée du texte.

Exemple(s) : `GradDroite(0,i*pi,xyticks,2,2,right,1,"\pi")` et `GradDroite(0,pi,xyticks,2,1,i,1,"\pi")` pour avoir des axes gradués de π en π.

10) LabelDot(<affixe>,<texte>,<orientation>, <DrawDot> [,<distance>])

Cette macro affiche un texte à coté d'un point (affixe). L'orientation peut être "N" pour nord, "NE" pour nord-est ...etc. Le point est également affiché lorsque DrawDot=1 et on peut définir la distance entre le point et le texte (0.25cm par défaut).

```
[ $frame:=framed*mod(div(LabelStyle,16),2),
if nil(%5) then $d:=0.25 else $d:=%5 fi,
 oldstyle:= LabelStyle,
if StrComp(%3,"N") then $angle:=pi/2, LabelStyle:=bottom+frame
elif StrComp(%3,"NO") then angle:=3*pi/4, LabelStyle:=bottom+right+frame
elif StrComp(%3,"O") then angle:=pi, LabelStyle:=right+frame
elif StrComp(%3,"SO") then angle:=5*pi/4, LabelStyle:=top+right+frame
elif StrComp(%3,"S") then angle:=-pi/2, LabelStyle:=top+frame
elif StrComp(%3,"SE") then angle:=-pi/4, LabelStyle:=top+left+frame
elif StrComp(%3,"E") then angle:=0, LabelStyle:=left+frame
elif StrComp(%3,"NE") then angle:=pi/4, LabelStyle:=bottom+left+frame
fi,
$x:=%1+ d*cos(angle)/Xscale+i*d*sin(angle)/Yscale,
if %4=1 then Point(%1) fi, Label(x, %2), LabelStyle:=oldstyle
]
```

11) markangle(<B>,<A>,<C>,<r>,<n>,<espacement>,<longueur>)

Même chose que markseg pour marquer un arc de cercle.

```
[Set($b,%1),Set($a,%2),Set($c,%3),
 Set($r,%4),Set($n,%5),Set($esp,%6/r),Set($long,%7/2),
 Set($dep,Arg(b-a)+(Arg((c-a)/(b-a))-(n-1)*esp)/2),
 Seq([Set($t,exp(i*(dep+k*esp))),Ligne([a+(r+long)*t,a+(r-long)*t],0)
     ],k,0,n-1)]
```

12) markseg(<A>,<B>,<n>,<espacement>,<longueur>)

Marque le segment $[A, B]$ avec n petits segments (penchés de 45 degrés), l'espacement et la longueur sont en unité graphique.

```
[$a:=%1, $b:=%2, $n:=%3, $esp:= %4, $long:=%5,
$A:=Re(a)*Xscale+i*Im(a)*Yscale, B:= Re(b)*Xscale+i*Im(b)*Yscale,
$v:= (b-a)/abs(B-A), $U:= long/2*exp(i*pi/4)*((A-B)/abs(A-B)),
$u:=Re(U)/Xscale+i*Im(U)/Yscale,
```

```
$c:= a+(abs(B-A)-(n-1)*esp)*v/2, $pas:= esp*v,
for k from 1 to n do Seg(c-u, c+u), Inc(c,pas) od
]
```

13) periodic(<f(t)>,<a>,<b> [,divisions, discontinuités])

Trace la courbe de la fonction périodique définie par *f(t)* 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).

```
[Set($oldtMin, tMin), Set($oldtMax, tMax),
 Set($Ndiv, Si(nil(%4), 5, %4)), Set($disc, Si(nil(%5), 0, %5)),
 Set(NbPoints, Ent(NbPoints*(%3-%2)/(tMax-tMin))+2),
 Echange($oldstock, stock), Set(tMin,%2), Set(tMax,%3),
 Set($oldXmin, Xmin), Set($oldXmax, Xmax),
 Fenetre(tMin+i*Ymax, tMax+i*Ymin),
 Set(stock, Get(Courbe(t+i*%1, Ndiv, disc))),
 Fenetre(oldtMin+i*Ymax, oldtMax+i*Ymin),
 Set($T, %3-%2),
 Set($k1, Ent((oldtMin-%2)/T)), Set($k2, Ent((oldtMax-%2)/T)+1),
 Seq( Ligne( Map( $z+$k*T, z, stock),0), k, k1, k2),
 Fenetre(oldXmin+i*Ymax, oldXmax+i*Ymin),
 Echange(stock, oldstock), Echange(tMin, oldtMin), Echange(tMax,oldtMax)
]
```

14) Seg(<A>,<B>)

Dessine le segment $[A, B]$.

```
Ligne([%1,%2],0)
```

15) suite(<f(t)>,<u0>,<n>)

Représentation graphique de la suite définie par $u_{n+1} = f(u_n)$, de premier terme u_0 et jusqu'au rang n. On ne représente que les « escaliers ».

```
Ligne( [Set($x,%2),x,
        Seq([Assign(%1,t,x),Set($y,%1),
             Si(nil(y),Set($k,%3),[x+i*y,y*(1+i),Echange(x,y)])
            ],k,1,%3)
       ],0)
```

16) tangente(<f(t)>,<t0> [,longueur])

Trace la tangente à la courbe cartésienne Cf au point d'abscisse t_0, on trace un segment de longueur indiquée ou la droite si la longueur est omise.

```
[Set($long,%3/2),Assign(%1,t,%2),Set($M,%2+i*%1),Set($df,1+i*Der(%1,t,%2)),
 Si(nil(long),Droite(M,M+df),[Set(df,df/abs(df)),Seg(M-long*df,M+long*df)])
]
```

17) tangenteP(<f(t)>,<t0> [,longueur])

Trace la tangente à la courbe paramétrée par $f(t)$ au point de paramètre t_0, on trace un segment de longueur indiquée, ou la droite si la longueur est omise.

```
[Set($long,%3/2),Assign(%1,t,%2),Set($M,%1),Set($df,Der(%1,t,%2)),
 Si(nil(long),Droite(M,M+df) [Set(df,df/abs(df)),Seg(M-long*df,M+long*df)])]
```

III) 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.

Pour TeXgraph, un point ou un vecteur de coordonnées (x, y, z) est représenté par la liste :

$$[x + i * y, z]$$

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**.

1) Variables prédéfinies

Variables prédéfinies relatives à la représentation en 3D :

- **theta** et **phi** : utilisées pour les calculs de projections des surfaces, elles sont initialisées respectivement à $\frac{\pi}{6}$ et $\frac{\pi}{3}$, 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.
- **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 $\frac{\pi}{36}$ (soit 5 degrés).

2) Fonctions prédéfinies

Fonctions prédéfinies relatives à la représentation en 3D :

i) Proj3D(< liste de points 3D >)

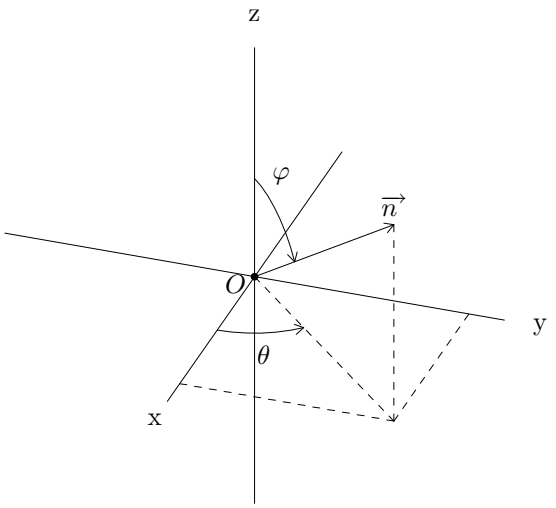
La fonction mathématique **Proj3D** calcule et renvoie la liste des projetés des points 3D sur le plan passant par l'origine et normal au vecteur :

$$\overrightarrow{n}(\sin(\varphi)\cos(\theta), \sin(\varphi)\sin(\theta), \cos(\varphi))$$

dirigé vers l'observateur, l'angle θ est la longitude et l'angle φ est la colatitude. Le projeté est calculé à partir des formules (projection orthographique) :

$$\begin{cases} X &= \cos(\theta)y - \sin(\theta)x \\ Y &= -\cos(\varphi)\cos(\theta)x - \cos(\varphi)\sin(\theta)y + \sin(\varphi)z \end{cases}.$$

le résultat est le complexe $X + iY$.



