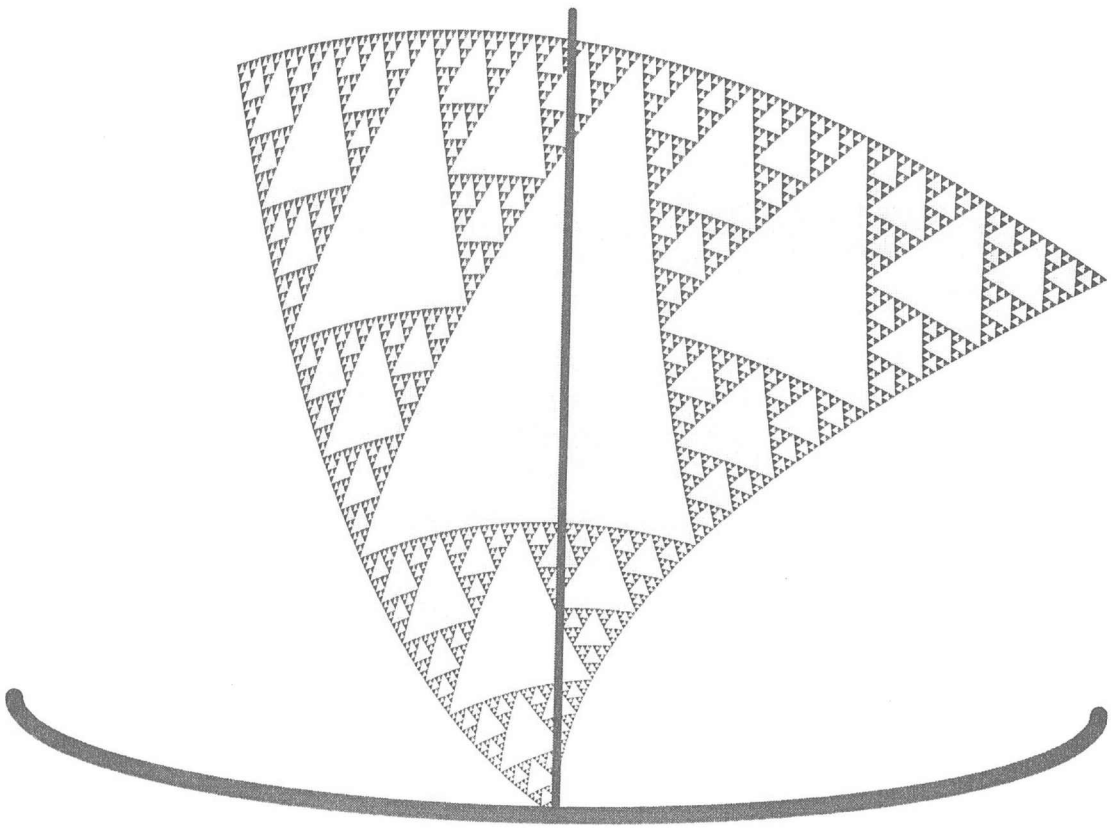
INSTITUT DE RECHERCHE SUR L'ENSEIGNEMENT DES MATHÉMATIQUES

Université Montpellier II

Place Eugène Bataillon
cc 040
34095 MONTPELLIER Cedex 05
Tél : 04.67.14.33.83/84
Fax : 04.67.14.39.09
e.mail : irem@math.univ-montp2.fr

Réaliser des graphiques et faire de la géométrie avec *Mathematica*



par
Yvon POITEVINEAU

IREM de MONTPELLIER
- 1998 -

Table des matières

Préface

Les logiciels de calcul formel sont devenus des auxiliaires précieux pour quiconque s'occupe de mathématiques :
– Ils permettent de se débarrasser des tâches de calcul fastidieuses ; ils peuvent même dans certains cas effectuer des démonstrations élaborées (par exemple en géométrie analytique et en géométrie algébrique grâce aux bases de Gröbner).
– Ils sont un outil d'expérimentation efficace : des calculs qui autrefois nécessitaient beaucoup de temps et d'organisation, peuvent être exécutés en une fraction de seconde, et recommencés autant de fois qu'on le désire en modifiant les différents paramètres à sa guise.
– Ils sont enfin, par leurs fonctions graphiques avancées, un outil de visualisation très commode ; en voici deux exemples (parmi bien d'autres) :
On s'intéresse aux propriétés d'une fonction (variation, signe, extremums etc.) ; avant d'entreprendre des calculs, on commence par tracer des courbes ou des surfaces en faisant varier le cadrage et l'échelle ; les outils de calculs numériques disponibles dans le logiciel permettent d'affiner encore plus ce genre d'exploration.
Une autre situation intéressante se présente lorsqu'on cherche les images de courbes ou de surfaces par des transformations géométriques ; la possibilité de pouvoir les observer facilement à l'écran facilite la résolution des problèmes. À cela s'ajoute le plaisir de découvrir de belles figures et les avantages que l'on peut en tirer sur le plan pédagogique.

Dans cet ouvrage, nous nous proposons de passer en revue les possibilités offertes par le logiciel *Mathematica* dans le domaine du graphisme, et d'étudier quelques applications en géométrie.
Parmi les logiciels de calcul formel actuellement disponibles, *Mathematica* est sans doute celui qui offre le plus de possibilités graphiques ; en particulier d'une part il permet la sortie d'images de très haute qualité au format PostScript, et d'autre part, il met à la disposition de l'utilisateur des outils de programmation efficaces et d'utilisation relativement simple.

Face à un problème à résoudre avec l'aide de *Mathematica*, qu'il soit de nature graphique ou autre, la démarche est toujours à peu près toujours la même :
– On cherche d'abord si les commandes standards ne permettent pas de le résoudre plus ou moins directement ; au début, on s'appuiera sur l'aide en ligne, mais rapidement, il faudra acquérir une bonne connaissance sur l'utilisation de ces commandes (qui, d'ailleurs, en ce qui concerne les graphiques, sont peu nombreuses).
– En cas d'insuffisance de ces dernières, on explorera les possibilités offertes par les librairies externes (fichiers de commandes) ; les fichiers concernant les graphiques sont tous situés dans le sous-répertoire `Graphics` du répertoire `StandardPackages` ; là encore, outre l'aide en ligne, une connaissance même très superficielle des différents fichiers disponibles s'avère utile.
– Enfin en dernier ressort, on peut être amené à créer ses propres outils. On sera ainsi conduit à enrichir la bibliothèque des fichiers de commandes *Mathematica* ; ces fichiers, ainsi que ceux que l'on aura pu se procurer de la main d'autres utilisateurs, ou bien sur le site internet de Wolfram Research (http://www.wolfram.com et plus particulièrement le répertoire MathSource) seront placés (version 3 de *Mathematica*) dans le répertoire `Applications`, où l'on créera des sous-répertoires analogues à ceux du répertoire `StandardPackages`.

Dans cet ouvrage, nous supposons le lecteur familiarisé avec l'utilisation courante de *Mathematica* ; pour l'élaboration des programmes, nous supposerons qu'il possède un minimum de connaissances sur la structure du logiciel et les techniques de programmation disponibles (structures de contrôles, utilisation des règles et définitions, fonctions pures et programmation applicative). Toutefois nous avons rappelé certaines particularités de *Mathematica* au moment de leur utilisation (notamment concernant l'évaluation des fonctions et l'emploi des principales structures utilisées en programmation).

Le lecteur désireux d'étudier le fonctionnement de *Mathematica* pourra consulter les ouvrages de référence du logiciel :

– `The Mathematica book` par Stephen Wolfram (également disponible en français).

– `Standard Add-on Packages` (pour la description des fichiers externes standards).

Ces deux ouvrages sont entièrement reproduits dans leur version électronique dans l'aide en ligne du logiciel.

Pour un apprentissage plus progressif ainsi que des applications et des exercices, on peut aussi consulter l'ouvrage suivant, du même auteur que la présente brochure :

– `Mathematica 3 par la pratique` (aux éditions Eyrolles).

L'auteur recevra avec intérêt les critiques et suggestions que le lecteur voudra bien lui formuler (lui écrire directement – adresse email : ypoit@gulliver.fr).

Quelques remarques sur la présentation de l'ouvrage :

Ce livre a été directement écrit dans *Mathematica* ; le lecteur pourra ainsi se rendre compte de ce qu'il est possible de faire avec la version 3 du logiciel en matière d'édition. Bien que ne présentant pas toutes les fonctionnalités des traitements de textes évolués, ses possibilités sont relativement étendues, notamment en matière d'édition de formules mathématiques.

Les cellules d'entrées et de sorties sont marquées d'un trait vertical avec les indications In[] := et Out[] = qui rappellent les prompteurs utilisés à l'écran. Lorsque les programmes écrits dans les cellules d'entrée sont longs, il ne nous a pas toujours été possible, pour des raisons de place, de respecter les règles de présentation servant à améliorer leur lisibilité ; nous avons pu le faire pour un certain nombre, et le lecteur les reconnaîtra facilement.

Également pour des raisons de place, nous n'avons pas montré toutes les figures correspondant aux commandes étudiées ; le lecteur qui nous suivra à travers les différents chapitres est invité à entrer au fur et à mesure ces commandes sur sa machine et à laisser s'afficher les résultats intermédiaires (en supprimant les point-virgules en fin de commandes).

La disquette d'accompagnement reprend le plan de l'ouvrage (un cahier par chapitre, présenté dans la version 3.01 du logiciel) et permet d'évaluer toutes les commandes ; en outre les programmes complets des fichiers de commandes sont fournis.

Chapitre 1

Utilisation des commandes graphiques

1 - Principes généraux

1.1 - Où trouver les commandes graphiques ?

- Lorsqu'on désire effectuer un certain travail graphique, on a besoin de savoir quel type de commande est la mieux adaptée et où la trouver :

  — en existe-t-il une en standard ?

  — sinon dans quels fichiers de commandes peut-on trouver ce que l'on cherche ?

  — et en dernier ressort comment construire soi-même une telle fonction ?

- Les commandes graphiques *Mathematica* se classent en trois grandes catégories :

  — les commandes représentant des listes de données numériques (`ListPlot`, `PlistPlot3D` etc.)

  — les commandes représentant graphiquement une ou plusieurs fonctions mathématiques (`Plot`, `ParametricPlot`, `Plot3D` etc.)

  — les commandes servant à manipuler des graphiques déjà produits (`Show`, `Animate` etc.)

- Nous donnons la liste de toutes les commandes graphiques standards *Mathematica* :

  — les commandes qui représentent des listes ou des fonctions mathématiques dans le plan :

  ListPlot, Plot, ParametricPlot

  — les commandes qui représentent des listes ou des fonctions mathématiques dans l'espace :

  ListPlot3D, Plot3D, ParametricPlot3D

  ListContourPlot, ContourPlot, ListDensityPlot, DensityPlot

  Remarquons que ces quatre dernières fonctions effectuent des représentations planes.

  — la commande **Show** qui retrace et combine des graphiques existants.

- Les fichiers de commandes graphiques sont tous regroupés dans le sous-répertoire *Graphics* du répertoire *AddOns* (*Packages* pour la version 2). Chaque fichier comporte souvent un grand nombre de fonctions graphiques.

Nous donnons les principaux (mais pas les commandes; se reporter pour cela à l'aide *Mathematica* ou bien plus loin dans cet ouvrage où certaines d'entre elles sont étudiées) :

— Graphics`Animation` (utilitaires pour réaliser des animation) .

Ce fichier est souvent chargé automatiquement au démarrage.

— Graphics`Arrow` (produit une primitive graphique qui dessine des flèches)

— Graphics`Colors` (produit un très grand nombre de couleurs)

— Graphics`ComplexMap` (représente des fonctions complexes dans le plan)

— Graphics`ContourPlot3D` (représente des surfaces à partir d'équations implicites ainsi que des surfaces de niveaux)

— Graphics`FilledPlot` (colorie les zones situées entre des courbes planes)

— Graphics`Graphics` (produit un grand nombre d'utilitaires pour les graphiques en deux dimensions)

— Graphics`Graphics3D` (produit des utilitaires pour les graphiques en trois dimensions)

— Graphics`ImplicitPlot` (dessine des courbes planes à partir de leurs équations implicites)

— Graphics`Legend` (produit des légendes à combiner avec les graphiques)

— Graphics`MultipleListPlot` (représente des listes numériques dans le plan)

— Graphics`PlotField` (représente des champs de vecteurs dans le plan)

— Graphics`PlotField3D` (représente des champs de vecteurs dans l'espace)

— Graphics`Polyhedra` (représente les polyèdres réguliers de l'espace)

— Graphics`Shapes` (représente des surfaces usuelles de l'espace)

— Graphics`Spline` (produit une primitive graphique qui dessine des splines)

— Graphics`SurfaceOfRevolution` (représente des surfaces de révolution)

— Graphics`ThreeScript` (produit une fonction d'interface pour convertir les graphiques 3D au format "3-Script")

- Pour apprendre à construire sa propre fonction graphique ou son propre fichier de commandes, voir la suite de cet ouvrage.

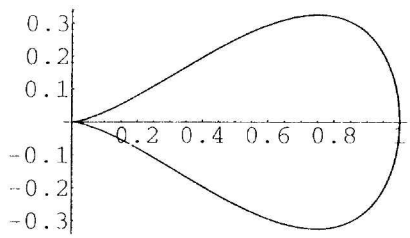
1.2 - Notions sur le mécanisme de génération des graphiques

1.2.1 - Production d'un graphique

Entrons une commande graphique simple et évaluons-là :

```
In[] := poire = Plot[{x Sqrt[x (1 - x)], -x Sqrt[x (1 - x)]}, {x, 0, 1},
          AspectRatio -> Automatic]
```

-Figure 1-




```
Out[] = - Graphics -
```

Comme toute fonction *Mathematica*, notre commande produit une expression en retour (à moins qu'on ne la termine par un point-virgule); ce résultat (que nous avons assigné au symbole `poire`) a une forme de sortie réduite au symbole `Graphics`; l'examen de sa forme d'entrée (dont nous n'affichons qu'une partie) montre qu'il s'agit d'une expression encombrante dont `Graphics` n'est que le type (tête) et qui contient toutes les informations nécessaires au tracé du graphique :

```
In[] := Short[InputForm[poire], 3]

Out[] = Graphics[{{{Line[<<1>>]}}, {{Line[<<1>>]}}}, {PlotRange ->
          Automatic, AspectRatio -> Automatic, DisplayFunction :>
          $DisplayFunction, <<21>>, FormatType :> $FormatType}]
```

Le premier argument contient les listes des coordonnées des points calculés à partir des deux fonctions à représenter.

```
In[] := Short[poire[[1, 1]], 4]

Out[] = {{Line[{{4.16667×10^-8, 8.50517×10^-12}, {0.00123443, 0.0000433441},
      <<67>>, {0.998992, 0.0316999}, {1., 0.000204124}}]}}
```

En "effet de bord", nous avons obtenu la production du graphique dans une cellule particulière (de style graphique). Ce graphique n'est pas qu'une simple image écran, mais derrière elle se trouve une traduction en langage *PostScript* de l'expression *Mathematica* précédente; cette représentation *PostScript* de l'image permet une reproduction de haute qualité sur différents périphériques (écrans, imprimantes, photocomposeuses).

Pour observer le programme *PostScript*, il suffit de convertir la cellule graphique (à l'aide de l'item de **menu** `Show Expression` en version 3, de l'item de menu `Formatted` en version 2). Nous n'en montrons ci-dessous qu'une toute petite partie, ce programme étant très long :

```
Cell[GraphicsData["PostScript", "\<\
%!
%%Creator: Mathematica
%%AspectRatio: .64952
MathPictureStart
/Mabs {
Mgmatrix idtransform
Mtmatrix dtransform
} bind def
/Mabsadd { Mabs
3 -1 roll add
3 1 roll add
exch } bind def
%% Graphics
/Courier findfont 10  scalefont  setfont
% Scaling calculations
0.0238095 0.952381 0.324759 0.952381 [
[.21429 .31226 -9 -9 ]
.....................................
.97523 .29457 L
.97619 .32456 L
Mfstroke
% End of Graphics
MathPictureEnd
\
\>"], "Graphics",
  CellLabel->"From In[13]:=",
  ImageSize->{168.5, 109.438},
  ImageMargins->{{4, 0}, {0, 0}},
  ImageRegion->{{0, 1}, {0, 1}}]
```

Cette façon de faire n'est pas économique, car un graphique est en fait stocké deux fois en mémoire : sous forme d'expression *Mathematica* et sous forme de programme *PostScript*; mais cela garantit une grande souplesse d'utilisation ainsi qu'une très haute qualité des figures.

1.2.2 - La fonction Show

Revenons au graphique `poire` précédent; pour produire ce graphique, la fonction `Plot` a procédé ainsi :

– elle a d'abord calculé les valeurs des fonctions à représenter à partir de points d'échantillonnage sur l'intervalle précisé par l'itérateur {x,0,1}

– à partir des données numériques obtenues, elle a fabriqué l'objet graphique `poire`; il s'agit d'une expression de type `Graphics` dont le premier argument comporte des fonctions (primitives graphiques) indiquant ce qu'il faut faire des données numériques; le second argument est une liste d'options concernant la présentation générale du graphique

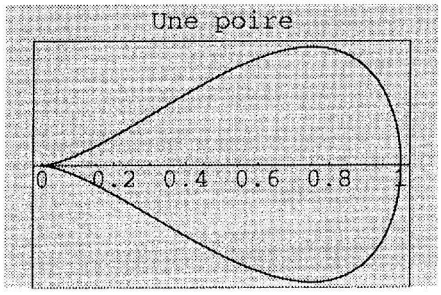
– cette expression a été ensuite convertie en un programme *PostScript*; la partie frontale de *Mathematica* possède un interpréteur *PostScript* capable de dessiner la figure à l'écran.

On peut vouloir réemployer ce graphique pour en modifier certaines caractéristiques qui ne nécessitent pas d'intervention sur les données numériques (c'est à dire sur le premier argument de l'expression); on peut vouloir aussi combiner plusieurs graphiques.

Il est inutile dans ce cas de refaire appel à la fonction `Plot`; la fonction `Show` prend en argument une ou plusieurs expressions graphiques pour en effectuer la concaténation et retourner une combinaison de ces graphiques; on peut en profiter pour modifier les valeurs des options.

```
In[] := Show[poire, Background -> Hue[0.5], PlotLabel -> "Une poire",
        Frame -> True, FrameTicks -> None]
```

-Figure 2-




```
Out[] = - Graphics -
```

Remarquons que la fonction `Show` a retourné une nouvelle expression graphique dans laquelle les nouvelles valeurs des options ont été prises en compte.

Notons que l'évaluation de l'expression graphique ne produit pas de graphique (il faut passer par `Show`) :

```
In[] := poire
Out[] = - Graphics -
```

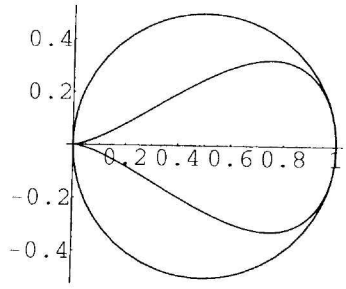
Ci-dessous, nous traçons un cercle de diamètre $[OA]$ avec $O(0, 0)$ et $A(1, 0)$; l'option `DisplayFunction` prend comme valeur normale `$DisplayFunction` (cette variable globale contient une fonction qui dirige le graphique vers l'écran; voir aussi à ce sujet le chapitre 2. En donnant à cette option la valeur `Identity`, aucun graphique n'est dessiné.

```
In[] := cercle = ParametricPlot[{1/2 (1 + Cos[t]), 1/2 Sin[t]}, {t, 0, 2 π},
          AspectRatio -> Automatic, DisplayFunction -> Identity];
```

Nous pouvons, à l'aide de la fonction `Show`, superposer les deux graphiques; l'expression retournée est une concaténation des expressions `poire` et `cercle`.

```
In[] := | Show[cercle, poire, DisplayFunction :> $DisplayFunction ]
```

-Figure 3-




```
Out[] = | - Graphics -
```

Exercice 1 :

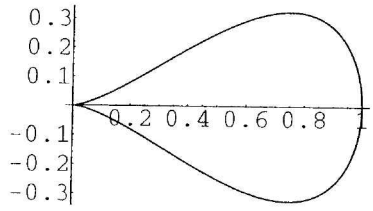
Dans la commande précédente, est-ce nécessaire de préciser l'option :
`DisplayFunction :> $DisplayFunction` ?

1.3 - Interventions sur un graphique à l'aide de la partie frontale

Nous donnons ci-dessous un aperçu rapide des possibilités de manipulations de graphiques à l'aide des commandes de la partie frontale. Nous n'entrons pas trop dans les détails car les possibilités varient avec la plateforme utilisée, ainsi que le détail des items de menus et des raccourcis-clavier. Le lecteur vérifiera par l'aide en ligne que les fonctionnalités indiquées existent sur sa machine, et comment les réaliser dans la pratique.

```
In[] := | Show[poire ]
```

-Figure 4-




```
Out[] = | - Graphics -
```

- **Déplacement et recadrage du graphique; utilisation de la règle**

  La sélection d'un graphique peut se faire en cliquant sur le crochet de sa cellule, ou en cliquant directement dessus. Dans ce dernier cas, on fait apparaître un cadre avec des poignées, comme dans les logiciels de dessins vectorisés.

  On peut, en utilisant les poignées, recadrer le graphique. Ses proportions sont conservées, sauf si on appuie simultanément sur la touche de majuscule. En faisant glisser le curseur de la souris, le graphique peut être déplacé horizontalement. Pendant ces manoeuvres, la partie de la règle correspondant à la position du graphique est en vidéo inverse, ce qui permet des recadrages précis. Enfin un item de menu permet de revenir à la taille initiale du graphique.

- **Conversion des graphiques**

  Un item de menu (`Cell/Convert To` pour la version 3) permet de convertir le graphique en un autre format (le format par défaut étant bien sûr `PostScript`).

  Les formats disponibles dépendent de la plateforme; on y trouve toujours le format `Bitmap` (tableau de pixels); il peut permettre d'alléger le poids du graphique en mémoire si on n'a pas en vue des impressions de qualité (par exemple pour des animations).

- **Sauvegarde des graphiques**

  Lorsqu'on désire exporter des graphiques, il est aussi nécessaire de les convertir en un format reconnaissable par l'application qui les recevra. On utilise pour cela un autre item de menu (`Edit/Save Selection As` pour la version 3).

  Un format couramment utilisé est `EPS` ou `EPSF` (Encapsuled PostScript Format) qui est reconnu par la plupart des logiciels de dessin, traitement de texte, mise en page, et garantit la haute qualité du graphique produit par Mathematica.

- **Lecture des coordonnées**

  Lorsque le graphique est sélectionné, on a la possibilité de lire les coordonnées de la position du curseur de la souris dans la barre d'état de la fenêtre; on appuie pour cela simultanément sur une touche (touche Commande sur Macintosh). Ces coordonnées sont les coordonnées réelles pour un graphique 2D, et les coordonnées écran (comprises entre 0 et 1) pour les graphiques 3D.

  Si on clique ou que l'on fait glisser la souris, on dessine des points sur le graphique dont on peut copier les coordonnées dans le presse-papier pour les coller ensuite dans une prochaine entrée.

  Si on maintient une seconde touche enfoncée (touche Option sur Macintosh), on dessine un rectangle dont on peut copier les coordonnées pour les coller ensuite comme valeur de l'option PlotRange; on peut ainsi montrer des "zooms" du graphique initial.

```
In[] := | Show[%, PlotRange -> <contenu du presse-papier>]
```

Exercice 2 :

Comment peut-on recadrer une zone d'un graphique 3D sélectionnée à la souris, à l'aide de l'option `PlotRange` ?

- **Sélection d'un point de vue (figures 3D)**

  Nous verrons au chapitre 4 plus en détail le fonctionnement des options pour les figures de l'espace; l'option `ViewPoint` est couramment utilisée pour modifier la projection de la figure; le lecteur se reportera à ce chapitre pour étudier le système de coordonnées utilisé pour fixer le point de vue.

  L'item de menu `Input/3D ViewPoint Selector` permet de choisir ce point de vue de façon plus intuitive qu'en entrant directement ses coordonnées. On modifie à l'aide de la souris le cube représentant la boite contenant le graphique; les coordonnées (rectangulaires ou sphériques) s'affichent au fur et à mesure. Le bouton `Paste` permet de copier le nom de l'option et sa valeur.

- **Sélection d'une couleur**

  Lorsqu'on désire donner à une directive de couleur (principalement `RGBColor`) une valeur, il n'est pas toujours facile de prévoir le résultat. On se sert alors de l'item de menu `Input/Color Selector`; il affiche à l'écran une roue des couleurs. On copie dans le presse-papier la valeur choisie pour la primitive RGBColor, que l'on peut ensuite coller en valeur d'une option (comme `Background`, `DefaultColor`, `AxesStyle`, `PlotStyle` etc.).

  Voir chapitres 2 et 4 pour plus de détails sur les directives de couleurs et l'éclairage des figures.

2 - Les principales commandes graphiques

2.1 - La représentation graphique de listes de données

Un graphique est très souvent la représentation géométrique d'une liste de données numériques (cette liste pouvant avoir une structure plus ou moins complexe). Ces données peuvent être fournies directement ou être le résultat de la tabulation d'une ou plusieurs fonctions numériques.

Les fonctions que nous étudions dans cette sous-section prennent en argument une liste de données et construisent un graphique; les fonctions étudiées dans la sous-section suivante prennent en argument une ou plusieurs fonctions numériques.

2.1.1 - Représentation de points du plan par une liste de couples

Une liste de couples de réels $\{\{x_1, y_1\}, \{x_2, y_2\}, ..., \{x_n, y_n\}\}$ représente une suite de points du plan que l'on peut représenter par la fonction `ListPlot`.

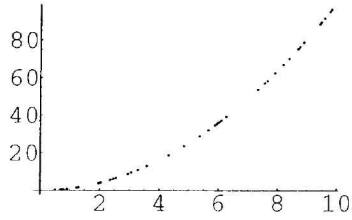
```
In[] := | liste = Table[{x = Random[Real, 10], x^2}, {x, 0, 10, 0.2}];
```

Remarquons que la liste ci-dessus ne donne pas le même résultat que :

```
In[] := | Table[{Random[Real, 10], Random[Real, 10]^2}, {x, 0, 10, 0.2}];
```

```
In[] := | ListPlot[liste];
```

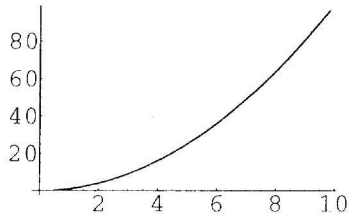
-Figure 5-



L'option `PlotJoined` relie les points par un segment (ici, il faut trier la liste) :

```
In[] := | ListPlot[Sort[liste], PlotJoined -> True];
```

-Figure 6-



Remarquons que la fonction `ListPlot` admet comme argument une liste simple $\{x_1, x_2, ..., x_n\}$; dans ce cas, c'est la liste $\{\{1, x_1\}, \{2, x_2\}, ..., \{n, x_n\}\}$ qui est représentée.

2.1.2- Représentation de points de l'espace par une liste de triplets

Une liste de triplets de réels $\{\{x_1, y_1, z_1\}, \{x_2, y_2, z_2\}, ..., \{x_n, y_n, z_n\}\}$ représente une suite de points de l'espace représentables par la fonction `ScatterPlot3D` du fichier de commandes `Graphics`Graphics3D`` :

```
In[] := | liste =
          Table[{x = Random[Real, 10], Sqrt[x], Log[x]}, {x, 0, 10, 0.2}];
```

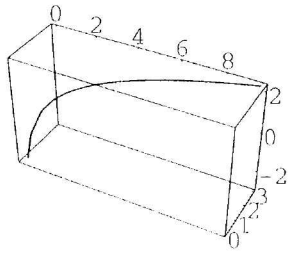
```
In[] := | << Graphics`Graphics3D`
```

```
In[] := | ScatterPlot3D[liste];
```

(Figure non montrée)

```
In[] := | ScatterPlot3D[Sort[liste], PlotJoined -> True];
```

-Figure 7-



2.1.3 - Représentation de points de l'espace par une matrice de nombres

Une matrice de nombres réels :

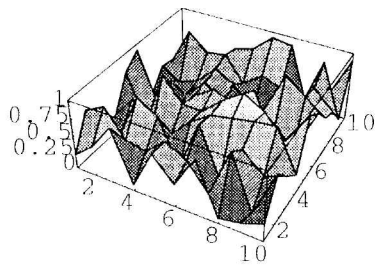
$$\{\{z_{11}, z_{12}, ..., z_{1p}\}, \{z_{21}, z_{22}, ..., z_{2p}\}, ..., \{z_{q1}, z_{q2}, ..., z_{qp}\}\}$$

représente une liste de points de cotes z_{ij} sur les noeuds d'un quadrillage du rectangle $[1, p]\times[1, q]$ du plan des (x, y) en p subdivisions horizontales et q subdivisions verticales.

La fonction standard `ListPlot3D` représente cette liste en traçant des polygones ayant ces points pour sommets. La fonction accepte en second argument une matrice de dimensions $(q-1)\times(p-1)$ dont les termes sont des valeurs de directives de couleurs (`GrayLevel`, `RGBColor`, `Hue`) qui colorient les polygones. En l'absence de ce second argument, ces derniers sont coloriés par l'éclairage simulé (voir chapitre 4). Avec l'option `Lighting` à `False`, les polygones sont coloriés en fonction de l'altitude; la fonction de coloriage est passée en argument de l'option `ColorFunction` (`GrayLevel` par défaut).

```
In[] := matrice = Table[Random[], {10}, {10}];

In[] := ListPlot3D[matrice];

In[] := ListPlot3D[matrice, Lighting -> False];
```

-Figure 8-




```
In[] := ListPlot3D[matrice, Lighting -> False, ColorFunction -> Hue];
```

(Figure non montrée)

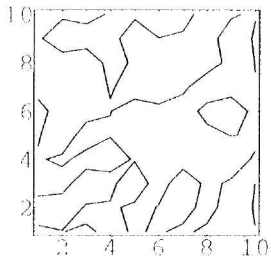
```
In[] := couleurs = Table[Hue[Random[]], {9}, {9}];

In[] := ListPlot3D[matrice, couleurs];
```

(Figure non montrée)

La fonction standard `ListContourPlot` donne une représentation plane la même surface par des lignes de niveaux. L'option `Contours` règle le nombre de lignes de niveaux (en intoduisant une liste de nombres on fait tracer les lignes de niveaux correspondant à ces valeurs). L'option `ContourShading` précise si on veut colorier les zones situées entre les lignes de niveaux..

```
In[] := ListContourPlot[matrice, Contours -> {1/2},
         ContourShading -> False];
```

-Figure 9-



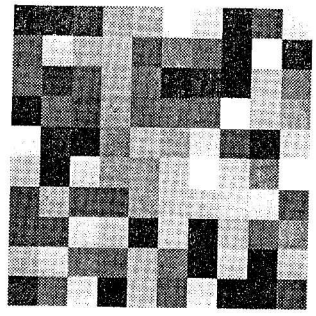

```
In[] := ListContourPlot[matrice, Contours -> 7, ColorFunction -> Hue];
```

(Figure non montrée)

La fonction standard `ListDensityPlot` donne une représentation plane la surface par un tableau de cellules colorées. La couleur est réglée par l'option `ColorFunction` (`GrayLevel` par défaut).

```
In[] := | ListDensityPlot[matrice, Mesh -> False, Frame -> False];
```

-Figure 10-

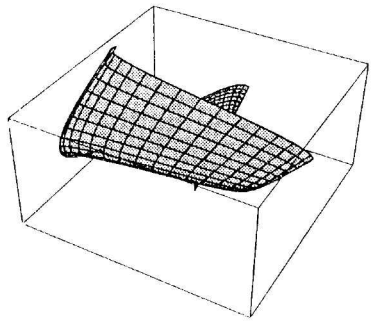


2.1.4 - Représentation de points de l'espace par une matrice de triplets

On peut enfin représenter une matrice $(m_{ij})_{1\le i\le q, 1\le j\le p}$ où les m_{ij} sont des triplets de nombres réels : $m_{ij} = \{x_{ij}, y_{ij}, z_{ij}\}$. Il s'agit d'une généralisation de la situation précédente. La fonction qui représente ce type de liste en joignant les points 4 par 4 pour former des polygones, est la fonction `ListSurfacePlot3D` du fichier de commandes `Graphics`Graphics3D`` :

```
In[] := | nappe = Table[{u Sin[v], v Cos[u], (u + v) / 2}, {u, 0, 5, 0.2},
            {v, 0, 5, 0.2}];

In[] := | ListSurfacePlot3D[nappe];
```

-Figure 11-



Nous verrons plus loin que la fonction standard `ParametricPlot3D` représente directement une nappe paramétrée à partir de 3 fonctions.

2.2 - La représentation graphique de fonctions mathématiques

2.2.1 - Principe général de fonctionnement; évaluation

- Ces fonctions prennent en arguments :
  - une ou plusieurs expressions définissant les fonctions mathématiques à représenter
  - un itérateur $\{t, t_{min}, t_{max}\}$ pour une courbe
  - deux itérateurs $\{u, u_{min}, u_{max}\}$, $\{v, v_{min}, v_{max}\}$ pour une surface
  - des options
- L'évaluation de ces fonctions comporte deux étapes :
  - le calcul numérique des expressions constituant le premier argument
  - le traitement des listes numériques obtenues, ce qui nous ramène aux fonctions étudiées dans la sous-section 2.1 (production d'une expression *Mathematica*, et génération d'un programme *PostScript*)

- La façon dont s'effectue la première étape de cette évaluation est importante à comprendre :

  – D'abord le premier argument contenant les expressions à calculer n'est pas immédiatement évalué (pas plus d'ailleurs que les suivants comme on le voit ci-dessous) :

```
In[] := listeCom = {Plot, ParametricPlot, Plot3D, ContourPlot,
          DensityPlot, ParametricPlot3D};

In[] := Attributes /@ listeCom

Out[] = {{HoldAll, Protected}, {HoldAll, Protected}, {HoldAll, Protected},
         {HoldAll, Protected}, {HoldAll, Protected}, {HoldAll, Protected}}
```

  – À partir des valeurs des itérateurs et de la valeur de l'option `PlotPoints`, une suite de points d'échantillonnage est calculée : $\{t_{min} = t_0, t_1, \ldots, t_n = t_{max}\}$.

```
In[] := Options[#, PlotPoints] & /@ listeCom

Out[] = {{PlotPoints → 25}, {PlotPoints → 25}, {PlotPoints → 15},
         {PlotPoints → 15}, {PlotPoints → 15}, {PlotPoints → Automatic}}
```

  Pour une surface, une valeur n correspond à un quadrillage $n \times n$ (on peut entrer des valeurs rectangulaires $\{n, m\}$); la valeur `Automatic` pour `ParametricPlot3D` correspond aux valeurs 75 pour les courbes gauches et 15 pour les sufaces paramétrées.

  – Le premier membre de la fonction est ensuite évalué numériquement en chacun des points d'échantillonnage, produisant des nombres ou des listes de nombres (il s'agit de flottants en précision machine).