La liste de points 3D peut contenir la constante de saut `jump`, elle sera recopiée dans le résultat.

Exemple(s) :

- La commande `Proj3D( [[0,1], [1+i,0], [1-2*i,2]] )` renvoie la liste :

  `[0.866025*i,0.366025-0.683013*i,-2.232051+1.799038*i]`

- Représentation d'une courbe gauche paramétrée par $x(t), y(t)$ et $z(t)$: créer un élément graphique `Courbe/Paramétrée` avec la commande `Proj3D([x(t)+i*y(t), z(t)])`.
- La projection orthographique étant linéaire, elle conserve les barycentres, on peut donc dessiner une courbe de Bezier dans l'espace en utilisant la fonction Bezier 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 Bezier d'extrêmités A et B avec C comme point de contrôle.

ii) Surface(<x(u,v)>,<y(u,v)>,<z(u,v)> [,uMin+i*uMax,vMin+i*vMax, uNbLg+i*vNbLg,degrade])

Cette fonction graphique trace la surface paramétrée par *x(u,v)*, *y(u,v)* et *z(u,v)*. Le quatrième paramètre représente l'intervalle du paramètre u ($[-5, 5]$ par défaut), le cinquième paramètre représente l'intervalle du paramètre v ($[-5; 5]$ par défaut), le sixième paramètre représente sous forme complexe, le nombre de lignes pour u et le nombre de lignes pour v (25 lignes par défaut), et le dernier paramètre est un booléen (qui doit valoir 1 ou 0) qui indique si le tracé doit être fait en utilisant ou non un dégradé de couleurs (valeur 1 par défaut).

La représentation à l'écran et l'exportation aux formats Pgf, PsTricks, Eps (et donc Psf et Pdf) utilisent l'algorithme du peintre[6]. L'exportation au format LaTeX élimine les facettes dont le centre n'est pas visible, et dessine les lignes de coordonnées ; le résultat n'est pas toujours parfait lorsqu'il y a des facettes dont une partie seulement est cachée, car toute facette considérée comme visible est entièrement dessinée, et une facette considérée comme cachée n'apparaît pas du tout. D'autre part, comme il n'y a pas de remplissage en LaTeX, la surface exportée est transparente.

3) Les variables de TeXgraph.mac pour la 3D

- **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).

4) Les macros mathématiques de TeXgraph.mac pour la 3D

- **dot(<x>,<y>,<z>)** : désigne le point de coordonnées (x,y,z), celui-ci est representé par la liste : [x+i*y, z].

```
[Set($u,%1+i*%2), Si((nil(u)=0) And (nil(%3)=0),
Si( (u=jump)=0 And (%3=jump)=0, [%1+i*%2,%3]))]
```

- **n()** : vecteur unitaire normal au plan de projection, dirigé vers l'observateur.

```
[sin(phi)*exp(i*theta),cos(phi)]
```

- Macros donnant les coordonnées d'un point 3D :
  - **Xde(<point3D>)** : renvoie l'abscisse, elle est définie par la commande `Re(Copy(%1,1,1))`.
  - **Yde(<point3D>)** : renvoie l'ordonnée, elle est définie par la commande `Im(Copy(%1,1,1))`.
  - **Zde(<point3D>)** : renvoie la cote, elle est définie par la commande `Copy(%1,2,1)`.
- Macros de projection d'un point 3D sur les axes :
  - **px(<point3D>)** : projeté sur Ox, elle est définie par la commande `[\Xde(%1),0]`.

6. Les facettes sont dessinées (et éventuellement remplies) partant de la plus éloignée (cote minimale sur l'axe $(O, \vec{n})$) jusqu'à la plus proche.

  - **py(<point3D>)** : projeté sur Oy, elle est définie par la commande `[i*\Yde(%1),0]`.
  - **pz(<point3D>)** : projeté sur Oz, elle est définie par la commande `[0,\Zde(%1)]`.

- Macros de projection d'un point 3D sur les trois plans du repère :
  - **pxy(<point3D>)** : projeté sur xOy, elle est définie par la commande `[Copy(%1,1,1),0]`.
  - **pxz(<point3D>)** : projeté sur xOz, elle est définie par la commande `[\Xde(%1),Copy(%1,2,1)]`.
  - **pyz(<point3D>)** : projeté sur yOz, elle est définie par la commande `[i*\Yde(%1),Copy(%1,2,1)]`.

- **prodvec(<point3D1>, <point3D2>)** : renvoie le résultat du produit vectoriel entre les deux vecteurs.
```
[Set($ppu,%1),Set($ppv,%2),
Set($ppx,Re(Copy(ppu,1,1))), Set($ppy,Im(Copy(ppu,1,1))),
Set($ppz,Copy(ppu,2,1)),
Set($ppx',Re(Copy(ppv,1,1))), Set($ppy',Im(Copy(ppv,1,1))),
Set($ppz',Copy(ppv,2,1)),
(ppy*ppz'-ppy'*ppz)+i*(ppx'*ppz-ppx*ppz'),ppx*ppy'-ppx'*ppy
]
```

- **prodscal(<point3D1>, <point3D2>)** : renvoie le résultat du produit scalaire entre les deux vecteurs.
```
Re(bar(Copy(%1,1,1))*Copy(%2,1,1))+Copy(%1,2,1)*Copy(%2,2,1)
```

- **det(<point3D1>, <point3D2>, <point3D3>)** : renvoie le déterminant des trois vecteurs.
```
\prodscal(%3,\prodvec(%1,%2))
```

- **norm(< point3D>)** : renvoie la norme du vecteur.
```
[Set($sum,0), Map(Inc(sum,sqr(abs($z0))),z0,%1),sqrt(sum)]
```

- **normalize(< point3D>)** : renvoie le vecteur normalisé.
```
%1/norm(%1)
```

- **angle(<point3D1>, <point3D2>)** : renvoie l'écart angulaire entre les deux vecteurs.
```
[$angleX:=\prodscal(\normalize(%1),\normalize(%2)),
if angleX>=1 then 0 elif angleX<=-1 then pi else arccos(angleX) fi]
```

- **proj3d(<liste point3D>,<plan>)** : calcule la liste des projetés orthogonaux des points de *liste point3D* sur le *plan*. Le *plan* est une liste de la forme : [point3D, vecteur normal].
```
[$A:= Copy(%2,1,2), $u:=\normalize(Copy(%2,3,2)), $k:=0,
for $z in %1 do
 if z=jump then jump, k:=0, $M:=1/0
  else
     Inc(k,1), Insert(M,z),
     if k=2 then $x:=M-A, M-\prodscal(u,x)*u, k:=0, M:=1/0 fi
 fi
od]
```

- **sym3d(<liste point3D>,<plan>)** : calcule la liste des symétriques orthogonaux des points de *liste point3D* par rapport au *plan*. Le *plan* est une liste de la forme : [point3D, vecteur normal].
```
[$A:= Copy(%2,1,2), $u:=\normalize(Copy(%2,3,2)), $k:=0,
for $z in %1 do
 if z=jump then jump, k:=0, $M:=1/0
  else
     Inc(k,1), Insert(M,z),
     if k=2 then $x:=M-A, M-2*\prodscal(u,x)*u, k:=0, M:=1/0 fi
 fi
od]
```

- **interPP(<plan1>,<plan2>)** : intersection plan-plan. Chaque plan est de la forme : [point3D, vecteur normal] et la macro renvoie une droite sous la forme d'une liste du type [point3D, vecteur directeur].
```
[$A:=Copy(%1,1,2), $u:=Copy(%1,3,2), $B:=Copy(%2,1,2), $v:=Copy(%2,3,2),
$N:=\prodvec(u,v), $M:= A+ \prodscal(B-A,v)/\prodscal(N,N)*\prodvec(N,u),
M, N]
```