  Dans la première évaluation ci-dessous de Plot, en chaque point d'échantillonnage, la somme de monômes est recalculée pour la valeur correspondante de x :

```
In[] := Timing[Plot[Sum[x^n/n!, {n, 0, 5}], {x, -2, 4}, PlotPoints -> 500,
          DisplayFunction -> Identity]][[1]]
Out[] = 1.06667 Second
```

  Dans la seconde, cette somme est déjà effectuée, mais on a autant d'appels au symbole `expr`, ce qui retarde encore l'évaluation :

```
In[] := expr = Sum[x^n/n!, {n, 0, 5}];

In[] := Timing[Plot[expr, {x, -2, 4}, PlotPoints -> 500,
          DisplayFunction -> Identity]][[1]]
Out[] = 0.35 Second
```

  Enfin, les trois commandes ci-dessous produisent pratiquement les mêmes calculs (les différences de temps ne sont pas significatives) :

```
In[] := Timing[Plot[Evaluate[Sum[x^n/n!, {n, 0, 5}]], {x, -2, 4},
          PlotPoints -> 500, DisplayFunction -> Identity]][[1]]
Out[] = 0.0666667 Second

In[] := Timing[Plot[Evaluate[expr], {x, -2, 4}, PlotPoints -> 500,
          DisplayFunction -> Identity]][[1]]
Out[] = 0.05 Second
```

```
In[] := Timing[Plot[1 + x + x^2/2 + x^3/6 + x^4/24 + x^5/120, {x, -2, 4},
          PlotPoints -> 500, DisplayFunction -> Identity]][[1]]
Out[] = 0.0666667 Second
```

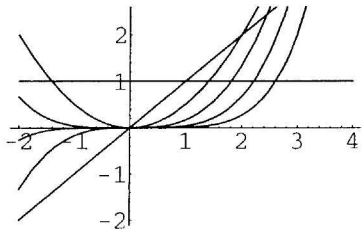
L'emploi de la fonction `Evaluate` dans les commandes graphiques est très fréquente; elle est même parfois indispensable, comme dans l'exemple ci-dessous où on veut représenter simultanément plusieurs fonctions : le premier argument, qui n'a pas une structure de liste, doit, pour chaque valeur de x, produire un nombre (or il produit une liste de nombres) :

```
In[] := Plot[Table[x^n/n!, {n, 0, 5}], {x, -2, 4},
         DisplayFunction -> Identity];
Plot::plnr :
 Table[x^n/n!, {n, 0, 5}] is not a machine-size real number at x = -2..
```

Maintenant, c'est une liste d'expressions que *Mathematica* trouve dans le premier argument de `Plot` :

```
In[] := Plot[Evaluate[Table[x^n/n!, {n, 0, 5}]], {x, -2, 4}];
```

-Figure 12-



2.2.2 - Représentation de courbes planes

On utilise `Plot` pour représenter les courbes explicites $y = f(x)$ et `ParametricPlot` pour les courbes paramétrées $\{x = f(t), y = g(t)\}$. Si on veut représenter des courbes d'équation polaire $\rho = f(\theta)$, on représente paramétriquement la courbe $\{x = f(\theta)\cos\thea, y = f(\theta)\sin\theta\}$ (on peut également utiliser la fonction `PolarPlot` du fichier de commandes ``Graphics`Graphics` ``).

Exercice 3 :

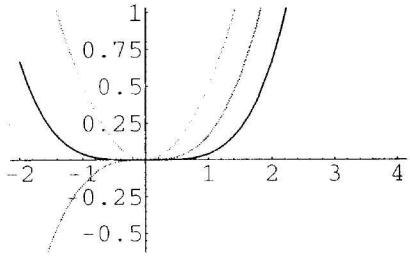
On veut représenter une fonction sur un domaine qui n'est pas un intervalle, mais une réunion d'intervalles; par exemple la fonction $\sqrt{(x-2)(x+1)}$ sur $]-\infty, -1] \cup [2, +\infty[$; comment faire ?

En entrant une liste de fonctions (pour `Plot`) ou une liste de liste de couples de fonctions (pour `ParametricPlot`), on peut superposer plusieurs courbes.

L'option `PlotStyle` permet de différencier les styles de chaque courbe en appliquant en parallèle à chacune une directive (et même une liste de directives). La valeur {} (liste vide) a un effet neutre.

```
In[] := Plot[{x^2/2, x^3/6, x^4/24}, {x, -2, 4},
         PlotStyle -> {GrayLevel[0.75], GrayLevel[0.5], {}}];
```

-Figure 13-

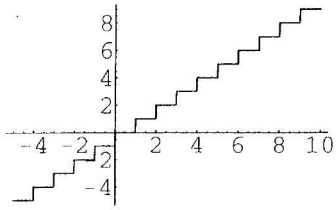


⊟ Exercice 4 :

La commande suivante représente la fonction partie entière avec des "contre-marches" indésirables :

```
In[] := Plot[Floor[x], {x, -5, 10}];
```

-Figure 14-



Comment éviter ces "contre-marches" ?

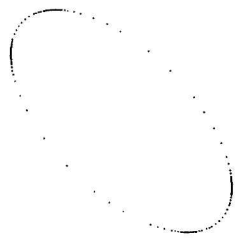
Dans le cas des commandes graphiques produisant des courbes planes, les expressions à représenter ne sont pas calculées seulement aux valeurs déterminées par les bornes de l'itérateur et les valeurs de l'option `PlotPoints`. Ces valeurs ne sont qu'un point de départ, et un algorithme de lissage est utilisé, qui augmente les points de calcul si nécessaire.

Le nombre de points d'échantillonnage est augmenté avec la courbure de la courbe et également lorsque la tangente tend à devenir parallèle à un axe de coordonnées. On peut contrôler les paramètres de ce processus de lissage à l'aide des options `MaxBend` et `PlotDivisions`.

L'exemple ci-dessous montre les parties de la courbe où le nombre de points a été augmenté (le lecteur pourra se reporter au chapitre suivant pour comprendre l'utilisation des primitives graphiques dans les programmes) :

```
In[] := fig = ParametricPlot[{Cos[t] + 2 Sin[t], Cos[t] - 2 Sin[t]},
          {t, 0, 2 π}, AspectRatio -> Automatic,
          DisplayFunction -> Identity, Axes -> False];

In[] := Show[fig /. Line[liste_] :> Map[Point, liste],
         DisplayFunction -> $DisplayFunction];
```

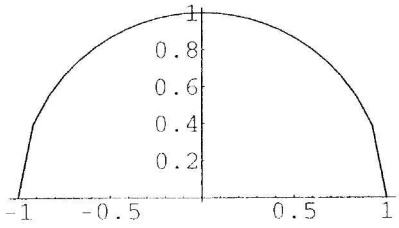
-Figure 15-



L'exemple ci-dessous montre pourquoi augmenter les points d'échantillonnage lorsque la pente de la tangente est grande (cela est surtout utile pour les courbes $y = f(x)$). Nous montrons d'abord une fonction tracée sans lissage (les points d'échantillonnage sont régulièrement répartis sur l'axe des abscisses) :

```
In[] := arc1 = Plot[√(1 - x^2), {x, -1, 1}, AspectRatio -> Automatic,
          MaxBend -> 45, PlotDivision -> 1];
```

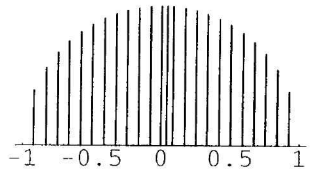
-Figure 16-




```
In[] := {Options[Plot, PlotPoints][[1, 2]], Length[arc1[[1, 1, 1, 1]]]}
Out[] = {25, 26}
```

```
In[] := Show[arc1 /. Line[liste_] :> Map[Line[{{#[[1]], 0}, #}]&, liste],
         Axes -> {True, False}];
```

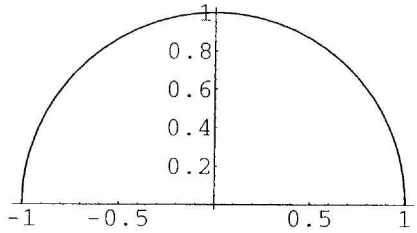
-Figure 17-



Voici le même arc, avec lissage (les valeurs de `MaxBend`, `PlotDivision` remises à leur valeur par défaut) :

```
In[] := arc2 = Plot[Sqrt[1 - x^2], {x, -1, 1}, AspectRatio -> Automatic];
```

-Figure 18-




```
In[] := {Options[Plot, PlotPoints][[1, 2]], Length[arc2[[1, 1, 1, 1]]]}
Out[] = {25, 73}
```

```
In[] := Show[arc2 /. Line[liste_] :>
          Prepend[Map[Point, liste], PointSize[1/75]], Axes -> False];
```

-Figure 19-

Exercice 5 :

Trouver une courbe paramétrée pour laquelle le système de lissage est pris en défaut (on rappelle que, à courbure égale, le lissage est meilleur pour les points à tangente de petite ou de grande pente).

2.2.3 - Représentation de surfaces $z = f(x, y)$

Les trois fonctions `Plot3D`, `ContourPlot`, `DensityPlot` représentent une fonction de deux variables $f(x, y)$ sur un rectangle $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$.

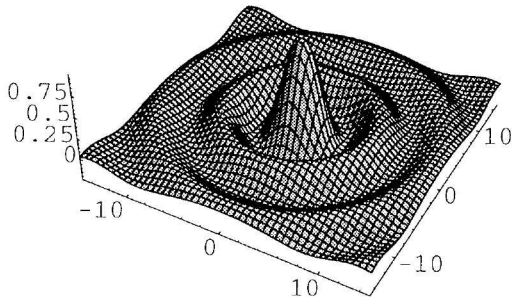
À partir de la valeur $\{p, q\}$ de l'option `PlotPoints` ({15, 15} par défaut), les valeurs de la fonction sont calculées et le premier argument de l'expression retournée est une matrice $(f_{i,j})$ de dimensions (p, q) (le second étant une liste d'options). Seul le type de l'expression est différent : `SurfaceGraphics`, `ContourGraphics`, `DensityGraphics`.

```
In[] := f[0., 0.] = 1

        f[x_, y_] = Sin[Sqrt[x^2 + y^2]] / Sqrt[x^2 + y^2]
```

(Sortie non montrée)

```
In[] := surf = Plot3D[f[x, y], {x, -15, 15}, {y, -15, 15},
         PlotPoints -> 50, PlotRange -> All, Boxed -> False]
```

-Figure 20-




```
Out[] = - SurfaceGraphics -
```

Il n'y a pas de lissage; si on désire affiner le maillage au voisinage de (0, 0), il faut augmenter la valeur de `PlotPoints`, sachant qu'on encombre vite la mémoire de la machine.

Voici la structure du premier argument de `surf` :

```
In[] := Dimensions[surf[[1]]]
Out[] = {50, 50}

In[] := Short[surf[[1]], 5]
Out[] = {{0.033084, 0.0449523, 0.0490289, 0.0448191, 0.03328,
           <<41>>, 0.0448191, 0.0490289, 0.0449523, 0.033084},
          <<49>>}
```

Pour des détails sur l'éclairage de la surface, voir au chapitre 4.

On peut introduire dans le premier argument de `Plot3D` une directive de couleur dont l'argument est une fonction de x et de y :

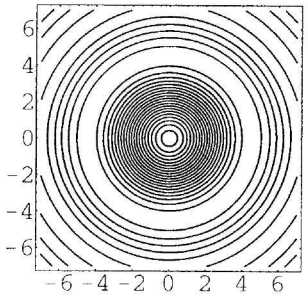
```
In[] := Plot3D[{f[x, y], Hue[ArcTan[y / x]]}, {x, -15, 15}, {y, -15, 15},
         PlotPoints -> 50, PlotRange -> All, Boxed -> False];
```

(Figure non montrée)

Si on veut que la fonction `ContourPlot` trace des lignes de niveaux précises, il ne faut pas hésiter à augmenter la valeur de `PlotPoints`.

```
In[] := ContourPlot[f[x, y], {x, -7, 7}, {y, -7, 7},
         PlotPoints -> 75, Contours -> 20, PlotRange -> All,
         ContourShading -> False]
```

Figure 21-

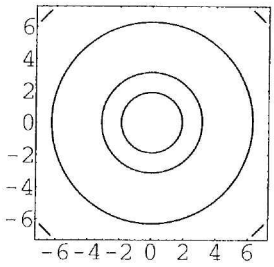



```
Out[] = - ContourGraphics -
```

On peut demander à tracer des lignes de niveaux de valeurs données :

```
In[] := ContourPlot[f[x, y], {x, -7, 7}, {y, -7, 7},
         PlotPoints -> 75, Contours -> {0, 1/2, -1},
         ContourShading -> False];
```

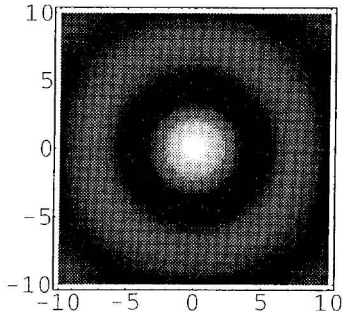
-Figure 22-



La fonction `DensityPlot` permet d'obtenir de belles figures là encore si on augmente la valeur de `PlotPoints` et que l'on supprime les contours des cellules à l'aide de l'option `Mesh`.

```
In[] := DensityPlot[f[x, y], {x, -10, 10}, {y, -10, 10},
         PlotPoints -> 80, Mesh -> False, PlotRange -> All]
```

-Figure 23-




```
Out[] = - DensityGraphics -
```

Les trois fonctions précédentes admettent l'option `ColorFunction` à laquelle on passe une fonction d'une variable, qui doit retourner les valeurs d'une directive de couleur; la variable de la fonction prend les valeurs de $f(x, y)$ (altitude). Pour que cette option soit effective, l'option `Lighting` de `Plot3D` doit être à `False`, et l'option `ContourShading` de `ContourPlot` à `True`.

Dans les exemples ci-dessous nous en profitons pour montrer que :

– ces options peuvent se passer directement dans `Show` sans avoir à recalculer $f(x, y)$ (ce sont en fait des options attachées aux fonctions `SurfaceGraphics, ContourGraphics, DensityGraphics`)

– l'objet `surf` de type `SurfaceGraphics` construit par `Plot3D` peut être aisément converti en objet d'un autre type; là encore cela évite de recalculer la fonction (par contre il faudrait le faire bien sûr si on voulait modifier la valeur d'une option comme `PlotPoints`).

```
In[] := Show[surf, Lighting -> False,
         ColorFunction -> (RGBColor[#^(1/3), #^(1/4), #^(1/2)]&)];
```

(Figure non montrée)

```
In[] := Show[ContourGraphics[surf],
         ColorFunction -> (RGBColor[#^(1/3), #^(1/4), #^(1/2)]&)];
```

(Figure non montrée)

```
In[] := Show[DensityGraphics[surf], Mesh -> False,
         ColorFunction -> (RGBColor[#^(1/3), #^(1/4), #^(1/2)]&)];
```

(Figure non montrée)

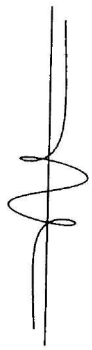
2.2.4 - Représentation de courbes et surfaces paramétrées

Si on veut représenter des courbes et des surfaces dans l'espace à partir d'un système d'équations paramétriques, on utilise la fonction `ParametricPlot3D`.

Le premier argument de cette fonction est une liste de trois fonctions. Si ces fonctions dépendent d'une seule variable, on obtiendra une courbe de l'espace; dans ce cas on devra augmenter la valeur de `PlotPoints`.

```
In[] := ParametricPlot3D[{Sin[3 t], Sin[2 t], Tan[t / 2]}, {t, 0, 2 π},
         Boxed -> False, Axes -> False, PlotPoints -> 100]
```

-Figure 24-



```
Out[] = - Graphics3D -
```

On peut introduire une quatrième fonction du paramètre dans la liste, qui doit retourner les valeurs d'une directive de couleur :

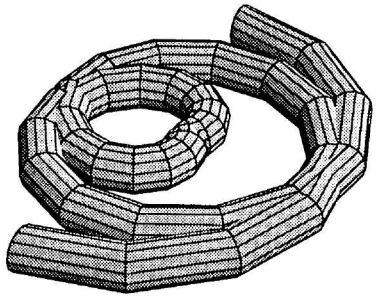
```
In[] := ParametricPlot3D[
         {Sin[3 t], Sin[2 t], Tan[t / 2], Hue[t / (2 π)]}, {t, 0, 2 π},
         Boxed -> False, Axes -> False, PlotPoints -> 100];
```

(Figure non montrée)

Avec des fonctions de 2 variables (et deux itérateurs), on aura des surfaces paramétrées (on peut là encore introduire une quatrième fonction pour colorier les polygones).

```
In[] := ParametricPlot3D[{Cos[u] + u Sin[u] - Sin[u] Cos[v],
          Sin[u] - u Cos[u] + Cos[u] Cos[v], Sin[v]},
         {u, -3 π, 3 π}, {v, -π, π},
         Boxed -> False, Axes -> False, PlotPoints -> {40, 15}]
```

-Figure 25-



```
Out[] = - Graphics3D -
```

Les objets graphiques produits par cette fonction sont de type `Graphics3D`.

2.2.5 - Représentation de courbes et surfaces implicites

Pour tracer une courbe à partir d'une équation implicite $f(x, y) = 0$, on utilise la fonction `ImplicitPlot` du fichier de commandes `Graphics`ImplicitPlot``.

```
In[] := | << Graphics`ImplicitPlot`
```

Voici une équation cartésienne d'une courbe de Lissajoux $x = \sin t$, $y = \sin 4t$:

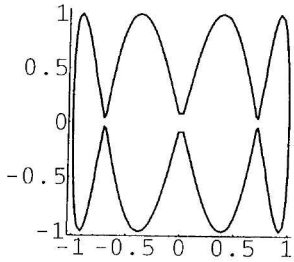
```
In[] := | eq = y^2 == 16 x^2 (1 - x^2) (2 x^2 - 1)^2
Out[] = | y^2 == 16 x^2 (1 - x^2) (-1 + 2 x^2)^2
```

En indiquant deux itérateurs, la fonction appelle `ContourPlot` avec l'option `Contours -> {0}`.

Le calcul est assez rapide, mais le résultat pas toujours très satisfaisant.

```
In[] := | ImplicitPlot[eq, {x, -1, 1}, {y, -1, 1}, PlotPoints -> 50]
```

-Figure 26-

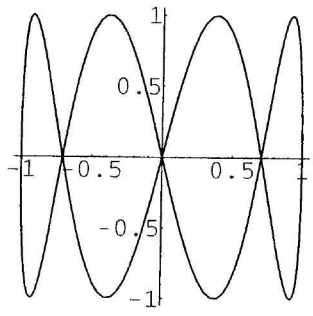



```
Out[] = | - ContourGraphics -
```

En indiquant un seul itérateur, une méthode à base de résolution d'équations est utilisée; le résultat est meilleur (la valeur par défaut de `PlotPoints`, à 25, suffit).

```
In[] := | ImplicitPlot[eq, {x, -1, 1}]
```

-Figure 27-




```
Out[] = | - Graphics -
```

Pour tracer des surfaces à partir d'une équation implicite $f(x, y, z) = 0$, on utilise la fonction `ContourPlot3D` du fichier `Graphics`ContourPlot3D``.

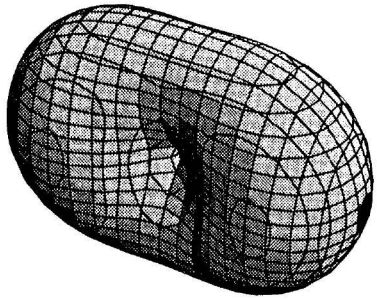
En fait cette fonction trace des surfaces de niveau dans l'espace; avec la valeur d'option : `Contours -> {0}` (valeur par défaut), on obtient une surface implicite.

Le nombre de points à calculer est fonction des valeurs combinées des options `MaxRecursion` et `PlotPoints`. Si la figure ci-dessous est trop longue à sortir, on peut réduire à `PlotPoints -> {3,7}`.

```
In[] := | << Graphics`ContourPlot3D`
```

```
In[] := ContourPlot3D[(x^2 + y^2 + z^2)^2 - 2 x^2 + y^2 - z^2,
          {x, -1.4, 1.4}, {y, -1, 1}, {z, -1, 1},
          PlotPoints -> {3, 10}, Boxed -> False];
```

-Figure 28-



3 - Combinaisons et animations de graphiques

3.1 - Conversion de graphiques

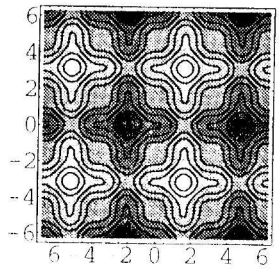
- Rappelons les différents types de graphiques produits par *Mathematica* :
  - **Graphics** pour les graphiques 2D construits à partir de primitives graphiques; cette structure sera étudiée plus en détails au chapitre 2.
  - **SurfaceGraphics**, **ContourGraphics**, **DensityGraphics** pour représenter des surfaces à partir d'une matrice de valeurs de cotes (altitudes).
  - **Graphics3D** pour les graphiques 3D construits à partir de primitives graphiques; cette structure sera étudiée plus en détails au chapitre 4.

Pour effectuer une conversion de type graphique, on passe l'objet graphique en argument de la fonction représentant le type que l'on désire obtenir.

L'objet `graph` ci-dessous, de type `ContourGraphics` est représenté par une simple matrice numérique :

```
In[] := graph = ContourPlot[Sin[x]^3 - Cos[y]^3, {x, -2 π, 2 π}, {y, -2 π, 2 π},
          PlotPoints -> 50];
```

Figure 29-

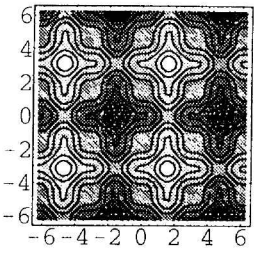



```
In[] := Short[graph[[1]], 6]

Out[] = {{-1., -0.98368, -0.881833, -0.663308, -0.37466, -0.118942, <<38>>,
          -1.88106, -1.62534, -1.33669, -1.11817, -1.01632, -1.},
         <<48>>, {-1., <<48>>, -1.}}
```

Nous le convertissons en un ensemble de primitives graphiques 2D :

```
In[] := | Show[Graphics[graph]]
```

-Figure 30-




```
Out[] = | - Graphics -
```

```
In[] := | Short[%[[1]], 10]

Out[] = | {{GrayLevel[0.5],
          Polygon[{{-6.28319, 6.28319}, {6.28319, 6.28319},
            {6.28319, -6.28319}, {-6.28319, -6.28319}}]}, {<<1>>},
         <<63>>, {GrayLevel[0.8], Polygon[<<1>>], {<<1>>}}}
```

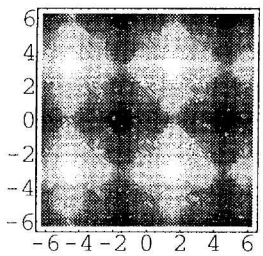
Le même graphique se convertit facilement dans le type `DensityGraphics`, très proche; sa traduction dans le type `Graphics` sera alors à base de primitive `Raster` représentant des cellules de niveaux de gris (voir chapitre 2).

```
In[] := | Show[DensityGraphics[graph], Mesh -> False]
```

(Figure non montrée)

```
In[] := | Show[Graphics[%]];
```

-Figure 31-

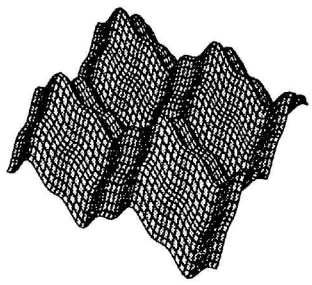



```
In[] := | Short[%[[1]], 10]

Out[] = | Raster[{<<1>>}, {{-6.28319, -6.28319}, {6.28319, 6.28319}},
          {-1.99846, 1.99231}]
```

Seul le type SurfaceGraphics peut être converti dans le type Graphics3D :

```
In[] := | Show[Graphics3D[SurfaceGraphics[graph]], Boxed -> False]
```

-Figure 32-



```
Out[] = | - Graphics3D -
```

```
In[] := Short[%[[1]], 7]

Out[] = {Polygon[{{-6.28319, -6.02673, -0.905058},
     {-6.28319, -6.28319, -1.}, {-6.02673, -6.28319, -0.98368},
     {-6.02673, -6.02673, -0.888738}}], <<2399>>,
   Polygon[{<<1>>, <<3>>}]}
```

- Pour résumer, voici les possibilités de conversions offertes :
  - Tous les types graphiques sont convertibles dans le type `Graphics`.
  - Les types `SurfaceGraphics`, `ContourGraphics`, `DensityGraphics` sont totalement interchangeables.
  - Le type `SurfaceGraphics` peut se convertir dans le type `Graphics3D`.

3.2 - Superposition de graphiques

Il y a plusieurs méthodes pour superposer des graphiques; nous en avons pratiqué quelques unes et aurons souvent l'occasion de les employer dans la suite; nous nous contentons de les passer rapidement en revue :

3.2.1 - Utilisation de la commande Show

La fonction `Show` permet de superposer un nombre quelconque de graphiques; une mise à l'échelle est faite, et si les graphiques sont de types différents, les conversions nécessaires sont effectuées.

3.2.2 - Utilisation des arguments de certaines fonctions graphiques

On sait que certaines fonctions graphiques permettent d'introduire plusieurs fonctions dont les représentations graphiques seront superposés. Parmi les fonctions que nous avons étudiées à la sous-section 2.2, nous avons ainsi : `Plot`, `ParametricPlot`, `ParametricPlot3D`, `ImplicitPlot`.

3.2.3 - Utilisation des options Epilog et Prolog

Ces options permettent d'introduire commodément des primitives (de dimension 2 seulement) à partir de n'importe quelle commande produisant des graphique.

Nous les utiliserons fréquemment dans les chapitres qui suivent.

3.3 - Tableaux de graphiques

3.3.1 - La fonction GraphicsArray

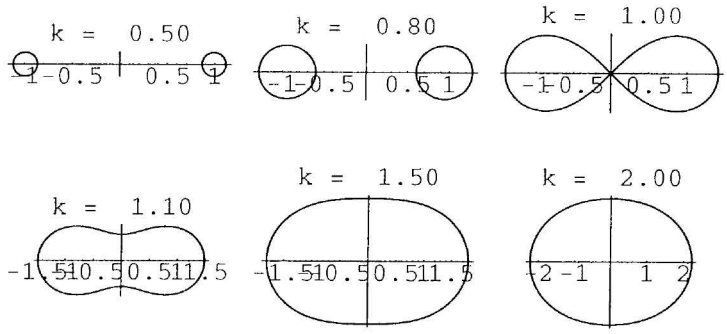
La fonction `GraphicsArray` prend en premier argument une liste (liste simple ou matrice) de graphiques de tous types que l'on peut ensuite tracer à l'aide de `Show`. L'objet produit est de type `GraphicsArray`.

Ci-contre nous traçons une famille de courbes dépendant d'un paramètre.

```
In[] := << Graphics`ImplicitPlot`

In[] := cassini[k_] := ImplicitPlot[(x^2 + y^2 + 1)^2 == 4 x^2 + k^4,
          {x, -Sqrt[1 + k^2], Sqrt[1 + k^2]},
          PlotLabel -> PaddedForm[SequenceForm["k = ", k], {3, 2}],
          AspectRatio -> Automatic, DisplayFunction -> Identity,
          Ticks -> {Automatic, None}, PlotPoints -> 50]
```

La fonction `PaddedForm` utilisée pour le titre du graphique sert à fixer le format des nombres (3 chiffres dont 2 décimales), ce qui sera utile pour l'animation proposée plus loin.

```
In[] := Show[GraphicsArray[{{cassini[0.5], cassini[0.8], cassini[1]},
          {cassini[1.1], cassini[1.5], cassini[2]}}], AspectRatio -> 0.5]
```



-Figure 33-

```
Out[] = - GraphicsArray -
```

Chaque graphique garde ses options propres, et les options passées dans `GraphicsArray` ou `Show` concernent le tableau dans son ensemble.

Notons qu'on peut imbriquer des tableaux de graphiques; le lecteur expérimentera lui-même.

3.3.2 - Utilisation de la primitive Rectangle

Pour l'utilisation de cette primitive, voir le chapitre suivant, sous-section 3.5.

3.4 - Animation de graphiques

Pour animer une séquence de graphiques, on commence par produire cette séquence; on utilise pour cela fréquemment une fonction d'itération telle que `Do` ou `Table`.

Ces figures sont placées dans des cellules consécutives, formant un film; ces cellules peuvent être groupées : elles sont alors superposées de telle sorte que seule la première est apparente.

On utilise ensuite les outils de la partie frontale pour effectuer l'animation : après avoir sélectionné le graphique ou le crochet de cellule, on choisit l'item `Animate Selected Graphics` (menu `Cell` en version 3), ou bien le raccourci-clavier correspondant, ou bien on double-clique simplement sur le graphique.

Un menu local en bas de la fenêtre permet d'effectuer différents réglages.

Le lecteur pourra essayer sur sa machine l'animation ci-dessous; il est important de fixer la même valeur de `PlotRange` pour toutes les images; de même le titre doit toujours avoir le même nombre de caractères pour éviter des "sauts".

```
In[] := Do[Show[cassini[k], DisplayFunction -> $DisplayFunction,
          PlotRange -> {{-2, 2}, {-1.5, 1.5}}],
        {k, 0.8, 1.6, 0.05}]
```

(Figures non montrées)

Notons qu'on peut animer toutes sortes de cellules, pas seulement des cellules graphiques, bien que cette possibilité soit plus rarement intéressante.

Le fichier `Graphics`Animation`` offre des utilitaires pour faciliter la création d'animations; la fonction `Animate` peut ainsi être utilisée à la place de `Do`; la fonction `ShowAnimation` permet d'animer une liste de graphiques préalablement construits. Notons également dans ce fichier la fonction `SpinShow` qui fait tourner le point de vue autour d'une figure de l'espace.

Nous construirons dans les chapitres suivants un certain nombres d'animations; le lecteur s'y reportera pour apprendre à résoudre les principaux problèmes qui se posent.

Réponses aux exercices

Exercice1

Cela dépend de l'ordre dans lequel sont placés les arguments `cercle` et `poire` (le premier seulement ayant la valeur d'option `Identity` pour `DisplayFunction`. On essaiera successivement les commandes :

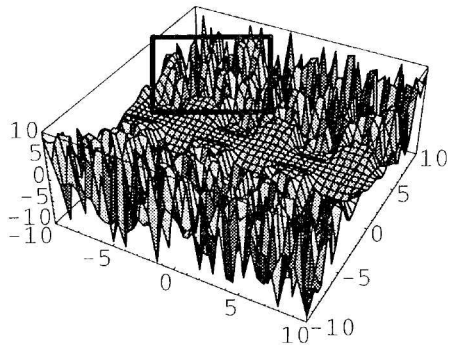
```
In[] := Show[cercle, poire]

In[] := Show[poire, cercle]
```

Exercice 2

Après avoir tracé le graphique et l'avoir sélectionné, on trace un cadre délimitant la zone à zoomer comme expliqué plus haut, et on la copie dans le presse-papier.

```
In[] := Plot3D[y Sin[y Sin[y Sin[x]]], {x, -10, 10}, {y, -10, 10},
         PlotPoints -> 40];
```

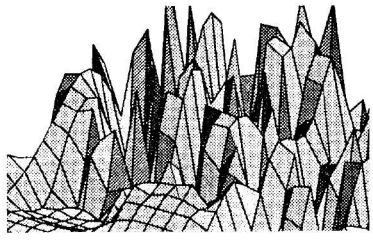
-Figure 34-



On fait retracer le graphique, mais en le convertissant en un graphique 2D (voir le chapitre 4 pour plus de détails sur les conversions de types dans les graphiques). Dans le graphique ainsi converti, les coordonnées utilisées sont les coordonnées écran; il suffit donc de coller la valeur à donner à `PlotRange` pour avoir le recadrage exact.

```
In[] := Show[Graphics[%],
         PlotRange -> {{0.267598, 0.571728}, {0.568162, 0.773119}}];
```

-Figure 35-



Exercice 3

Il suffit de représenter la fonction sur un intervalle qui recouvre les différents domaines.

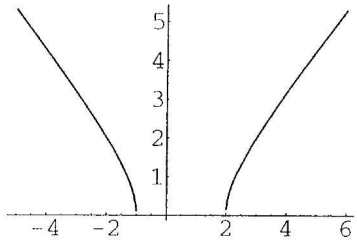
```
In[] := Plot[Sqrt[(x - 2) (x + 1)], {x, -5, 6}]
```

(Figure non montrée)

L'inconvénient est la production de messages; il suffit alors de les supprimer (après un premier essai, on sait tout de suite le nom du message à supprimer) :

```
In[] := Off[Plot::plnr]
        Plot[√((x - 2) (x + 1)), {x, -5, 6}];
```

-Figure 36-



On rétablit ensuite le message :

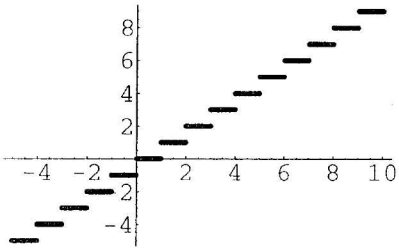
```
In[] := On[Plot::plnr]
```

Exercice 4

Une méthode consiste à tracer séparément chaque palier comme la courbe d'une fonction différente :

```
In[] := Clear[x];
        f = Table[If[k ≤ x < k + 1, Evaluate[k]], {k, -5, 10}];

In[] := Off[Plot::plnr]

In[] := Plot[Evaluate[f], {x, -5, 10},
         PlotStyle -> Thickness[1 / 75]];
```

-Figure 37-




```
In[] := On[Plot::plnr]
```

Notons la nécessité de forcer l'évaluation dans `If` car cette fonction est `HoldRest` et la condition ne peut s'évaluer tant que `x` n'est pas affecté.

```
In[] := k = 1; If[k ≤ x < k + 1, k]
Out[] = If[1 ≤ x < 2, k]
```

De même on doit forcer l'évaluation du premier argument de `Plot` pour y "faire entrer" la liste des expressions précédente, la fonction étant `HoldAll`.

Exercice 5

On trace un cercle en le coupant par des droites parallèles à la bissectrice des axes; les zones de la courbe voisines de la droite d'équation $y = -x$ ne seront pas suffisamment lissées.