- **interDP(<droite>,<plan>)** : intersection droite-plan. La droite est de la forme : [point3D, vecteur directeur] et le plan de la forme [point3D, vecteur normal], la macro renvoie une droite un point3D.
```
[$A:=Copy(%1,1,2), $u:=Copy(%1,3,2), $B:=Copy(%2,1,2), $v:=Copy(%2,3,2),
 A+ \prodscal(A-B,v)/\prodscal(u,v)*u]
```

5) Les macros graphiques de TeXgraph.mac pour la 3D

i) Arc3D(<B>, <A>, <C>, <rayon>, <sens> [,<vecteur normal>])

Dessine un arc de cercle dans le plan (ABC) allant de [A,B) vers [A,C) dans le sens direct si sens>0. Lorsque A, B, C sont alignés on peut préciser un vecteur normal au plan souhaité. Les arguments B,A,C et vecteur normal sont des points3D.

```
[Set($n1,%1-%2), Set($n2,%3-%2),
 Set($vecu,%4*n1/norm(n1)),
 if nil(%6) then Set($vecn,\prodvec(n1,n2)) else vecn:=%6 fi,
 $ecart:=\angle(n1,n2),
 Set(vecn,vecn/norm(vecn)),
 Set($vecv,\prodvec(vecn,vecu)),
 Set($tfin, Si(%5>0,ecart,ecart-2*pi)),
 Set($Nbp,1+Ent(abs(tfin)*NbPoints/(2*pi))), Set($tpas,tfin/(Nbp-1)), $vart:=0,
 Ligne(Proj3D(
  for $ka from 1 to Nbp do
   %2+cos(vart)*vecu+sin(vart)*vecv, Inc(vart,tpas)
 od),0)
]
```

ii) Axes3D(<Ox>, <Oy>, <Oz> [, <gradx>, <grady>, <gradz>, <hauteur taquets>])

Trace les axes de repère spatial, on donne les coordonnées de l'origine et le pas des graduations sur les axes (0= aucune graduation, c'est la valeur par défaut), et la hauteur des graduations (xyticks par défaut).

```
[Set($Ox,%1), Set($Oy,%2), Set($Oz,%3),Set($gradx,%4), Set($grady,%5), Set($gradz,%6),
 Set($taquet,%7/2), Set($oldarrows, Arrows),
 Si(nil(gradx), Set(gradx,0)), Si(nil(grady), Set(grady,0)), Si(nil(gradz), Set(gradz,0)),
 Si(nil(taquet), taquet:=xyticks/Xscale/2),
 Ligne(Proj3D([[Xinf+i*Oy,Oz],[Xsup+i*Oy,Oz],jump,
 [Ox+i*Yinf,Oz],[Ox+i*Ysup,Oz],jump,
 [Ox+i*Oy,Zinf],[Ox+i*Oy,Zsup]]),0),
 Set($LabelDep,0.5/Xscale),
 Label(Proj3D([Xsup+LabelDep+i*Oy,Oz]),"x"),
 Label(Proj3D([Ox+i*(Ysup+LabelDep),Oz]),"y"),
 Label(Proj3D([Ox+i*Oy, Zsup+LabelDep]),"z"),
 Set(Arrows,0),
 if gradx>0 then
  Ligne( Proj3D(
    for $k from Xinf to Xsup step gradx do
       [k+i*Oy,Oz+taquet],[k+i*Oy,Oz-taquet], jump od),0),
  for $k from Xinf to Xsup step gradx do
      Label(Proj3D([k+i*(Oy),Oz])-0.3*i/Yscale, k) od
 fi,
 if grady>0 then
  Ligne( Proj3D(
    for $k from Yinf to Ysup step gradx do
      [Ox+i*k,Oz+taquet],[Ox+i*k,Oz-taquet], jump od),0),
  for $k from Yinf to Ysup step grady do
    Label(Proj3D([Ox+i*k,Oz])-0.3*i/Yscale, k) od
```

```
 fi,
 if gradz>0 then
  Ligne( Proj3D(
   for $k from Zinf to Zsup step gradz do
    [Ox+i*(Oy+taquet),k],[Ox+i*(Oy-taquet),k], jump od),0),
  for $k from Zinf to Zsup step gradz do
    Label(Proj3D([Ox+i*Oy,k])-0.3/Xscale, k) od
 fi,
 Set(Arrows, oldarrows)
]
```

iii) Cercle3D(<centre>,<rayon>,<vecteur normal>)

Dessine un cercle dans l'espace, *centre* est donc un point de l'espace (`[x+i*y,z]`), le *vecteur normal* est normal au plan du cercle et non nul (de la forme `[a+i*b,c]`).

```
[ Set($n, \normalize(%3)),
  Set($A, %1), Set($R, %2), Set($u,\prodvec(\n,n)),
  Si( norm(u)=0, Cercle(Proj3D(A),R),
     [ {sinon}
       Set($u, R*\normalize(u)),  Set($v, \prodvec(n, u)),
       Set(tMin, -pi), Set(tMax, pi),
       Ligne(Get(Courbe(Proj3D( A+cos(t)*u+sin(t)*v),0)),1)
     ])
]
```

iv) Courbe3D(<x(t)>,<y(t)>,<z(t)> [, divisions [, discontinuités]])

Dessine une courbe gauche. On peut indiquer le nombre divisions par 2 autorisé entre 2 points consécutifs, et la prise en compte des discontinuités (0 ou 1).

```
 Courbe(Proj3D([%1+i*%2,%3]),%4,%5)
```

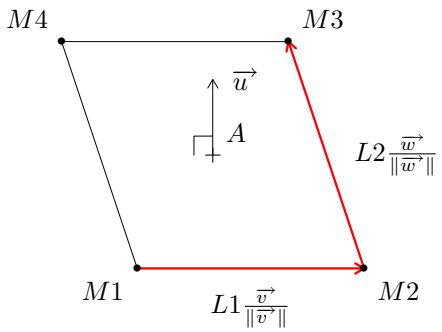
v) DrawDroite(<droite> [,<longueur gauche>,<longueur droite>])

Trace une droite de l'espace, celle-ci est de la forme [point3D, vecteur 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 $A - L1\frac{u}{\|u\|}$ à $A + L2\frac{u}{\|u\|}$ qui est dessiné.

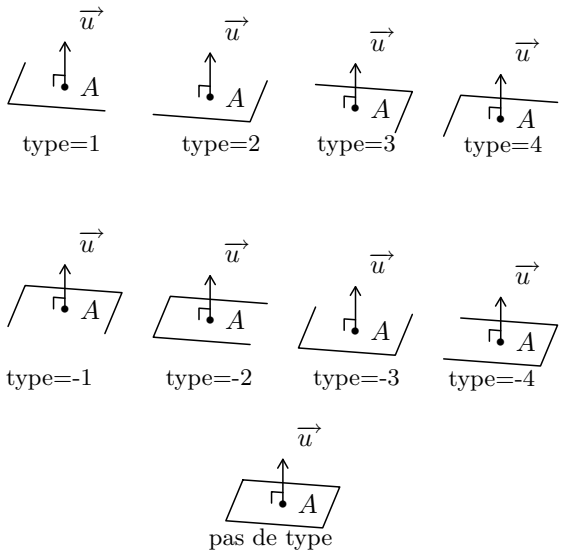
```
[$A:=Copy(%1,1,2), $u:=Copy(%1,3,2),
 if nil(%2) then Droite(Proj3D(A), Proj3D(A+u))
 else u:=\normalize(u), Ligne( Proj3D([A-%2*u, A+%3*u]),0)
 fi]
```

vi) DrawPlan(<plan>,<vecteur>,<longueur1>,<longueur2> [,<type>])

Permet de représenter un plan de l'espace, le paramètre *plan* est de la forme [point3D, vecteur normal], notons A le point et u le vecteur normal, le paramètre suivant est un vecteur du plan (notons le v), la macro calcule le vecteur $w = u \wedge v$ et détermine le parallélogramme suivant :



où $L1$ est le paramètre *longueur1* et $L2$ le paramètre *longueur2*. Si le dernier paramètre *type* est absent, alors c'est le parallèlogramme qui est dessiné, sinon voici les différentes valeurs possibles (le point A, le vecteur u et l'angle droit ont été ajoutés) :



vii) Ligne3D(<liste de point3D>, <fermée>)

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).

```
Ligne( Proj3D(%1), %2)
```

6) Les macros de polyedres.mac

Le fichier polyedres.mac est automatiquement chargé par le fichier TeXgraph.mac, celui-ci doit être présent dans le dossier contenant le programme. Ce fichier contient quelques macros pour faire des polyèdres élémentaires ainsi que leur section par un plan. Ce fichier contient une seule variable qui est :

HideStyle : initialisée à dotted, pour le style de tracé des arêtes cachées.