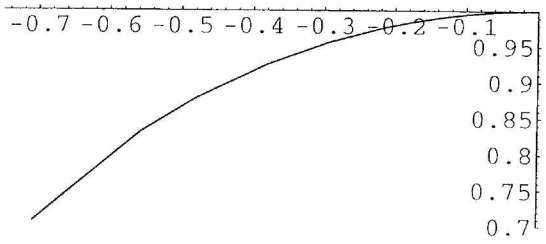
En grossissant la figure ci-dessous, on observe que les extrémités du demi-cercle ne sont pas bien arrondies.

```
In[] := fig1 = ParametricPlot[{Cos[ArcSin[λ] + π/4], Sin[ArcSin[λ] + π/4]},
         {λ, -1, 1}, AspectRatio -> Automatic];
```

(Figure non montrée)

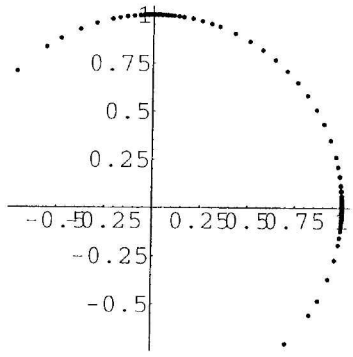
```
In[] := | Show[fig1, PlotRange -> {{-0.75, 0}, {0.7, 1}}];
```

-Figure 38-




```
In[] := | Show[fig1 /. Line[l_] :> Prepend[Point /@ l, PointSize[1 / 75]]];
```

-Figure 39-



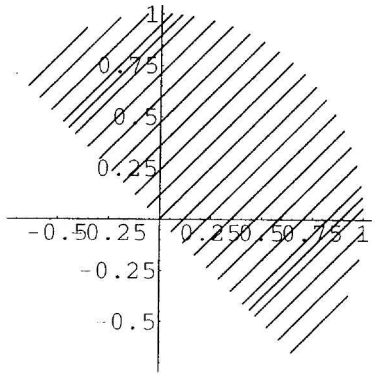
Le programme ci-dessous montre comment le paramétrage répartit des points d'échantillonnage sur la droite d'équation $y = -x$ (les coordonnées de la projection d'un point de $M(x, y)$ sur cette droite sont $(\frac{x-y}{2}, \frac{y-x}{2})$) :

```
In[] := | fig2 = ParametricPlot[{Cos[ArcSin[λ] + π/4], Sin[ArcSin[λ] + π/4]},
          {λ, -1, 1}, AspectRatio -> Automatic, MaxBend -> 45,
           PlotDivision -> 1];
```

(Figure non montrée)

```
In[] := | Show[fig2 /. Line[l_] :>
            Map[Line[{{(#[[1]] - #[[2]]) / 2, (#[[2]] - #[[1]]) / 2}, #}]&, l]];
```

-Figure 40-



Chapitre 2

Les graphiques en dimension 2

1 - Les primitives et directives graphiques 2D

1.1 La structure des graphiques 2D : le premier argument

Un objet graphique 2D est une expression de type `Graphics`, à deux arguments qui sont des listes :

```
In[] := | Graphics[{ < primitives et directives >}, { < options >}]
```

Les primitives graphiques sont des fonctions *Mathematica* qui représentent les objets graphiques de base.

Les directives graphiques sont des fonctions *Mathematica* qui agissent sur les primitives pour en modifier certaines caractéristiques (l'épaisseur d'une ligne, la grosseur d'un point, la couleur etc.).

Voici quelques principes régissant l'action des directives graphiques sur les primitives :

– Certaines directives n'agissent que sur des primitives déterminées (`PointSize` n'agit que sur la primitive `Point`, `Thickness` sur les primitives `Line`, `Circle` etc.), tandis que d'autres agissent sur tous les types de primitives (notamment les directives de couleurs).

– Une directive n'agit que sur les primitives placées dans la même liste ou dans des sous-listes.

– Une directive agit sur toutes les primitives placées après elle dans la liste où elle figure, mais son effet est annulé par la présence de la même directive placée après elle dans cette liste.

Dans l'objet graphique ci-dessous, le premier point est de grosseur 1/50 et le second de grosseur 1/75. Seuls le premier point et la ligne seront rouges (la couleur par défaut est noire) :

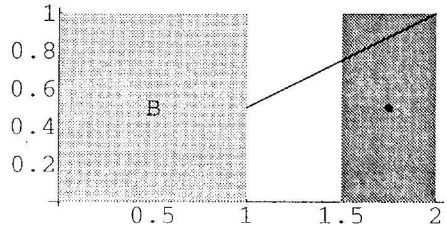
```
In[] := | Graphics[{PointSize[1/50],
            {RGBColor[1, 0, 0], Point[{0, 0}], Line[{{0, 0}, {1, 1}}]},
          PointSize[1/75], Point[{1, 1}]}] // Show
```

(Figure non montrée)

Les primitives graphiques sont "opaques" : celles placées après dans la liste cachent les précédentes. Notons également que les axes sont tracés en premier.

```
In[] := | Graphics[{PointSize[1/50], {GrayLevel[0.5],
            Rectangle[{1.5, 0}, {2, 1}]}, Text["A", {1/2, 1/2}],
          Point[{1.75, 0.5}], Line[{{0, 0}, {2, 1}}],
          {GrayLevel[0.8], Polygon[{{0, 0}, {1, 0}, {1, 1}, {0, 1}}]},
        Text["B", {1/2, 1/2}]}, Axes -> True,
          AspectRatio -> Automatic] // Show
```

-Figure 1-




```
Out[] = | - Graphics -
```

1.2 Les primitives graphiques 2D

Voici les primitives graphiques en dimension 2 :

- **Point :** `Point[{x, y}]` représente le point de coordonnées $\{x, y\}$; il est dessiné comme une petite tache.
- **Line :** `Line[{{x1, y1}, {x2, y2}, ..., {xn, yn}}]` représente une ligne brisée joignant les points $\{x_k, y_k\}$.
- **Circle :** `Circle[{x, y}, r]` représente un cercle de centre $\{x, y\}$, de rayon r; `Circle[{x, y}, {a, b}]` représente une ellipse de demi-axes a, b; `Circle[{x, y}, {a, b}, {θ1, θ2}]` représente un arc d'ellipse.
- **Disk :** Représente un disque plein; syntaxe analogue à celle de `Circle`.
- **Rectangle :** `Rectangle[{xmin, xmax}, {ymin, ymax}]` représente un rectangle plein à côtés parallèles aux axes. `Rectangle[{xmin, xmax}, {ymin, ymax}, graph]` représente un rectangle rempli par le graphique `graph` (de n'importe quel type); cela permet d'introduire des graphiques à l'intérieur d'autres graphiques.
- **Polygon :** `Polygon[{{x1, y1}, {x2, y2}, ..., {xn, yn}}]` représente un polygone plein (le dernier sommet est joint au premier).
- **Raster, RasterArray :** `Raster[`*matrice*`]` représente un tableau de cellules colorées, les éléments de la matrice indiquant la densité de couleur; la fonction de coloration est indiquée par l'option `ColorFunction`. `RasterArray[`*matrice*`]` représente aussi un tableau de cellules colorées, les éléments de la matrice étant des valeurs de directives de couleurs.
- **Text :** `Text["`*texte*`",{x, y}]` représente la chaîne de caractères "*texte*" centrée au point de coordonnées $\{x, y\}$; `Text["`*texte*`",{x, y}, {dx, dy}]` décale la chaîne de{dx, dy} ("offset"). Des options (dont `TextStyle`) permettent de choisir le style du texte.
- **Postscript :** Cette primitive permet d'introduire du code *PostScript* dans le graphique; on peut s'en servir pour créer de nouvelles primitives graphiques.

1.3 Les directives graphiques 2D

Les directives graphiques peuvent se classer en deux catégories :

Les directives servant à spécifier des dimensions. Elles se classent elle-même en deux catégories :

– les directives "relatives" : l'unité employée est la plus grande dimension du graphique; ces dimensions varieront donc avec celles du graphique

– les directives "absolues" : l'unité est le "point d'imprimante" (environ $\frac{1}{72}$de pouce, soit 0, 35 *mm*)

- **PointSize, AbsolutePointSize :** Précise la grosseur d'un point (essayer `PointSize[1/50]`, `PointSize[1/100]`).
- **Thickness, AbsoluteThickness :** Précise l'épaisseur d'un trait pour `Line` ou `Circle` (`Thickness[1/1000]` trace un filet).
- **Dashing, AbsoluteDashing :** Permet de tracer des lignes pointillées; `Dashing[{d1, d2, ..., dn}]` trace alternativement un trait, un blanc, un trait, ... de largeurs d_1, d_2, ...

Les directives servant à spécifier des couleurs :

- **GrayLevel :** Niveau de gris : `GrayLevel[s]` va du noir (pour $s = 0$) au blanc (pour $s = 1$).
- **RGBColor :** Couleur par chacune des 3 composantes fondamentales : `RGBColor[r, g, b]` où r dose le rouge, g le vert et b le bleu (valeurs entre 0 et 1). `RGBColor[1, 1, 1]` reconstitue le blanc.
- **Hue :** Couleur à partir de sa fréquence : `Hue[t]` représente les couleurs du spectre lorsque t croît de 0 à 1; pour les valeurs de t hors de l'intervalle [0,1], l'effet est répété périodiquement. On a deux autres arguments optionnels : `Hue[t, s, l]` où t, s, l indiquent la teinte, la saturation et la luminosité.
- **CMYKColor :** Couleur par ses composantes quadrichromiques (cyan, magenta, jaune, noir).

2 - Les options graphiques 2D

2.1 La structure des graphiques 2D : le second argument

Dans un objet graphique 2D :

```
In[] := | Graphics[{ < primitives et directives >}, { < options >}]
```

le second argument est une liste de règles de la forme : *symbole* -> *valeur*, chaque symbole étant le nom d'un argument "optionnel par nom" de la fonction `Graphics`.

Les valeurs de ces options précisent la façon dont le graphique (défini par les primitives et directives du premier argument) doivent être représentés.

Lorsqu'on construit un graphique, on n'indique à la fonction `Graphics` uniquement les options dont on désire changer la valeur par rapport à la valeur par défaut. Conformément aux règles de syntaxe des fonctions comportant des options, il est inutile de les regrouper dans une liste.

Ci-dessous nous traçons un cercle "rond" en précisant un "repère orthonormé", et nous demandons de montrer tous les éléments du graphique :

```
In[] := | gr = Graphics[{PointSize[1 / 50], Point[{0, 0}],
          Point[{2, 2}], Line[{{0, 0}, {2, 2}}], RGBColor[1, 0, 0],
          Circle[{0, 0}, 3 / 2]}, AspectRatio -> Automatic,
          PlotRange -> All] // Show
```

(Figure non montrée)

Dans l'objet graphique final, l'ensemble des options constitue bien le second argument :

```
In[] := | gr[[2]]
Out[] = | {AspectRatio → Automatic, PlotRange → All}
```

La commande `Options[gr]` donnerait le même résultat.

Remarquons que selon les versions de *Mathematica* (ici 3.0), toutes les valeurs des options, ou seulement celles qui sont modifiées, sont montrées; les valeurs des autres options sont les valeurs par défaut, que l'on obtient par la commande `Options[Graphics]`. Pour obtenir les valeurs effectives de toutes les options associées à la fonction `Graphics` et correspondant à notre objet graphique, on utilise la fonction `FullOptions` :

```
In[] := | Short[FullOptions[gr], 5]
Out[] = | {AspectRatio → 1.,
          Axes → {False, False}, AxesLabel → None, AxesOrigin → {0., 0.},
          AxesStyle → {None, None}, <<16>>, DefaultFont → {Courier, 10.},
          DisplayFunction → (Display[$Display, #1]&),
          FormatType → StandardForm, TextStyle → {}}
```

Pour modifire les options d'un graphique, il est inutile de recomposer ce dernier; on utilise la fonction `Show` :

```
In[] := | Options[
          Show[gr, AspectRatio -> 1 / 2, Background -> RGBColor[0, 1, 1]]];
```

(Figure non montrée)

```
Out[] = | {AspectRatio → 1/2, Background → RGBColor[0, 1, 1],
          AspectRatio → Automatic, PlotRange → All}
```

2.2 Description de quelques options graphiques 2D

Les options associées à la fonction `Graphics` sont nombreuses :

```
In[] := Length[Options[Graphics]]
Out[] = 25
```

et certaines sont rarement modifiées; d'autres sont d'un maniement plus délicat et nous les aborderons plus tard à l'occasion d'exemples.

Nous allons passer en revue celles qu'il est indispensable de connaître dans un premier temps.

```
In[] := Options[Graphics]
Out[] = {AspectRatio → 1/GoldenRatio, Axes → False,
  AxesLabel → None, AxesOrigin → Automatic, AxesStyle → Automatic,
  Background → Automatic, ColorOutput → Automatic,
  DefaultColor → Automatic, Epilog → {}, Frame → False,
  FrameLabel → None, FrameStyle → Automatic, FrameTicks → Automatic,
  GridLines → None, ImageSize → Automatic, PlotLabel → None,
  PlotRange → Automatic, PlotRegion → Automatic, Prolog → {},
  RotateLabel → True, Ticks → Automatic, DefaultFont :→ $DefaultFont,
  DisplayFunction :→ $DisplayFunction, FormatType :→ $FormatType,
  TextStyle :→ $TextStyle}
```

- **DisplayFunction** permet de préciser la sortie sur laquelle envoyer le graphique. La valeur par défaut `$DisplayFunction` envoie la figure vers l'écran de la machine; en modifiant la valeur de cette variable globale (exprimée en termes de la fonction `Display`), on peut envoyer directement le graphique dans un fichier. La valeur `Identity` a pour effet de ne pas produire de sortie graphique.

- **AspectRatio** précise les proportions du rectangle contenant le graphique (rapport $\frac{\text{hauteur}}{\text{largeur}}$). La valeur par défaut donne un "rectangle d'or" (ce rapport valant environ 0.618). Si on donne comme valeur le symbole `Automatic`, les proportions réelles du graphique sont respectées (repère orthonormé).

- **PlotRange** précise la zone de graphique à représenter. La valeur `Automatic` met en oeuvre un algorithme qui "cadre" au mieux la partie la "plus intéressante" du graphique (utile pour des représentations de fonctions mathématiques). La valeur `All` représente tout le graphique (utile si on construit un graphique par programme). On peut enfin préciser des valeurs par des listes : *{ymin,ymax}* ou *{{xmin,xmax},{ymin,ymax}}*.

- **PlotLabel**, **Background**, **GridLines** précisent respectivement un titre pour le graphique, une couleur de fond, un quadrillage (la valeur de cette dernière option est assez compliquée, mais on peut toujours se contenter de mettre `Automatic`). Remarquons que si on laisse la valeur de l'option `DefaultColor` à `Automatic`, la couleur du tracé est complémentaire de celle du fond; essayer un fond noir ou bleu (`RGBColor[0,0,1]`) ou jaune (`RGBColor[1,1,0]`).

- **Axes**, **AxesLabel**, **AxesOrigin**, **AxesStyle**, **Ticks** configurent les axes de coordonnées. Les 4 dernières n'ont d'effet que si la première a pour valeur `True`. On peut par exemple écrire : `AxesLabel --> {"x","y"}`; les valeurs de `Ticks` sont assez compliquées; on peut utiliser les fonctions toutes faites `UnitScale`, `PiScale` du fichier `Graphics.m`.

- **Frame**, **FrameLabel**, **FrameStyle**, **FrameTicks** configurent un cadre autour du graphique à la place des axes de coordonnées.

- **Prolog**, **Epilog** permettent d'introduire des primitives graphiques supplémentaires: ces primitives sont rendues avant le graphique proprement dit avec la première option, après avec la seconde. Elles sont très utiles pour "habiller" un graphique produit à partir d'une commande graphique comme `Plot`, `ListPlot` etc.

3 - Compléter des graphiques produits par les commandes usuelles

3.1 - Utilisation des options

L'utilisation des options permet d'améliorer la présentation des graphiques.

Certaines sont d'un usage constant; par exemple si on choisit : `AspectRatio -> Automatic` et qu'on laisse la valeur de `PlotRange` à `Automatic`, on a de bonnes chances d'obtenir un graphique disproportionné; il faudra alors ajuster "à la main" la valeur de cette dernière option.

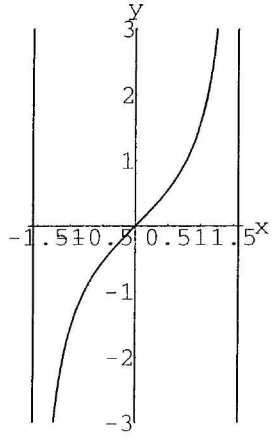
```
In[] := Plot[Tan[x], {x, -π/2, π/2}, AspectRatio -> Automatic];
```

(Figure non montrée)

Dans cet exemple, *Mathematica* a coupé le graphique à $-100 \leq y \leq 100$, ce qui ne convient pas. Nous introduisons ci-dessous quelques options supplémentaires (noms pour les axes, asymptotes verticales) :

```
In[] := Plot[Tan[x], {x, -π/2, π/2}, AspectRatio -> Automatic,
         PlotRange -> {-4, 4}, AxesLabel -> {"x", "y"}, Epilog ->
          {Line[{{-π/2, 4}, {-π/2, -4}}], Line[{{π/2, 4}, {π/2, -4}}]}];
```

-Figure 2-



3.2 - Utilisation de l'option PlotStyle

L'option `PlotStyle`, disponible avec les fonctions standards `Plot`, `ParametricPlot`, `ListPlot` (et des fonction analogues du fichier `Graphics.m`) permet d'introduire des directives graphiques qui agiront sur les courbes tracées; nous l'avons rencontrée au chapitre 1.

Le principe de cette option est de rajouter dans la liste premier argument de l'objet graphique, les directives en question; on peut aussi se servir de cette option pour introduire des primitives graphiques (encore qu'il soit préférable d'utiliser dans ce cas les options `Prolog` et `Epilog`).