Un polyedre est représenté par une liste de facettes séparées par la constante *jump*, une facette est une succession de point3D représentant les sommets de la facette, ceux-ci doivent être écrits dans un certain ordre de telle sorte que la facette soit orientée par la normale extérieure au polyèdre.

Il y a trois mode de représentation :

- mode 0 : le dessin se fait arête par arête, y compris les arêtes cachées (qui seront dessinées dans le style HideStyle).
- mode 1 : le dessin se fait par face visible, celles-ci peuvent être remplies en fonction de l'attribut FillStyle.
- mode 2 : le dessin est fait comme dans le mode 1, puis on rajoute les arêtes cachées.

Macros permettant de dessiner :

i) DrawPoly(<polyedre>, <mode>)

Elle permet de dessiner le polyèdre dans le mode voulu.

ii) DrawAretes(<aretes>,<mode>)

Elle permet de dessiner une liste d'arêtes dans le mode voulu. Une liste d'arêtes est une succession de segments définis par deux points3D, deux arêtes sont séparées par la constante *jump*, cette constante contient aussi une information sur l'arête dans sa partie imaginaire (0= arête cachée, 1= arête visible), la liste se termine donc par *jump*.

iii) Dcone(<point3D>,<vecteur3D>,<rayon>,<mode>)

Dessine un cône à partir de son sommet, d'un vecteur de l'axe qui indique la direction et la hauteur du cône, et du rayon r de la face circulaire.

iv) Dcylindre(<point3D>,<vecteur3D>,<rayon>,<mode>)

Dessine un cylindre à partir d'un point qui est le centre d'une des deux faces circulaires, d'un vecteur de l'axe qui indique la direction et la hauteur du cylindre, et d'un rayon r.

v) Dsphere(<point3D>,<rayon>,<mode>)

Dessine une sphère à partir de son centre et de son rayon r.

Macros de construction de polyèdres

vi) MakePoly(<sommets3D>,<liste faces avec n° des sommets>)

Cette macro prend en entrée la liste des point3D représentant les sommets d'un polyèdre, et la liste des facettes mais avec les numéros des sommets à la place des coordonnées, elle renvoie la liste des facettes avec les coordonnées des sommets à la place des numéros (ce qui donne un polyèdre). Par exemple, pour une pyramide de base A(1,0,0),B(1,1,0),C(0,1,0),D(0,0,0) et de sommet E(1,1,1) (dans un élément graphique utilisateur) :

[P :=MakePoly([[1,0],[1+i,0],[i,0],[0,0],[1+i,1]],[1,4,3,2, jump,1,2,5, jump,2,3,5, jump,3,4,5, jump,4,1,5, jump]), DrawPoly(P,0)]

Le mieux est sans doute de créer une macro qui renvoie le polyèdre en sortie.

vii) Parallelep(<sommet>,<vecteur1>,<vecteur2>,<vecteur3>)

Cette macro construit et renvoie la liste des facettes d'un parallélépipède à partir d'un sommet et de 3 vecteurs, supposés dans le sens direct.

viii) Prisme(<base>,<vecteur>)

Cette macro construit et renvoie la liste des facettes d'un prisme à partir d'une base et d'un vecteur qui représente le vecteur de translation de la base à la face opposée. La base est une liste de point3D coplanaires, cette doit être dans le sens direct, le plan étant orienté par le vecteur de translation.

ix) Pyramide(<base>,<sommet>)

Cette macro construit et renvoie la liste des facettes d'une pyramide à partir de sa base et du sommet. La base est une liste de point3D coplanaires, cette doit être dans le sens direct, le plan étant orienté par le sommet.

x) Tetra(<sommet>,<vecteur1>,<vecteur2>,<vecteur3>)

Cette macro construit et renvoie la liste des facettes d'un tétraèdre à partir d'un point et 3 vecteurs, supposés dans le sens direct.

xi) Section(<plan>,<polyedre>)

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.

Exemple d'utilisation (dans un élément graphique utilisateur) :

```
[cube:= Parallelep(Origin, 3*vecI, 3*vecJ, 3*vecK),
 plan:= [dot(3,0,0), -vecI-vecK],
 S:=Section(plan, cube),
 DrawPoly(S,0)]
```

xii) Intersection(<plan>,<polyedre>) [,<sortie facette>])

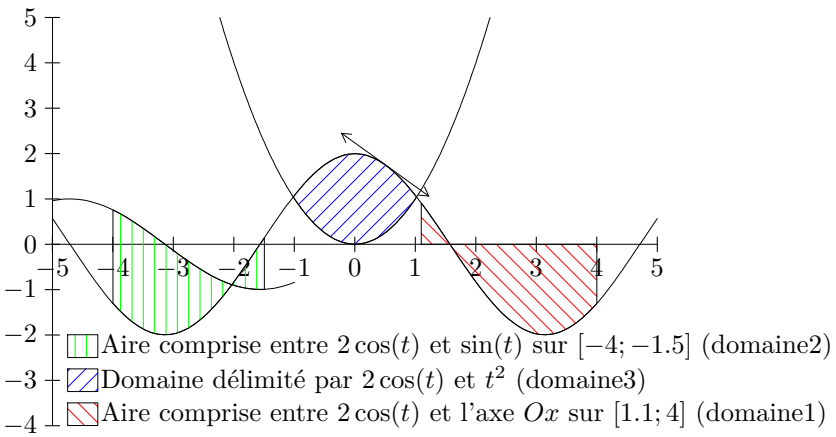
Le plan doit être de la forme : [S,u], cela représente le plan passant par le point S et normal au vecteur u. La macro détermine l'intersection du polyèdre avec ce plan et renvoie celle-ci sous forme d'une **liste d'arêtes** (que l'on peut dessiner avec la macro DrawAretes). Il est possible de récupérer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

Formellement, les deux dernières macros ne s'appliquent pas aux : cylindres, cônes, et sphères. Pour rendre cela possible il y a trois macros qui définissent les cylindres, cônes et sphères sous forme de polyèdres :

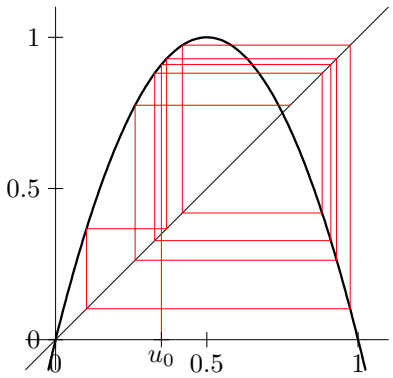
- **Cylindre(<point3D>,<vecteur3D>,<rayon>,<nombre faces>)**
  Cette macro renvoie un polyèdre représentant le cylindre construit à partir d'un point qui est le centre d'une des deux faces circulaires, d'un vecteur de l'axe qui indique la direction et la hauteur du cylindre, d'un rayon r et du nombre de faces. Attention : les deux faces circulaires font parties des facettes.
- **Cone(<point3D>,<vecteur3D>,<rayon>,<nombre faces>)**
  Cette macro renvoie un polyèdre représentant le cone construit à partir de son sommet, d'un vecteur de l'axe qui indique la direction et la hauteur du cône, du rayon r de la face circulaire et du nombre de faces. Attention : les deux faces circulaires font parties des facettes.
- **Sphere(<centre>,<rayon>,<nb fuseaux>,<nb tranches>)**
  Cette macro renvoie un polyèdre représentant la sphère construite à partir de son centre et de son rayon. Les deux autres paramètres déterminent le nombre de faces.

Ces macros ne sont pas prévues pour dessiner cylindres, cônes et sphères, outre l'aspect esthétique, elles sont trop lentes, surtout en mode 0 (dessin par arête).

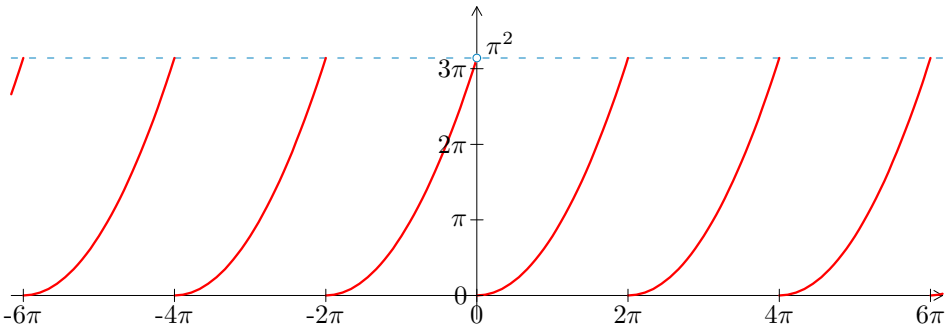
IV) Exemples



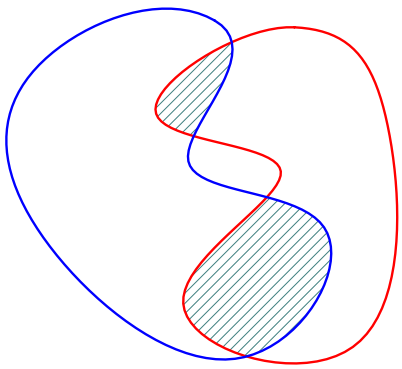
Utilisation des macros `domaine1,2,3` et `tangente`



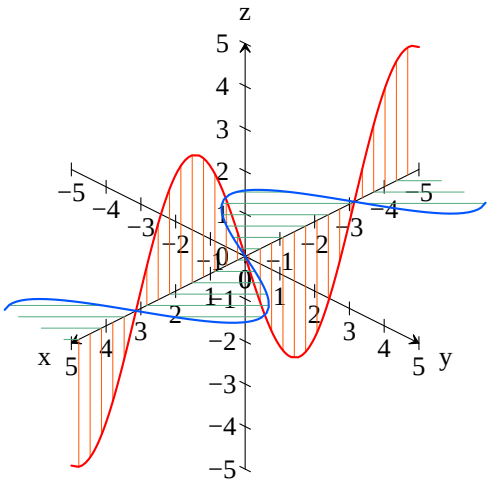
Suite $u_{n+1} = 4u_n(1 - u_n)$ avec $u_0 = 0.35$ et $n = 10$ (macro `suite`)



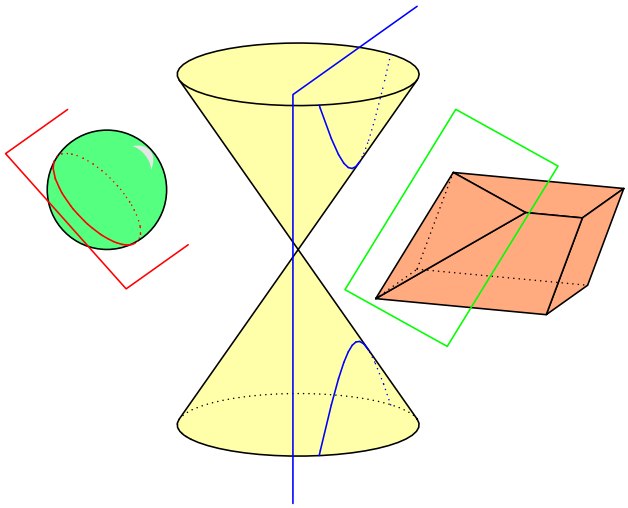
Fonction périodique définie par $f(t) = \frac{t^2}{4}$ sur $[0; 2\pi]$ (macro `periodic`)



Utilisation de la macro `Clip` pour représenter une intersection.



Utilisation de la fonction `Proj3D` pour travailler dans l'espace.



Utilisation des macros de polyedres.mac.

Pour ce dernier exemple, voici les commandes définissant les trois solides (éléments graphiques utilisateur)

Pour la sphère :

```
[$A:=[1-3*i,1], $S:=Sphere(A,1,25,25),
 $B:=A-0.25*(vecJ+vecK)/sqrt(2),
 $L:=Intersection([B, vecK+vecJ], S),
 Dsphere(A,1,1),
 HideStyle:=dotted, FillStyle:=none, Color:=red,
 DrawAretes(L,0),
 DrawPlan( [ B, vecK+vecJ], vecJ-vecK, 3,3,-3)]
```

Pour le cône :

```
[Dcone( Origin, -3*vecK, 2, 2),
 Dcone( Origin, 3*vecK, 2, 2),
 $C:=Cone(Origin, -3*vecK, 2, 35), FillStyle:=none,
 $L:=Intersection( [[i,0], -vecJ], C,$S),
 Del(L,1,5), Color:=blue,
 DrawAretes(L,0),
 $C:=Cone(Origin, 3*vecK, 2, 35),
 $L:=Intersection( [[i,0], -vecJ], C,$S),
```

```
 Del(L,1,5), Color:=blue,
 DrawAretes(L,0),
 DrawPlan( [ [i,0], -vecJ], vecK, 7, 6,2)]
```

Pour le parallèlpipède :

```
[$A:=[-3+1*i,-1], $B:=A+2*vecI,
$C:=A+vecI+vecJ+2*vecK, $D:=C+2*vecI+2*vecJ,
$S:=Parallelep(A, 2*vecI, 3*vecJ, vecI+vecJ+2*vecK),
$u:=-prodvec(D-B,C-B),
$S':=Section( [B, u], S),
DrawPoly(S',2), FillStyle:=none, Color:=green,
DrawPlan( [(B+C+D)/3,u], D-C, 4,3.5,5)]
```

Chapitre 4

Les Macros

I) Qu'est ce qu'une macro ?

1) Généralités

Une macro est une fonction créée par l'utilisateur. 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 `*.teg`.
- celles qui sont chargées par le menu avec l'option `Fichier/ Charger des macros`, ou par l'instruction `InputMac` : celles-ci sont considérées comme **prédéfinies** et n'apparaissent pas dans la liste des macros modifiables, elles ne sont pas enregistrées dans les fichiers `*.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 `*.teg`.

Un fichier de macros est tout simplement un fichier `*.teg`[1] 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.

2) 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*[2]*, cela ne peut que faciliter la lisibilité. De plus il est possible de documenter une commande en insérant des commentaires entre deux accolades : { <commentaire> }.*

Voici par exemple la macro `racine(n,z)` qui donne la liste des racines n-ièmes de z :
Nom : `racine`, Commande :

```
{racine(n,z): donne les racines nièmes de z}
Si(Ent(%1)=%1 And %1>0,
        [Set($a, abs(%2)^(1/%1)),
         Seq(a*exp(i*(Arg(%2)+$k*2*pi)/%1), k, 0, %1-1)]
   )
```

Commentaire : on teste si n (*i.e.* `%1` le premier paramètre) est un entier strictement positif, auquel cas on stocke dans une variable locale a la valeur de $|z|^{\frac{1}{n}}$ (*i.e.* `abs(%2)^(1/%1)`) puis on donne la liste des solutions (sinon la macro renvoie **Nil**).

Exemple(s) : l'exécution de `[Set($a,3), racine(a,i)]` donne :

1. L'extension `teg` n'est pas obligatoire, on peut très bien en prendre une autre comme `mac` par exemple.
2. Ceci est également valable pour la commande des éléments graphiques `Utilisateurs`.

`[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** (ex : `racine(3)` donnera **Nil**), s'il y en a trop, ceux qui sont en surplus sont ignorés.

II) Développement d'une macro

1) 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[3]. C'est lors de cette analyse qu'une macro peut être développée tout de suite ou non.