Pour la syntaxe relative à l'option `PlotStyle`, deux cas à distinguer :

- **Un seul graphique est tracé :**

  On peut donner comme valeur à `PlotStyle` une primitive ou directive graphique, ou bien une liste de primitives ou directives.

```
In[] := ListPlot[Table[{Random[], Random[]}, {100}],
        Frame -> True, FrameTicks -> None, AspectRatio -> Automatic,
         FrameStyle -> {Thickness[1/50], RGBColor[0, 0, 1]},
        PlotStyle -> {PointSize[1/100], RGBColor[1, 0, 0]}];
```

(Figure non montrée)

```
In[] := Plot[Sin[x], {x, -2 π, 2 π}, AspectRatio -> Automatic,
         PlotStyle -> {Text["O", {0, 0}, {-2, 2}], RGBColor[1, 0, 0]}];
```

(Figure non montrée)

Contrairement aux options `Prolog` et `Epilog`, les primitives introduites par `PlotStyle` ne pourront plus être modifiées si on retrace le graphique par `Show`.

- **Plusieurs graphiques sont superposés (fonctions Plot et ParametricPlot seulement) :**

On doit passer comme valeur à `PlotStyle` une liste de même longueur que la liste de fonctions à tracer constituant le premier membre de la commande.

Chaque élément de la liste est une primitive ou directive graphique, ou bien une liste de primitives ou directives (on met la liste vide { } pour un graphique à ne pas modifier).

```
In[] := ParametricPlot[
         Evaluate[Table[{Sin[(k - 1) t], Sin[k t]}, {k, 2, 10, 2}]],
         {t, 0, 2 Pi}, PlotStyle ->
          Table[{Hue[k/10], Thickness[1/100]}, {k, 2, 10, 2}],
         AspectRatio -> Automatic, Axes -> False];
```

(Figure non montrée)

3.3 - Tracés de flèches

Donnons un exemple d'utilisation des options pour "habiller" un graphique dans le style "scolaire". On utilise le fichier `Arrow.m` qui produit des primitives graphiques supplémentaires permettant de dessiner des flèches.

```
In[] := << Graphics`Arrow`
```

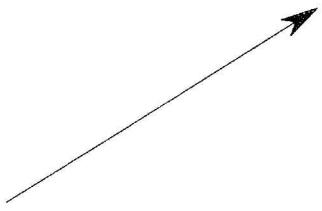
La fonction `Arrow` possède deux arguments (point de départ, point d'arrivée muni de la flèche), et un certain nombre d'options pour régler la forme de cette flèche.

En donnant à `HeadScaling` la valeur `Absolute` (resp. `Relative`), la longueur de la tête de flèche (`HeadLength`) sera exprimée en points d'imprimante (resp. avec la longueur de la flèche comme unité).

Le centre de la flèche (`HeadCenter`) est compté à partir de la pointe, avec comme unité la longueur de la tête de flèche. Le lecteur modifiera les valeurs dans la commande suivante pour comprendre :

```
In[] := Show[Graphics[{Arrow[{0, 0}, {1, 1}, HeadScaling -> Relative,
            HeadLength -> 1/10, HeadCenter -> 2/3]}]];
```

-Figure 3-



La fonction `contact` ci-dessous contruit un objet graphique "élément de contact" : `c = {x,y}` est le point de contact, `dir = {x',y'}` la direction et `r` le rayon; le lecteur peut modifier les caractéristiques de cet objet, ou introduire des options pour les contrôler.

```
In[] := contact[c_, dir_, r_] := With[{v = r / Sqrt[dir . dir] dir},
          {{AbsolutePointSize[2], Point[c]},
           Arrow[c, c + v, HeadScaling -> Absolute,
            HeadLength -> 5, HeadCenter -> 1 / 3],
           Arrow[c, c - v, HeadScaling -> Absolute, HeadLength -> 5,
            HeadCenter -> 1 / 3]}]
```

```
In[] := x[t_] = t / ((1 + t^2) (1 + 4 t));
        y[t_] = t^2 / ((1 + t^2) (1 + 4 t));
```

```
In[] := xp[t_] = Together[x'[t]]
Out[] = (1 - t^2 - 8 t^3) / ((1 + 4 t)^2 (1 + t^2)^2)
```

```
In[] := yp[t_] = Together[y'[t]]
Out[] = -(2 (-t - 2 t^2 + 2 t^4) / ((1 + 4 t)^2 (1 + t^2)^2))
```

Nous récupérons les points qui annulent une dérivée et construisons l'élément de contact correspondant (la valeur du rayon s'obtient par tâtonnements, après un premier tracé du graphique) :

```
In[] := NSolve[xp[t] == 0]
Out[] = {{t → -0.293307 - 0.429837 I}, {t → -0.293307 + 0.429837 I},
         {t → 0.461614}}
```

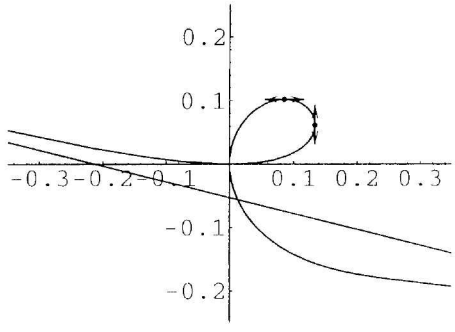
```
In[] := c1 = contact[{x[t], y[t]}, {xp[t], yp[t]}, 0.03] /. Last[%];
```

```
In[] := NSolve[yp[t] == 0]
Out[] = {{t → -0.595744 - 0.254426 I}, {t → -0.595744 + 0.254426 I},
         {t → 0.}, {t → 1.19149}}
```

```
In[] := c2 = contact[{x[t], y[t]}, {xp[t], yp[t]}, 0.03] /. Last[%];
```

```
In[] := ParametricPlot[{x[t], y[t]}, {t, -20, 20},
         AspectRatio -> Automatic,
         PlotRange -> {{-0.35, 0.35}, {-0.25, 0.25}},
        Epilog -> {c1, c2} ];
```

-Figure 4-



L'asymptote oblique s'est construite toute seule; mais si nous voulons obtenir son équation, nous pouvons procéder comme ci-dessous :

```
In[] := Series[x[t], {t, -1/4, 0}]
```

$$\text{Out[]} = -\frac{1}{17\,(t+\frac{1}{4})} + \frac{60}{289} + O\left[t+\frac{1}{4}\right]^1$$

```
In[] := xas = Normal[%]
```

$$\text{Out[]} = \frac{60}{289} - \frac{1}{17\,(\frac{1}{4}+t)}$$

```
In[] := yas = Normal[Series[y[t], {t, -1/4, 0}]]
```

$$\text{Out[]} = -\frac{32}{289} + \frac{1}{68\,(\frac{1}{4}+t)}$$

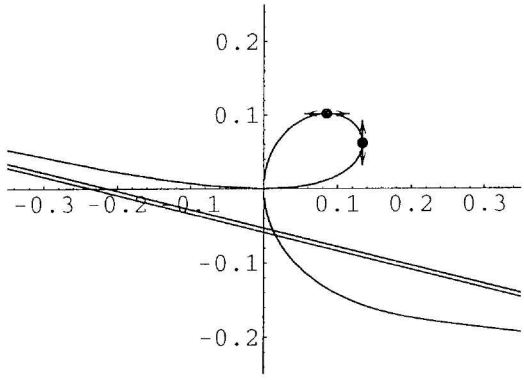
L'équation canonique s'obtient en résolvant par rapport à y et en éliminant t :

```
In[] := y /. Solve[{x == xas, y == yas}, y, t][[1]] // Expand
```

$$\text{Out[]} = -\frac{1}{17} - \frac{x}{4}$$

```
In[] := ParametricPlot[{{x[t], y[t]}, {xas, yas}},
         {t, -20, 20}, AspectRatio -> Automatic,
         PlotRange -> {{-0.35, 0.35}, {-0.25, 0.25}},
        Epilog -> {c1, c2} ];
```

-Figure 5-



- Exercice 1 :

  On voit que la droite tracée directement par *Mathematica* est fausse; nous proposons au lecteur d'expliquer pourquoi et de la supprimer.

- Exercice 2 :

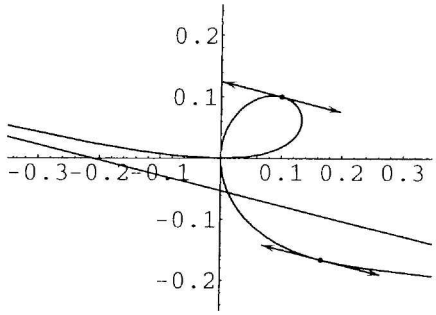
  Plus généralement, écrire un programme qui supprime dans un graphique, toutes les "fausses asymptotes" obtenues lorsque deux points consécutifs se trouvent à l'opposé dans le graphique..

Nous faisons ci-dessous une recherche des tangentes à la courbe parallèles à l'assymptote :

```
In[] := NSolve[xp[t] == -4 yp[t]]
Out[] = {{t → -1.}, {t → 1.}}

In[] := c3 = contact[{x[t], y[t]}, {xp[t], yp[t]}, 0.1] /. %;
```

```
In[] := ParametricPlot[{x[t], y[t]}, {t, -20, 20},
         AspectRatio -> Automatic,
         PlotRange -> {{-0.35, 0.35}, {-0.25, 0.25}},
        Epilog -> c3 ];
```

-Figure 6-



3.4 - Introduction de texte dans les graphiques

Les indications ci-dessous seront aussi valables pour les graphiques 3D, le texte étant de toutes façons un dessin plan.

1) - Le choix des polices et du style pour les graphiques a été modifié de la version 2 à la version 3.

Nous allons expliquer les principes de base dans les deux cas. Ces choix peuvent se faire à trois niveaux :

- Au niveau global pour tous les graphiques à tracer : on règle pour cela les valeurs de variables globales, qui sont les valeurs par défaut des options graphiques concernées. Ces variables sont :

  — Pour la version 3, `$FormatType` (pour l'option `FormatType`), et `$TextStyle` (pour l'option `TextStyle`).

  — Pour la version 2, `$DefaultFont` (pour l'option `DefaultFont`).

- Au niveau d'un graphique, à l'aide d'options graphiques (tous les textes introduits par l'utilisateur, y compris les graduations des axes etc.) :

  — Pour la version 3, `FormatType` dont les valeurs peuvent être `InputForm`, `OutputForm`, `StandardForm`, `TraditionalForm` etc. et `TextStyle`, acceptant comme valeurs, soit un style de texte (à l'aide de `StyleForm`) soit des spécifications pour les fontes (polices)

  — Pour la version 2, `DefaultFont`, qui prend comme valeur une liste {"police", taille} (par exemple {"Times",10}).

- Au niveau de la primitive `Text` (ne concernera que le bloc de texte introduit par cette primitive) :

  — Pour la version 3, la fonction `Text` accepte les mêmes options `FormatType` et `TextStyle` ci-dessus.

  — Pour la version 2, on utilise la fonction `FontForm`; on écrira par exemple : `Text[FontForm["toto",{"Times",10}],{2,-1}]`.

2) - La fonction `Text` possède quatre arguments, les deux derniers étant optionnels, et des options :

```
In[] := Text[ expr, coords, offset:{0,0}, dir:{1,0}, options ]
```

Dans les graphiques 2D, cette primitive est rendue comme toute autre primitive 2D, en fonction de sa position dans le liste des primitives où elle figure.

Dans les graphiques 3D, où la position relative des objets dépend de leurs coordonnées relatives et non de leur place dans la liste des primitives, elle est rendue en dernier, donc est toujours visible.

Enfin on peut affecter le rendu d'un objet graphique `Text` par une des directives graphiques suivantes : `GrayLevel`, `RGBColor`, `Hue`, `CMYKColor`.

Nous allons étudier la syntaxe et le rôle de chacun de ces arguments :

- **Le premier argument de la primitive Text :**

Il indique l'expression à rendre. Ce peut être une chaine de caractères, mais aussi n'importe quelle expression Mathematica. On peut préciser le format de l'expression, et aussi donner au texte un style prédéfini à l'aide de `StyleForm`.

```
In[] := Show[ Graphics[{Text[StyleForm["Voici une formule :",
            Section],{0,1}],
        Text[TraditionalForm[Cos[x]^2 == 1/(1 + Tan[x]^2)],{0,0}]},
        AspectRatio -> 1/4, PlotRange -> All] ];
```

Voici une formule :

$$\cos^2(x) == \frac{1}{\tan^2(x) + 1}$$

-Figure 7-

```
In[] := Show[Graphics[{Text[ColumnForm[{
            StyleForm["Voici une formule :",Section],
            TraditionalForm[Cos[x]^2 == 1/(1 + Tan[x]^2)]
            },Center],{0,0}]},AspectRatio -> 1/4, PlotRange -> All] ];
```

Voici une formule :

$$\cos^2(x) == \frac{1}{\tan^2(x)+1}$$

-Figure 8-

- **Le second argument de la primitive Text :**

Il indique la position dans le graphique du centre de l'expression à rendre. Si nous appellons *boite de texte* un rectangle imaginaire encadrant au plus près le texte à rendre; il s'agit des coordonnées du centre de cette boite de texte. Ces coordonnées peuvent être:

— Des coordonnées absolues (ou utilisateur): `{x,y}` en 2D (et `{x,y,z}` en 3D).

— Des coordonnées d'affichage (par rapport au rectangle, ou à la boite contenant le graphique): `Scaled[{dx,dy}]` en 2D et `Scaled[{dx,dy,dz}]` en 3D). Les coordonnées doivent être comprises entre 0 et 1.

```
In[] := Show[ Graphics[{
            Text[StyleForm["Voici une formule :",
            Section],Scaled[{1/2,3/4}]],
            Text[TraditionalForm[Cos[x]^2 == 1/(1 +
        Tan[x]^2)],Scaled[{1/2,1/4}]]
            },AspectRatio -> 1/4, PlotRange -> All] ];
```

Voici une formule :

$$\cos^2(x) == \frac{1}{\tan^2(x) + 1}$$

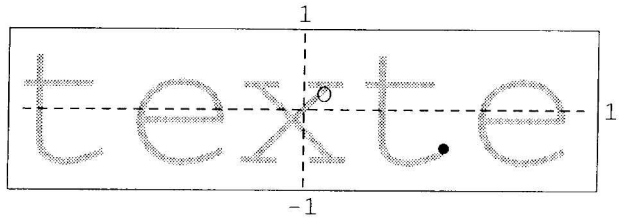
-Figure 9-

L'utilisation de `Scaled` permet de contrôler et de fixer la position des blocs de texte dans le rectangle $[0, 1]\times[0, 1]$.

- **Le troisième argument de la primitive Text :**

  Cet argument optionnel indique le décalage *(offset)* de la boite de texte par rapport au point spécifié dans le second argument. Il s'agit d'une liste de deux coordonnées `{dx,dy}` qui indique les coordonnées du point `{x,y}` du graphique spécifié par le second argument, relativement à un repère centré sur la boite de texte comme le montre le shéma suivant :

-Figure 10-



Ci-dessus, O est le centre de la boite de texte et la petite tache montre le point de coordonnées (x,y) (ses coordonnées dans le repère lié à la boite sont (0.5,-0.5)). La commande correspondante est :

```
In[] := | Text["texte", {x, y}, {0.5, -0.5}]
```

On peut utiliser des valeurs dx et dy extérieures à l'intervalle [-1,1] pour placer le point à l'extérieur de la boite de texte.

- **Le quatrième argument de la primitive Text :**

  Cet argument optionnel est utilisé pour effectuer une rotation de la boite de texte.

  La syntaxe est: `{a,b}` où {a,b} désignent les composantes d'un vecteur déterminant la direction de la ligne de texte (ou de l'axe de la boite de texte).

  Exemples de valeurs possibles:

  — {1,0} pour un texte horizontal orienté gauche-droite (valeur par défaut).

  — {-1,0} pour un texte horizontal orienté droite-gauche.

  — {0,1} pour un texte vertical orienté bas-haut.

  — {0,-1} pour un texte vertical orienté haut-bas.

  Il s'agit de rotations de la boite de texte, non de symétries: par exemple {-1,0} indique une symétrie centrale: le texte est renversé.

```
In[] := | Show[ Graphics[{Text["ceci est une ligne de texte",{0,0},
            {0,0},{1,-1/3}]}, Frame -> True, FrameTicks -> None] ];
```

-Figure 11-

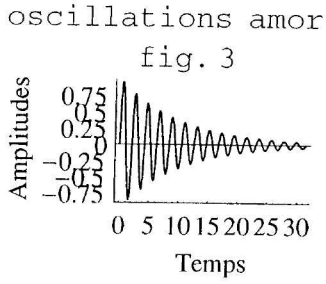
- **Les options de la primitive Text (version 3) :**

  Nous les avons signalées plus haut.

  D'abord un premier exemple où on manipule l'option `TextStyle` et la fonction `StyleForm` :

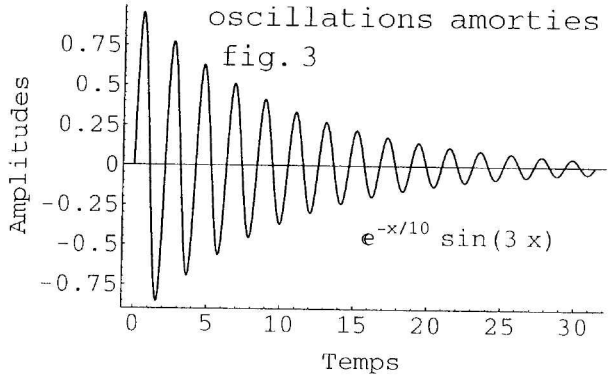
```
In[] := Plot[E^(-x/10) Sin[3x],{x,0,10Pi},
            PlotPoints -> 100,
            PlotLabel -> StyleForm["oscillations amorties\n
                fig. 3",FontFamily -> "Time-Bold", FontSize -> 12],
            TextStyle -> {FontFamily -> "Times",FontSize -> 10},
            Frame -> {True, True, False, False},
            FrameLabel -> {"Temps","Amplitudes"}  ];
```

-Figure 12-




```
In[] := Plot[E^(-x/10) Sin[3x],{x,0,10Pi},
            PlotPoints -> 100,
            Epilog -> {Text["oscillations amorties\n fig.
        3",Scaled[{0.6,0.9}],TextStyle -> {FontFamily -> "Time-Bold",
        FontSize -> 12}],
            Text[E^(-x/10) Sin[3x],Scaled[{0.7,0.25}],FormatType ->
        TraditionalForm]},
            Frame -> {True, True, False, False},
            FrameLabel -> {"Temps","Amplitudes"}
        ];
```

-Figure 13-



Exercice 3 :

Programmer une fonction triangle[{s1,s2,s3},{a1,a2,a3}] qui représente un triangle de sommets de coordonnées s1, s2, s3, de noms a1, a2, a3 placés automatiquement à l'extérieur du triangle.

3.5 - Animation de graphiques composés

3.5.1 - Tangentes à une courbe

Ci-dessous, nous proposons d'animer un ensemble de tangentes à une parabole. Notons que la valeur de l'option `Epilog` doit être une liste de primitives, non un objet de type `Graphics`, soit ici `parabole[[1]]`.

```
In[] := parabole = Plot[x^2, {x, -3, 3}, AspectRatio -> Automatic,
          PlotStyle -> RGBColor[1, 0, 0]];
```

(Figure non montrée)

```
In[] := Animate[Plot[2 t x - t^2, {x, -3, 3},
          AspectRatio -> Automatic, Epilog -> parabole[[1]]],
        {t, -3, 3, 0.5}, PlotRange -> {{-3, 3}, {-1, 9}}]
```

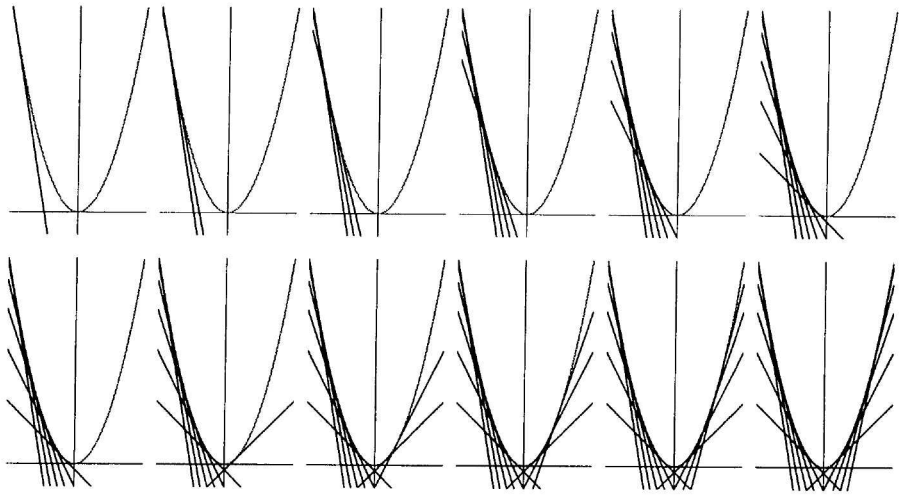
(Figures non montrées)

Nous voulons cumuler les tangentes dans l'animation; nous commençons par créer une liste de graphiques pour chaque tangente; ensuite nous appliquons la fonction `Show` à l'aide de `FoldList`; le double slot ## dans la fonction pure vaut pour une séquence quelconque d'arguments.

```
In[] := Table[Plot[2 t x - t^2, {x, -3, 3},
          DisplayFunction -> Identity, AspectRatio -> Automatic],
        {t, -3, 2.5, 0.5}];
```

```
In[] := Rest[FoldList[
          Show[##, PlotRange -> {{-3, 3}, {-1, 9}}, Ticks -> None]&,
          parabole, %]];
```

L'animation est représentée ici par un tableau de graphiques (voir aussi page suivante l'animation représentée par la figure 16).



-Figure 14-

3.5.2 - Cercle osculateur à une courbe

Nous nous proposons d'illustrer à l'aide d'une animation la génération de la développée d'une courbe plane par les centres de ses cercles osculateurs. La fonction `m` paramétrise une cardioïde :

```
In[] := norme[u_] := √(u . u)
```

```
In[] := Clear[x, y, t, m, c];
```

```
In[] := m[t_] = {(1 + Cos[t]) Cos[t], (1 + Cos[t]) Sin[t]}
```

```
In[] := cardioide = ParametricPlot[Evaluate[m[t]], {t, 0, 2 π},
          PlotStyle → RGBColor[1, 0, 0], AspectRatio -> Automatic];
```

(Figure non montrée)

• Ecrivons une équation de la normale au point de paramètre t :

```
In[] := normale = ({x, y} - m[t]) . m'[t] == 0;
```

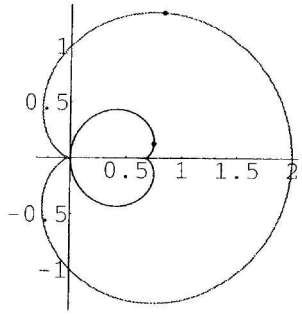
• Puis la recherche d'un système d'équations paramétriques pour la développée; le système suivant définit les coordonnées {x,y} du point caractéristique que nous notons `c` :

```
In[] := Solve[{normale, ∂t normale}, {x, y}];

In[] := c[t_] = Simplify[{x, y} /. %[[1]]]

Out[] = {1/6 (3 + 2 Cos[t] - Cos[2 t]), 1/6 (2 Sin[t] - Sin[2 t])}

In[] := developpee = ParametricPlot[Evaluate[c[t]], {t, 0, 2 π},
          PlotStyle -> RGBColor[0, 0, 1]];
```

(Figure non montrée)

```
In[] := Show[cardioide, developpee,
         Epilog -> {PointSize[1 / 50], Point[m[1]], Point[c[1]]}];
```

-Figure 15-



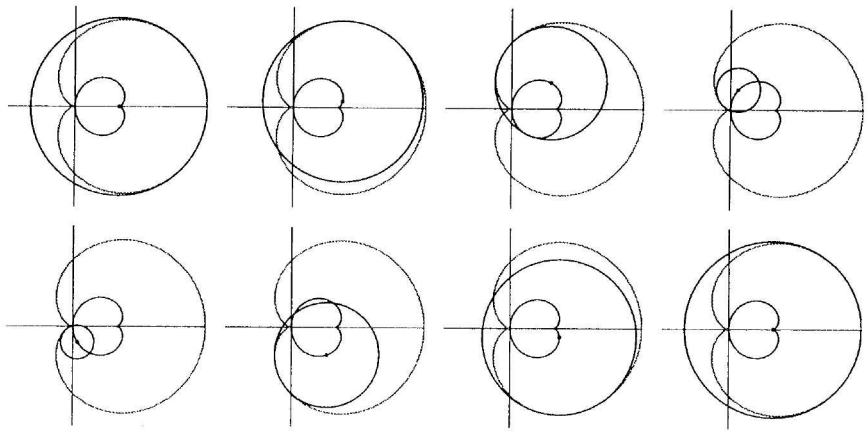

```
In[] := Do[Show[cardioide, developpee, PlotRange → {{-1, 2}, {-3 / 2, 3 / 2}},
          Epilog → {Circle[c[t], norme[c[t] - m[t]]], PointSize[0.02],
            Point[c[t]]}, Ticks → None], {t, 0, 6, 0.5}]
```

(Figure non montrée)

Pour représenter cette animation, nous programmons le tableau suivant :

```
In[] := Table[ Show[cardioide, developpee,
        PlotRange → {{-1, 2}, {-3 / 2, 3 / 2}}, Epilog → {Circle[c[t],
        norme[c[t] - m[t]]], PointSize[0.02], Point[c[t]]}, Ticks -> None],
         {t, 0, 6.16, 0.88}];

In[] := Show[GraphicsArray[Partition[%, 4]]];
```

-Figure 16-



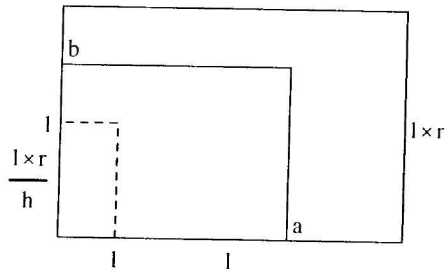
3.6 - Rendu des primitives; dimensions réelles d'un graphique

Ce qu'il faut savoir au sujet du rendu des primitives dans un graphique 2D :

- Les primitives sont placées dans l'ordre où elles apparaîssent dans la liste du premier membre de l'objet graphique. C'est de ce point de vue que les options `Prolog` et `Epilog` peuvent se distinguer.
- Les axes sont toujours rendus en premier; ils seront donc recouverts par n'importe quelle primitive opaque.
- La primitive `Rectangle` produit un rectangle opaque, sauf lorsqu'elle est remplie par un graphique, auquel cas aucune directive extérieure à la structure de ce graphique n'a d'effet (ainsi `Rectangle[{a,b},{c,d},Graphics[{}]]` produit un rectangle transparent).

Ce qu'il faut savoir au sujet de la taille réelle d'un graphique 2D :

Supposons qu'on ait construit un graphique 2D avec pour les options `AspectRatio` et `PlotRange`, les valeurs `r` et $\{\{0, l\}, \{0, h\}\}$. Supposons que l'unité sur l'axe des abscisses soit l'unité graphique réelle dans le plan; les dimensions réelles du graphique sont alors l et l×r, et la longueur réelle de l'unité sur les ordonnées est $\frac{l \times r}{h}$.



Introduisons maintenant un rectangle (à l'aide de la primitive `Rectangle`) de position $\{\{0, 0\}, \{a, b\}\}$ (noter que cette matrice est transposée par rapport à celle qui indique une valeur de `PlotRange`).

Les dimensions réelles de ce rectangle seront alors a et b×r×$\frac{l}{h}$ et ses proportions : $r \times \frac{l}{h} \times \frac{b}{a}$.

Un programme pour illustrer ces remarques

Nous proposons ci-dessous un petit programme qui rajoute en encadré dans un graphique, une légende pour les courbes tracées.

```
In[] := plotLegende[graph_, dim_, texte_, style_: {{}}] := Module[
          {letexte = If[Head[texte] === List, texte, {texte}],
           lestyle = If[Head[style] === List, style, {style}],
           ar = FullOptions[graph, AspectRatio],
           pr = FullOptions[graph, PlotRange], lt, motifs, textes, h, k},
          lt = Length[letexte];
          If[Length[lestyle] < lt, lestyle = Nest[Join[lestyle, #]&,
             lestyle, Ceiling[lt / Length[lestyle]] - 1]];
          h = 1 / (lt + 1);
          lestyle = Map[Flatten[{#}]&, lestyle];
          motifs = Table[Append[lestyle[[k]],
             Line[{Scaled[{1 / 10, k h}], Scaled[{1 / 2, k h}]}]], {k, lt}];
          textes = Table[
            Text[letexte[[k]], Scaled[{0.6, k h}], {-1 / 2, 0}], {k, lt}];
          pr = 1 / (Divide @@ (Subtract @@ Transpose[pr]));
```

(suite du programme page suivante)

```
    Show[graph, Epilog ->
    {GrayLevel[1], Rectangle[Sequence @@ dim],
    Rectangle[Sequence @@ dim, Graphics[{motifs, textes},
        PlotRange -> All, Frame -> True, FrameTicks -> None,
      FrameStyle -> {GrayLevel[0.7], Thickness[1 / 50]},
      AspectRatio -> ar / (Divide @@ (Subtract @@ dim)) / pr]]}
     ]
    ]
```

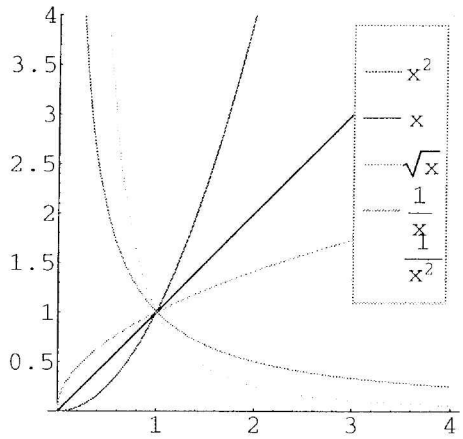
On prépare un graphique comportant plusieurs courbes avec des styles différents :

```
In[] := listefonc = {x^-2, x^-1, √x, x, x^2};
        listestyles = Table[Hue[k/6], {k, 5}];

In[] := Plot[Evaluate[listefonc], {x, 0, 4}, PlotStyle -> listestyles,
        AspectRatio -> Automatic, PlotRange -> {0, 4}];
```

(Figure non montrée)

Après avoir repéré la zone du graphique où on désire introduire la légende (ici :{{2.5, 1.5}, {4, 4}}; il faut penser aux proportions du rectangle pour loger le texte prévu), on utilise notre fonction `plotLegende` en introduisant le texte et le style (pour ce dernier, ce sera la même valeur que pour l'option `PlotStyle`) :

```
In[] := plotLegende[%, {{3, 1}, {4, 4}}, listefonc, listestyles];
```



-Figure 17-

Quelques remarques sur le programme :

– L'argument `texte` peut être une expression simple ou une liste; on la transforme en une liste dans tous les cas; de même pour `style`. Bien sûr, on ne peut pas réaffecter les paramètres; on crée donc de nouvelles variables (`letexte`, `lestyle`).

– Si le nombre d'éléments de la liste `letexte` est inférieur à celui de `lestyle`, on la rallonge en la reproduisant cycliquement.

– Les éléments de la liste `lestyle` peuvent être des directives ou des listes de directives; on les transforme tous en listes (simples) de directives.

– Dans chacune de ces listes, on ajoute la primitive `Line` (segment de courbe à représenter); l'ordonnée est calculée en coordonnées écran utilisant la fonction `Scaled` .

– La liste `letexte` est ensuite transformée en liste de primitives Text.

– Il reste à composer le graphique à l'intérieur de la primitive `Rectangle`. En fait, on superpose deux primitives `Rectangle` pour rendre la légende opaque. La valeur de l'option `AspectRatio` est calculée comme expliqué plus haut, pour que le graphique remplisse toute la zone prévue pour la primitive `Rectangle`.

– Si on voulait éviter de recopier la valeur de l'option `plotStyle`, comme on l'a fait pour `AspectRatio` et `PlotRange`, il faudrait modifier le programme de la fonction `Plot` au lieu de créer une nouvelle fonction, car cette option n'est pas attachée à la fonction `Graphics`.

Le lecteur intéressé pourra étudier le fichier de commandes `Graphics`Legend``, où figure un programme plus élaboré; et naturellement, s'il veut introduire des légendes dans ses graphiques, il utilisera ce fichier.

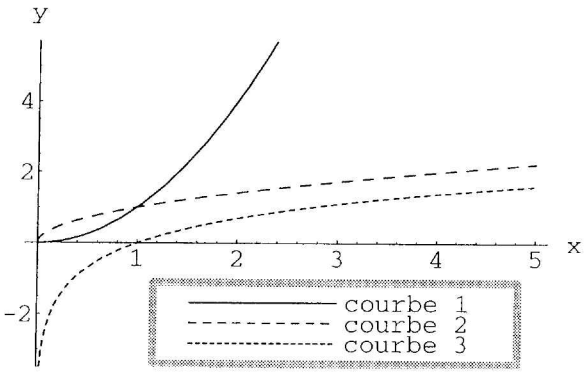
Toutefois, le programme ci-dessus a l'avantage de la simplicité si on désire maîtriser les dimensions et la position de la légende sans avoir à connaître les nombreuses options du fichier précédent.

```
In[] := Plot[{Log[x], √x, x^2},
          {x, 0, 5}, AxesLabel -> {"x", "y"}, PlotStyle ->
           {Dashing[{1/100, 1/100}], Dashing[{1/50, 1/50}], {}}];
```

(Figure non montrée)

```
In[] := plotLegende[%,
          {{1, -3.5}, {5, -1}}, {"courbe 3", "courbe 2", "courbe 1"},
          {Dashing[{1/100, 1/100}], Dashing[{1/50, 1/50}], {}}];
```

-Figure 18-



Réponses aux exercices

Exercice 1 :

Les deux subdivisions de l'intervalle {tmin, tmax} qui encadrent le pôle $-\frac{1}{4}$ correspondent aux deux "extrémités" de la branche infinie et sont joints par un segment qui approche l'asymptote oblique.

```
In[] := gr = ParametricPlot[{x[t], y[t]}, {t, -20, 20},
          AspectRatio -> Automatic,
          PlotRange -> {{-0.35, 0.35}, {-0.25, 0.25}} ];
```

Voici les valeurs extrêmes de x pour les points du graphique (faire afficher successivement `gr[[1]]`, `gr[[1,1]]`, ... pour mieux comprendre `gr[[1,1,1,1]]`):

```
In[] := {Min[#], Max[#]}& @ (First /@ gr[[1, 1, 1, 1]])
Out[] = {-1.90235, 125.9}
```

Pour supprimer ce segment, il faut séparer la primitive :

```
In[] := Line[{{x1, y1}, ..., {xp, yp}, {xp+1, yp+1}, ..., {xn, yn}}]
```

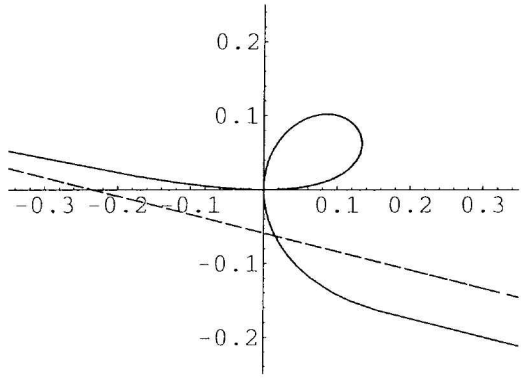
en deux primitives `Line` en coupant au bon endroit :

```
In[] := {Line[{{x1, y1}, ..., {xp, yp}}], Line[{{xp+1, yp+1}, ..., {xn, yn}}]}
```

Une première méthode consiste à construire le graphique en deux temps :

```
In[] := Show[ ParametricPlot[{x[t], y[t]},
          {t, -20, -1/4}, DisplayFunction -> Identity],
        ParametricPlot[{x[t], y[t]}, {t, -1/4, 20},
        DisplayFunction -> Identity],
        ParametricPlot[{xas, yas}, {t, -20, 20}, PlotStyle ->
          Dashing[{1/50, 1/50}], DisplayFunction -> Identity],
        DisplayFunction -> $DisplayFunction, AspectRatio -> Automatic,
        PlotRange -> {{-0.35, 0.35}, {-0.25, 0.25}} ];
```

-Figure 19-



On peut raccourcir ce programme en utilisant la structure `Block` pour changer localement la valeur de `$DisplayFunction` :

```
In[] := Show[ Block[{$DisplayFunction = Identity},
          {ParametricPlot[{x[t], y[t]}, {t, -20, -1/4}],
          ParametricPlot[{x[t], y[t]}, {t, -1/4, 20}],
          ParametricPlot[{xas, yas}, {t, -20, 20},
        PlotStyle -> Dashing[{1/50, 1/50}]]}],
        AspectRatio -> Automatic,
        PlotRange -> {{-0.35, 0.35}, {-0.25, 0.25}} ];
```

(Figure non montrée)

Une seconde méthode consiste à modifier la structure du graphique par programme et fait l'objet de l'exercice suivant.