Lors de **l'analyse** de `[Set($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 `[Set($a,3), \racine(a,i)]` : TEXgraph remplace `\racine` par l'expression de la macro en remplaçant le paramètre %1 par la variable a et le paramètre %2 par i, ce qui revient à analyser la commande :
`[Set($a,3), Si(Ent(a) Egal a And a Sup 0, [Set($a, abs(i)^(1/a)), Seq(a*exp(i*(Arg(i)+$k*2*pi)/%1), k, 0, a-1)])]` : **c'est le développement immédiat**. On remarquera que cette fois-ci il y a une seule variable a, ce qui fait que cette commande donne comme résultat : `i`. Par contre `[Set($b,3), \racine(b,i)]` donne le bon résultat.

On l'aura compris, le développement immédiat est à proscrire lorsque la macro possède des variables locales et qu'il y a un risque d'ambiguïté[6]. 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)`.

2) La récursivité

Comme TEXgraph ne génère pas d'erreur lorsqu'il rencontre un nom de macro qui n'existe pas, et grâce au développement différé, les macros peuvent être récursives. Cependant il faut savoir qu'une macro récursive qui ne se termine pas va « planter » le programme (saturation de la mémoire).

Donnons quelques exemples :

- La sempiternelle fonction factorielle : Nom : `fac`, Commande : `Si(%1=0,1,%1*fac(%1-1))`.
  Attention, c'est la version minimaliste, il faudrait tester si %1 est bien un entier positif... L'exécution de `fac(-2)` par exemple plantera le programme.
- La fonction dichotomie(f, variable, a, b, epsilon) : Nom : `dichotomie`, Commande :
  `[Set($c,(%4+%3)/2), Si(abs(%4-%3) InfOuE %5, $c, [Set(%2,c), Set($fc,%1), Set(%2,%3), Set($fa,%1), Si(fa*fc Sup 0, dichotomie(%1,%2,c,%4,%5), dichotomie(%1,%2,%3,c,%5))])]`
- Voici un exemple graphique, on part d'une fonction affine f_0 sur un segment, on partage ce segment en trois parties égales et on construit la fonction f_1 comme l'indique la figure Fig 4.1.
  on recommence le processus sur chacune des trois parties pour construire la courbe de f_2 etc... Voici la macro `trace(a,b,n)` qui effectue (voir Fig 4.2) la construction de la courbe de f_n[7], a et b sont les affixes du segment initial :

3. Cette chaîne est transformée en arbre.
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 ».
6. On peut toujours chercher des noms « tordus » pour les variables locales de macros mais cela ne fait que diminuer le risque.
7. On peut montrer que la suite (f_n) converge simplement vers une fonction continue sur le segment initial, mais dérivable nulle part.

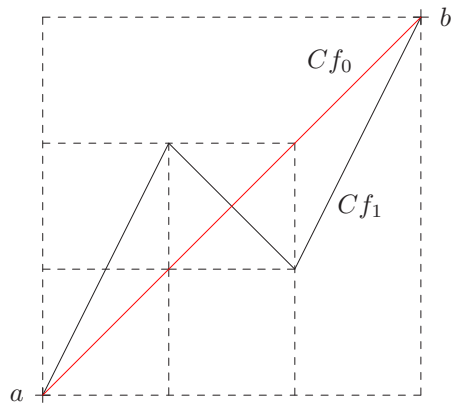


FIG. 4.1 – Construction de f_1

```
Si(%3 Egal 0, Seg(%1, %2),
[Set($a,%1), Set($b,%2), Set($n,%3), (Set($c, (2*a+b)/3), Set($d, (a+2*b)/3),
 trace(a,Re(c)+i*Im(d),n-1), trace(Re(c)+i*Im(d),Re(d)+i*Im(c),n-1),
 trace(Re(d)+i*Im(c),b,n-1)])
```

`trace(0,5*(1+i),5)`

(Commande dans un élément graphique Utilisateur)

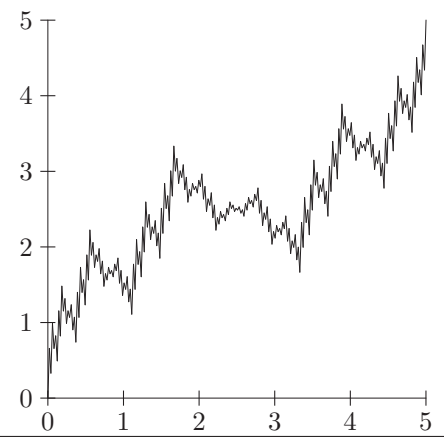


FIG. 4.2 – La fonction f_5

Ces trois exemples sont faciles à « dérécursifier », par exemple la macro `trace` dérécursifiée peut s'écrire :

```
Ligne(
[Set($L,[%1,%2]),
 Seq(Set(L, [free($u),
             Seq([Set($a,Copy(L,$t,1)), Set($b,Copy(L,t+1,1)),
                  Set($c,(2*a+b)/3), Set($d,(a+2*b)/3),
                  Set($u,[u,a,Re(c)+i*Im(d),Re(d)+i*Im(c)])
                  ],t,1,Nops(L)-1),
              [u,b]
            ]),k,1,%3), L], 0)
```

Dans cette version, la macro génère la liste (`L`) des points à relier et trace la ligne polygonale, on peut alors créer un élément graphique `utilisateur` avec la commande `trace(0,5*(1+i),5)`.

Donnons un dernier exemple, les L-systèmes : voir FIG 4.3.

Les symboles adoptés sont :

```
Ligne(
[Set(dir,0),Set(Pc,-5*i),
Lsystem([3,3,3,3,1],4)],0)
```

(Commande dans un élément graphique Utilisateur)

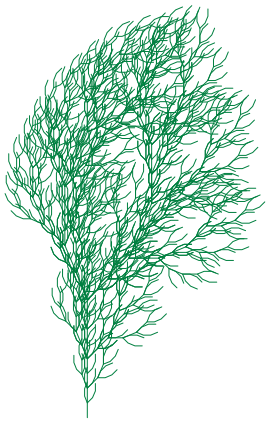


FIG. 4.3 – Exemple de L-système.

- 1 : avance en traçant,
- 0 : avance sans tracer,
- 2 : tourne d'une valeur donnée (angle) dans le sens négatif,
- 3 : tourne de la valeur angle dans le sens positif,
- 4 : mémorise le point courant et la direction,
- 5 : revient au dernier point mémorisé (avec sa direction).

Le principe est le suivant : on a un axiome de départ, dans l'exemple ci-dessus il s'agit de `[3,3,3,3,1]`, et une règle qui régit l'évolution, dans l'exemple la règle est : `1` est remplacé par :

`[1,1,2,4,2,1,3,1,3,1,5,3,4,3,1,2,1,2,1,5]`

le niveau est le nombre de fois où la règle est appliquée [au niveau 0 la règle n'est pas appliquée]. Pour l'exemple précédent nous avons défini les variables globales :

- `regle=[1, 1, 2, 4, 2, 1, 3, 1, 3, 1, 5, 3, 4, 3, 1, 2, 1, 2, 1, 5]`,
- `angle=pi/8`,
- `Pc=-5*i` : le point courant,
- `long=0.15` : longueur d'un déplacement,
- `dir=0` : direction du point courant,
- `niveau=4`,
- `memo=1/0` : liste (vide) pour mémoriser les points.

La macro `Lsystem(axiome,niveau)` :

```
{Lsystem(axiome, niveau): applique la règle au niveau requis récursivement sur
l'axiome de départ.}
[Si(%2=0,Set($L,Pc)),
 Map(Si($u=1, Si(%2=0, [Set(Pc, Pc+long*exp(i*dir)), Append(L, Pc)],
                       Append(L,Lsystem(regle, %2-1))),
          u=0, [Set(Pc, Pc+long*exp(i*dir)), Append(L, jump)],
          u=2, Set(dir, dir-angle),
          u=3, Set(dir, dir+angle),
          u=4, Append(memo, [Pc, dir],1),
          u=5, [Set(Pc, Copy(memo,1,1)), Set(dir, Copy(memo,2,1)),
                     Del(memo,1,2), Append(L,jump), Si(%2=0,Append(L,Pc))]
        ), u, %1),L,
$rec\recoit rectangle(L),
Fenetre(rec[1], rec[2])
]
```

On construit la liste (`L`) des points à relier ; dès qu'il y a un saut (`u=0` ou `u=5`) on ajoute `jump`, puis le point courant si on est au niveau 0. Avant de terminer, la macro le plus petit restangle contenant la ligne polygonale et ajuste la fenêtre sur lceui-ci. On peut créer ensuite un élément graphique `utilisateur` (voir Fig 4.3).

Voici un autre exemple de L-systèmes : `règle= [1,3,1,2,2,1,3,1]`, `angle=pi/3`, `niveau=3` : voir Fig 4.4.

```
Ligne(
[Set(dir,0), Set(Pc,-6-4*i),
Lsystem([3,1,2,2,1,2,2,1],niveau)],0)
```

(Commande dans un élément graphique Utilisateur)

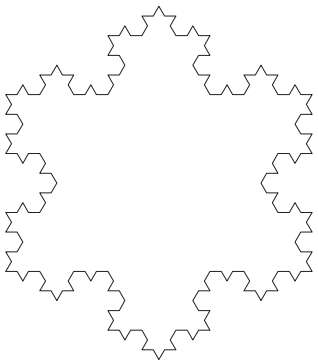


Fig. 4.4 – Le flocon de Koch.

Chapitre 5

Les fonctions et macros spéciales

I) La macro Init()

1) Utilisation

Si un fichier `*.teg` contient une macro intitulée `Init` alors TEXgraph exécutera automatiquement cette macro dès la fin du chargement du fichier. Cette macro peut être utilisée pour faire certaines initialisations ou pour afficher un menu ou pour demander à l'utilisateur des valeurs. Ceci est utile surtout lorsque le fichier est destiné à servir de **modèle**[1] pour certains types de graphiques susceptibles de revenir souvent[2].

2) Exemple

Voici le texte de la macro `Init()` du modèle `Lsystem.mod` :

```
[Si(Input("Règle="), Set(regle,Eval(chaine()))),
 Si(Input("Axiome="), Set(axiome,Eval(chaine()))),
 Si(Input("Angle="), Set(angle, Eval(chaine()))),
 Si(Input("Direction départ="), Set($dirDep, Eval(chaine())), Set(dirDep,0) ),
 Si(Input("Niveau="), Set(niveau, Eval(chaine())), Set(niveau,1)),
 Creer("Arbre",
["[free(memo),Set(Pc,0), Set(dir,",dirDep,"),
  Ligne([Set($L,Lsystem(axiome,niveau)), Set($rec,rectangle(L)),
          Fenetre(Copy(rec,1,1),Copy(rec,2,1)),L], 0) ]"] )]
```

Cette macro demande à l'utilisateur la valeur de la règle, l'axiome, l'angle, la direction initiale et le niveau. Les réponses sont des chaînes stockées dans la macro `chaine()` et qui sont évaluées par la fonction spéciale `Eval`. Puis la macro crée un élément graphique `Utilisateur` appelé `Arbre` et dont la commande met à jour trois variables puis trace la ligne polygonale définie par la commande :

```
 [Set($L,Lsystem(axiome,niveau)), Set($rec,rectangle(L)),
          Fenetre(Copy(rec,1,1),Copy(rec,2,1)),L], 0)]
```

Cette commande exécute la macro `Lsystem`, stocke le résultat dans la variable locale `L`, calcule le plus petit rectangle contenant `L`, ajuste la fenêtre sur ce rectangle et renvoie la valeur de `L`.

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

La macro `Bsave` est automatiquement exécutée avant l'exportation du graphique en cours.
La macro `Esave` est automatiquement exécutée après l'exportation du graphique en cours.

1. Ce sont les fichiers `*.mod`. Un modèle se charge avec l'option `Fichier/Importer un modèle`, son contenu vient s'ajouter au graphique en cours (il est préférable en général de charger le modèle avant de commencer le graphique proprement dit). À l'issue du chargement du modèle, la macro `Init()`, si elle existe, est exécutée et le graphique est affiché dans la fenêtre. Si la macro `Init` est prédéfinie, alors elle est supprimée de la mémoire après son exécution.
2. Comme les tableaux de variation par exemple.

La constante `ExportMode` permet de connaître le mode d'exportation, sa valeur peut-être une des constantes suivantes : tex, pgf, pst, eps, 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 gobales.
- Les macros.
- Les éléments graphiques.

Juste avant la sauvegarde des variables globales, TEXgraph regarde si la macro `TegWrite` existe, 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.

Exemple(s) : si vous créez la macro `TegWrite` avec la commande `InputMac("MesMacros.mac")`, alors lors de la sauvegarde, juste avant la déclaration des variables on trouvera dans le fichier `*.teg` la ligne :

```
18##InputMac("MesMacros.mac")##
```

Cette ligne de commande provoquera lors de l'ouverture du fichier, le chargement automatique des macros du fichier MesMacros.mac (sous de forme de macros prédéfinies).

Ces trois macros n'existent pas par défaut et peuvent être créées par l'utilisateur.

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

1) Utilisation

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` permet de créer une variable globale.

Lorsque le bouton droit est relaché 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.

2) Exemple

Construire une ligne polygonale à la souris :

- On crée une variable globale `L` initialisée par exemple à `1/0` (la variable est alors 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,Nops(L),1), ReCalc(ligne)]` (cela efface le dernier point mémorisé).

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

IV) Les macros ClicGraph() et OnKey()

1) Utilisation

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 (spéciale) **NewGraph**. 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 caractère. Cette macro, qui n'existe pas par défaut, peut être créée par l'utilisateur.

V) Les fonctions spéciales

Dans les macros spéciales, dans les commandes (ou macros) associées à un bouton, ou dans les commandes exécutées dans la ligne de commandes, on peut utiliser plusieurs fonctions, dites fonctions spéciales. Utilisées dans un autre contexte, elles seront sans effet.

1) Attributs()

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.

2) DelGraph(<element1>,...<elementN>)

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**.

3) DelItem(<nom1>,...,<nomN>)

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**.

4) DelButton(<texte1>,...<texteN>)

Cette fonction permet de supprimer dans 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**.

5) Exec(<programme> [,<argument(s)>, <répertoire de travail>, <fenêtrer>])

Cette fonction permet d'exécuter un *programme* (ou un script) en précisant éventuellement des *arguments* et un *répertoire de travail*, ces trois arguments sont interprétés comme des chaînes de caractères. Le dernier argument doit valoir 0 ou 1. 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.

Exemple(s) : la macro `Apercu` contenue dans TeXgraph.mac est :

```
[Export(pgf,[\InitialPath,"file.pgf"]), Exec("apercu.bat","",\InitialPath,0)]
```

La macro prédéfinie \InitialPath contient le chemin et le répertoire du programme TeXgraph . La macro *Apercu* exporte le graphique courant dans ce dossier au format pgf dans le fichier file.pgf, puis elle exécute

le script `apercu.bat`, sans argument, le répertoire de travail étant le répertoire initial, l'exécution est non fenêtrée.

6) Export(<mode> ,<fichier>)

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, eps ou teg. L'exportation se fait dans *fichier* qui contient donc le nom du fichier, avec éventuellement le chemin.

7) Input(<message> [,titre,edit])

Cette fonction ouvre une boite de dialogue avec *titre* dans la barre de titre (par défaut le titre est vide), dans laquelle le paramètre *message* est affiché, et possédant une ligne de saisie initialisée à *edit*. Ces trois paramètres sont donc interprétés comme des chaînes de caractères. L'utilisateur est invité à faire une saisie, s'il valide et que la chaîne saisie est non vide, 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`. En association avec la fonction **Eval**, on peut faire des saisies numériques, par exemple :

```
Si(Input("x="), Set(x,Eval(chaine())))
```

8) LoadImage(<image>)

Cette fonction charge le fichier *image*, qui doit être une image jpg 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 jpg qui sera dans le document, mais pour les versions pst et tex il faut une version eps de l'image.

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.

L'argument est interprêté comme une chaine de caractères, et la fonction renvoie la valeur **Nil**.

9) Move(<element1>,...,<elementN>)

Cette fonction ne s'applique qu'aux éléments graphiques créés en mode NotXor, ce qui correspond à la valeur 1 de la variable PenMode. 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 rubrique n°18) (fonction Stroke).

10) NewButton(<Id>,<nom>,<affixe>,<taille>,<commande> [,aide])

Cette fonction crée à gauche de la zone de dessin, 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, 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 $\text{long} + i * \text{haut}$ 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, c'est la commande associée au bouton, chaque clic provoquera l'exécution de cette commande. Le paramètre facultatif *aide* est interprété comme une chaîne et représente la bulle d'aide s'affichant lorsque la souris passera 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**.

11) NewGraph(<chaine1>,<chaine2> [,<code>])

Cette fonction crée un élément graphique `utilisateur` ayant pour nom : *chaine1* et défini par la commande : *chaine2*. 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. Cette fonction s'appelle aussi **Creer**. 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 *Mouse1.mod* (dessin à la souris).

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 :

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

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)`.

12) NewItem(<nom>,<commande>)

Cette fonction ajoute dans la liste déroulante à gauche de la zone de dessin, 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**.

13) NewMac(<nom>,<corps>)

Cette fonction crée une macro appelée *nom* et dont le contenu est *corps*. Les deux arguments sont donc interprétés comme des chaînes de caractères. 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. Cette fonction s'appelle aussi **DefMac**.

14) NewVar(<nom>,<expression>)

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 celui-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**. Cette fonction s'appelle aussi **DefVar**.

15) ReCalc(<nom1>,$\cdots$,<nomN>)

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 un argument n'est pas le nom d'un élément graphique, alors cet argment est interprété comme une chaîne de caractères. Si cette liste est vide (`ReCalc()`) alors tous les éléments graphiques sont recalculés. Après le recalcul l'affichage est mis à jour et la fonction renvoie **Nil**.

Exemple(s) : `ReCalc(toto, ["a",1])` recalcule les éléments graphiques appelés `toto` et `a1`.

16) ReDraw()

Cette fonction provoque l'affichage de tous les éléments graphiques et renvoie la valeur **Nil**.

17) RenMac(<nom>,<new name>)

Cette fonction renomme la macro appelée *nom*. Les deux arguments sont donc interprétés comme des chaînes de caractères. Cette fonction renvoie la valeur **Nil**. S'il existait déjà une macro portant le nom *new name*, alors celle-ci est écrasée **si elle n'est pas prédéfinie**. Si le nom n'est pas valide, ou s'il n'y a pas de macro portant ce nom, ou s'il y a une macro prédéfinie appelée *new name*, alors la fonction est sans effet.

18) Stroke(<element1>,...,<elementN>)

Cette fonction, elle recalcule les éléments graphiques *element1*, ..., *elementN*, puis les redessine 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]
```

puis 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 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".

VI) Macros spéciales de TeXgraph.mac

1) NewLabel(%1)

Création de labels à la souris, cette macro est destinée à être utilisée dans la macro **ClicG()**.

```
if Input("Texte du label=","Création d'un Label",chaine()) then
 Inc(ComptGraph,1),
 Creer(["label",ComptGraph], ["Label(",%1,",""",chaine(),""")"]),
 ReDraw()
fi
```

2) Bouton(<position>,<nom>,<macro> [,aide])

Création d'un bouton, position=x+i*y avec x y en pixels, le nom, la macro associée et l'aide sont interprétés comme des chaînes de caractères.

```
[Eval(["NewButton(",NbBoutons,",""",%2,""",",
                    %1,",",tailleB,",""",%3,""","""",%4,""")"]),
Inc(NbBoutons,1)]
```

3) chaine()

Cette macro est un moyen détourné pour mémoriser une chaîne de caractères. La commande par défaut est `"toto"`.

VII) Le modèle variatio2.mod

Nous présentons ce fichier à titre d'exemple. Il peut servir de modèle pour « dessiner » des tableaux de variation (voir Fig. 5.1).

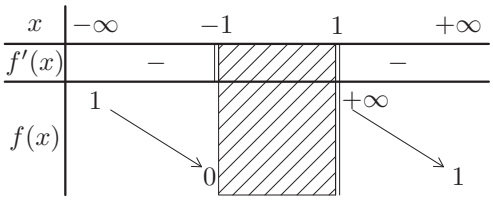


Fig. 5.1 – La fonction $f(x) = \sqrt{\dfrac{x+1}{x-1}}$

L'idée est de visualiser le tableau de variation sous la forme d'un tableau à la manière d'un tableur, avec des lignes et des colonnes indexées. Après le chargement du modèle, la macro `Init()` demande le nombre de lignes et le nombre de colonnes puis affiche un « menu » sous forme de boutons, et une grille de repérage (voir Fig. 5.2).

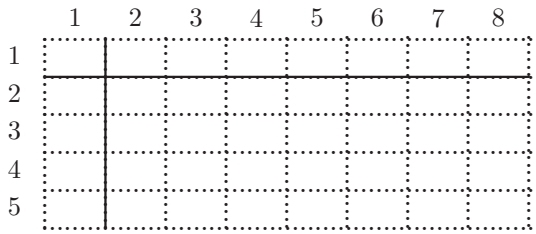


Fig. 5.2 – Après chargement du modèle.

L'utilisateur peut remplir les cases de ce tableau qui contiennent du texte (c'est à dire des labels) en cliquant dessus. Les éléments graphiques qui sont créés (par la macro `ClicG()`) portent le nom de la case (ex : `L1C1`) ce qui permet de les repérer facilement. S'il y avait déjà un label dans la case cliquée, alors l'ancien est effacé (voir Fig. 5.3).

Par l'intermédiaire des boutons, l'utilisateur peut également ajouter des flèches, des doubles barres, des zones hachurées, des cadres... en cliquant sur l'option souhaitée. Chaque élément graphique créé porte le nom de l'option suivi du numéro d'ordre d'apparition (ex : fleche1, fleche2,...), voir Fig. 5.4.

Dans l'exemple de la Fig. 5.4, on voit qu'il faut retoucher les labels `L5C4` et `L3C6` car par défaut ils sont centrés. Pour le premier on a modifié l'attribut `LabelStyle` en cochant l'option `right`, pour le second on a coché l'option `left` et on a ajouté un espace fin (`\,`) avant le signe + (sinon il touche la barre), voir Fig. 5.5.

La dernier bouton `Terminer`, enlève la grille puis ajuste la fenêtre ce qui donne la Fig. 5.6.

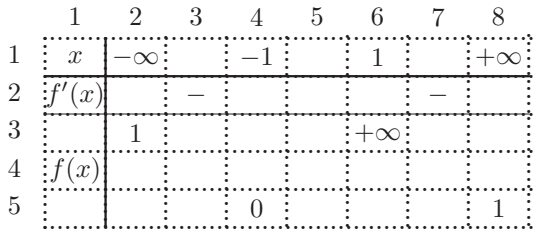


FIG. 5.3 – Remplissage des labels.

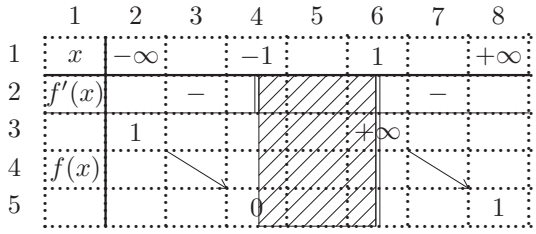


FIG. 5.4 – Ajout des flèches...

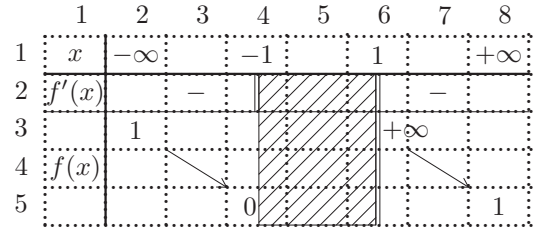


FIG. 5.5 – Ajustement des labels `L5C4` et `L3C6`.

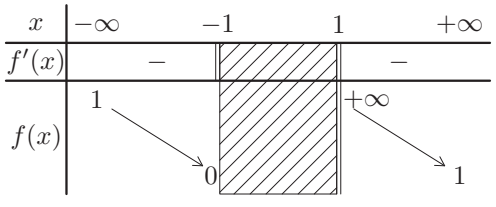


FIG. 5.6 – Après l'option Terminer

Index