Exercice 2 :

Le problème est de savoir quand deux points consécutifs sont "suffisamment éloignés" pour être considérés comme allant créer une "fausse asymptote"; sinon on risque de créer des "trous" dans la courbe.

Nous prendrons comme distance $|x|+|y|$ (ce qui accélère les calculs) et nous considérerons que deux points sont suffisamment éloignés si leur distance dépasse 2 fois la somme des dimensions du graphique (on peut faire d'autres choix). Une expression graphique normale conserve tous les points, y compris ceux qui n'apparaissent pas si on modifie la valeur de l'option `PlotRange`; tandis que notre programme dépendra de cette valeur; il pourra être nécessaire de l'appliquer au graphique initial si on augmente la valeur de cette option.

```
In[] := depasse[{a_, b_}, r_] := (Plus @@ Abs[b - a]) > r
```

```
In[] := coupeligne[l_, r_] := Module[
          {pos},
          pos = Union[
            Flatten[
             Position[
              Partition[l, 2, 1], p_?(depasse[#, r]&), 1
              ]
             ], {0, Length[l] + 1}
            ];
          pos = Cases[
            Partition[pos, 2, 1],
            {a_, b_} /; b - a > 2 -> {a + 1, b - 1}
            ];
          Line /@ (Take[l, #]& /@ pos)
         ]
```

```
In[] := coupegraph[gr_Graphics] := With[
          {r =
            Plus @@ (Last[#] - First[#]& /@ FullOptions[gr, PlotRange])},
          gr /. Line[l_] :> coupeligne[l, r] ]
```

Dans les deux graphiques ci-dessous, les fonctions `x[t]`, `y[t]`, `xas`, `yas` sont celles définies au paragraphe 3.3 :

```
In[] := ParametricPlot[{{x[t], y[t]}, {xas, yas}},
         {t, -20, 20}, AspectRatio -> Automatic,
         PlotRange -> {{-0.35, 0.35}, {-0.25, 0.25}}];
```

(Figure non montrée)

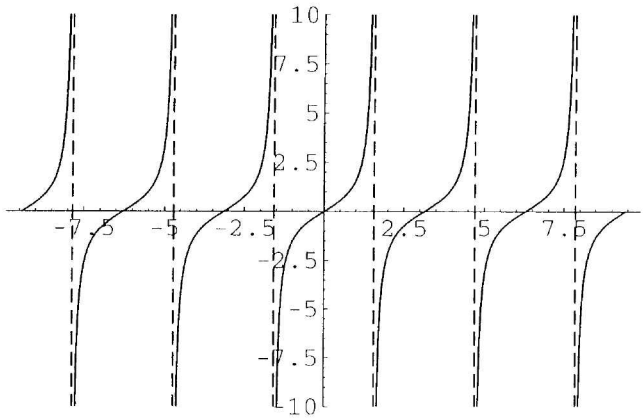
```
In[] := Show[coupegraph[%]];
```

(Figure non montrée)

On peut ainsi tracer des asymptotes "exactes" et dans le style de son choix :

```
In[] := Show[coupegraph[Block[{$DisplayFunction = Identity},
          Plot[Tan[x], {x, -3 Pi, 3 Pi}]]],
        PlotRange -> {-10, 10}, Epilog -> {Dashing[{1/50, 1/75}], Table[
          Line[{{Pi/2 + k Pi, 10}, {Pi/2 + k Pi, -10}}], {k, -3, 2}]}];
```

-Figure 20-

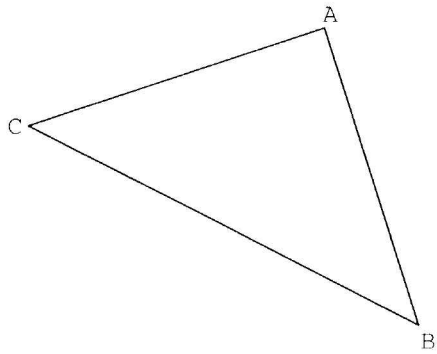


Exercice 3 :

```
In[] := triangle[s : {_, _, _}, a : {_, _, _}] := With[
          {g = (Plus @@ s) / 3},
          Graphics[{
           Line[Append[s, First[s]]],
            MapThread[Text, {a, s, Map[(g - #) / Sqrt[(g - #) . (g - #)]&, s]}]
           },
          AspectRatio -> Automatic, PlotRange -> All] ]
```

g est le centre de gravité du triangle; le décalage pour le sommet a1 de coordonnées s1 est calculé pour être le vecteur $\frac{g-s1}{\|g-s1\|}$.

```
In[] := Show[triangle[{{0, 5}, {1, 2}, {-3, 4}}, {"A", "B", "C"}]];
```



-Figure 21-

Chapitre 3

Un exemple de programmation : les coniques

1 - Tracé d'une ellipse

Passer de l'utilisation courante de *Mathematica* à la programmation se fait de façon naturelle grâce aux règles de réécriture. On pense le problème à résoudre en termes mathématiques, qu'il suffit ensuite de transcrire dans le langage du logiciel.

Les problèmes se compliquent toutefois lorsqu'on désire "soigner l'interface", en cherchant à construire des fonctions répondant au maximum de besoins et qui soient cohérentes avec le système *Mathematica*.

A titre d'exemple étudions le problème suivant : on veut faire tracer une ellipse donnée par ses demi-axes a, b, son centre $\{x_0, y_0\}$ et la direction de l'axe focal $\{\text{dir}_x, \text{dir}_y\}$.

- Le problème peut être abordé en plusieurs étapes:

  1 - Si l'ellipse est rapportée à ses axes, elle est paramétrée par le couple de fonctions $\{a\,\text{Cos}(t), b\,\text{Sin}(t)\}$ immédiatement dessinable par la commande :

  ```
  ParametricPlot[{f[t], g[t]},{t ,tmin, max}, options].
  ```

  2 - Si l'ellipse est seulement translatée par rapport aux axes, les fonctions deviennent: $\{a\,\text{Cos}(t) + x_0, b\,\text{Sin}(t) + y_0\}$.

  3 - Une ellipse quelconque, dont l'axe forme un angle φ avec le premier vecteur du repère sera paramétrée par une fonction : $M \mapsto$ rotation[φ] [M] + M_0, la matrice de rotation[φ] s'obtenant immédiatement à partir du vecteur $\{\text{dir}_x, \text{dir}_y\}$ après l'avoir normé.

  4 - La fonction qui trace notre ellipse devra se comporter comme une fonction *Mathematica* standard, en particulier pouvoir suporter les options graphiques usuelles.

1.1 - Fonction `Ellipse` : les deux premières étapes

Les deux premières étapes de notre problème sont immédiates à résoudre.

Le symbole `Ellipse` va servir pour toutes les fonctions que nous allons progressivement perfectionner; il est important de l'effacer chaque fois que l'on corrige le programme (sinon certaines règles s'ajoutent au lieu de s'écraser). Noter comment le système de programmation par règles de réécritures permet facilement de définir des arguments optionnels dans la fonction `Ellipse` (si on ne précise pas le centre, celui-ci est l'origine) :

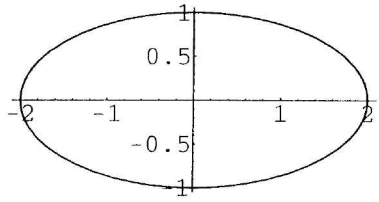
```
In[] := Clear[Ellipse]
        Ellipse[a_, b_] := ParametricPlot[
         {a Cos[t], b Sin[t]}, {t, 0, 2 Pi},
         AspectRatio -> Automatic ]

        Ellipse[a_, b_, {x0_, y0_}] := ParametricPlot[
         {a Cos[t] + x0, b Sin[t] + y0}, {t, 0, 2 Pi},
         AspectRatio -> Automatic ]
```

```
In[] := ?Ellipse
```

(Sortie non montrée)

```
In[] := | Ellipse[2, 1];
```

-Figure 1-



Remarquons qu'à ce stade, on pouvait obtenir les mêmes figures sans utiliser la fonction `ParametricPlot`, mais en écrivant un simple programme utilisant la primitive `Circle` (nous engageons le lecteur à le faire).

1.2 - Fonction `Ellipse` : la troisième étape (rotation du repère)

Au lieu d'entrer en bloc les équations paramétriques de l'ellipse (compliquées) dans la fonction `ParametricPlot`, définissons des fonctions auxilliaires suivantes :

- Il est aisé de calculer la norme d'un vecteur (le point est l'opérateur "produit scalaire").

  Rappelons que la forme `u_?VectorQ` filtre les correspondances, ne laissant passer que les expressions pour lesquelles `VectorQ[u]` retourne `True` (structurellement, un vecteur est une simple liste dont aucun élément n'est une liste) :

```
In[] := | Clear[norme]
        | norme[u_?VectorQ] := Sqrt[u . u]
```

```
In[] := | norme[{a1, a2, a3}]
```

$$\text{Out[]} = \sqrt{a_1^2 + a_2^2 + a_3^2}$$

- Nous pouvons définir une fonction `rotation` à deux arguments : `rotation[{α, β}, {x, y}]` retourne l'image du point $\{x, y\}$] par la rotation de centre O (origine du repère), d'angle $\varphi = (i, u)$, i premier vecteur unitaire du repère et u de composantes $\{\alpha, \beta\}$. (Nous verrons plus loin une syntaxe plus cohérente pour manipuler des fonctions, mais ici nous définissons un objet à usage "privé").

  La syntaxe `m:{_,_}` permet de manipuler le point en bloc, tout en précisant un "type"; on peut bien sûr écrire aussi `{x0_,y0_}`. On utilise enfin la propriété de "listablilité" de l'opération de division.

```
        | Clear[rotation];
In[] := | rotation[{α_, β_}, m:{_, _}] :=
        | ({{α, -β}, {β, α}}/norme[{α, β}]) . m
```

```
In[] := | rotation[{1, 2}, {1, 0}]
```

$$\text{Out[]} = \left\{\frac{1}{\sqrt{5}}, \frac{2}{\sqrt{5}}\right\}$$

- Enfin on complète le programme de la fonction `Ellipse`.

  – La fonction `Module[{symboles locaux},instructions]` qui enveloppe notre programme sert à "protéger" la variable t.

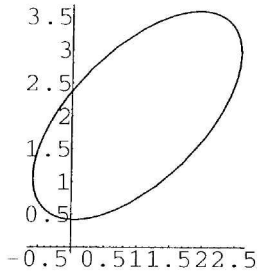
  – La fonction `Evaluate` qui enveloppe le premier argument de `ParametricPlot` sert à forcer l'évaluation de cet argument en premier (voir aussi les remarques à la fin de la section)

```
In[] := Ellipse[a_,b_,{x0_,y0_},dir:{_,_}] := Module[{t},
            ParametricPlot[
                Evaluate[rotation[dir,{a Cos[t],b Sin[t]}] + {x0,y0}],
                {t,0,2 Pi}, AspectRatio -> Automatic ] ]

In[] := ?Ellipse
```

(Sortie non montrée)

```
In[] := Ellipse[2,1,{1,2},{1,1}];
```

-Figure 2-



Remarques sur le programme de la troisième règle attachée à la fonction Ellipse :

- L'évaluation normale des arguments de la fonction `ParametricPlot` :

  Lors de l'appel de la fonction, l'itérateur $\{t, t_{min}, t_{max}\}$ est évalué en premier, d'abord à t_{min}, puis $t_{min} + h$, $t_{min} + 2h$, etc... jusqu'à t_{max} (h est un pas variable calculé par un algorithme de lissage), puis pour chacune de ces valeurs de t, la fonction procède à l'évaluation de son premier argument. On comprend donc que si on ne met pas `Evaluate`, la fonction `rotation` est appelée et évaluée après chacune des affectations de t (il y en a beaucoup). En entourant le premier argument à l'aide de `Evaluate`, il n'y a plus qu'un appel à la fonction rotation (le résultat dépendant de t, non encore affecté).

- Pour accélérer l'exécution des fonctions numériques et graphiques, *Mathematica* simule un pseudo-compilateur. Mais ce compilateur ne manipule que certaines fonctions dont ne font pas partie les fonctions utilisateur comme `rotation`, ce qui ralentit encore plus l'exécution. Là encore, la présence de `Evaluate` fait qu'on se retrouve dans le premier membre avec une expression de la forme $\{f(t), g(t)\}$ où $f(t)$ et $g(t)$ sont des expressions qui pourront être compilées.

- Enfin le symbole "muet" (ici t) est bien protégé à l'intérieur de la fonction en cas d'usage normal : s'il avait reçu une affectation avant l'appel de la fonction, cette valeur est sauvegardée, puis restituée en fin d'appel . Mais il n'en n'est plus de même si on force l'évaluation du premier argument, puisque `Evaluate[expression]` s'exécute de la même façon à l'intérieur de la fonction `ParametricPlot` qu'en toutes autres circonstances. Il faut donc dans ce cas soit "vider" la valeur de t par un "`Clear`" (mais elle est perdue définitivement), soit protéger le symbole t en enveloppant notre programme avec une fonction `Module` : `Module[{x, y, ...}, instructions]` crée un environnement dans lequel les symboles x, y, ... sont locaux (*Mathematica* crée en fait de nouveaux symboles qui se substituent à x, y, ... à l'intérieur de `Module` et ne servent plus après).

- Dans les expériences suivantes, on affecte t, on exécute le programme (les messages émis en cours d'évaluation ne sont pas montrés), on mesure le temps d'éxécution puis on contrôle la valeur de t :

Expérience 1 : on ne prend aucune précaution; dans ce cas, inutile de mettre `Module`, le symbole t étant protégé localement au moment de l'évaluation de l'itérateur.

```
In[] := Clear[Ellipse1]; Ellipse1[a_,b_,{x0_,y0_},dir:{_,_}] :=
        ParametricPlot[rotation[dir,{a Cos[t],b Sin[t]}]
            + {x0,y0},{t,0,2 Pi}, AspectRatio -> Automatic ]

In[] := t = 5; Timing[Ellipse1[2,1,{1,2},{1,1}]][[1]]
```

(Messages et graphique non montrés)

```
Out[] = | 1.1 Second

In[] := | t

Out[] = | 5
```

Expérience 2 : On met Evaluate:

```
In[] := | Clear[Ellipse1]
        | Ellipse1[a_,b_,{x0_,y0_},dir:{_,_}] := ParametricPlot[
        |     Evaluate[rotation[dir,{a Cos[t],b Sin[t]}] + {x0,y0}],
        |     {t,0,2 Pi},
        |     AspectRatio -> Automatic ]

In[] := | t = 5; Timing[Ellipse1[2,1,{1,2},{1,1}]][[1]]

Out[] = | 0.25 Second
```

(Messages et graphique non montrés)

Dans cet exemple, la fonction `Evaluate` a placé dans le premier membre de `ParametricPlot` l'expression suivante qui a dessiné un point (n'oublions pas que t vaut 5) :

```
In[] := | rotation[{1,1},{2 Cos[5],1 Sin[5]}] + {1,2}
```

$$\text{Out[]} = \left\{1 + \sqrt{2}\ \text{Cos}[5] - \frac{\text{Sin}[5]}{\sqrt{2}},\ 2 + \sqrt{2}\ \text{Cos}[5] + \frac{\text{Sin}[5]}{\sqrt{2}}\right\}$$

Expérience 3 : Protection de t par effacement :

```
In[] := | Clear[Ellipse1]
        | Ellipse1[a_,b_,{x0_,y0_},dir:{_,_}] := (Clear[t];
        |     ParametricPlot[
        |         Evaluate[rotation[dir,{a Cos[t],b Sin[t]}] + {x0,y0}],
        |         {t,0,2 Pi}, AspectRatio -> Automatic ] )

In[] := | t = 5; Timing[Ellipse1[2,1,{1,2},{1,1}]][[1]]
```

(Messages et graphique non montrés)

```
Out[] = | 0.316667 Second
```

Expérience 4 : Protection de t par la fonction `Module`. (Version définitive)

```
In[] := | Clear[Ellipse1]
        | Ellipse1[a_,b_,{x0_,y0_},dir:{_,_}] := Module[{t},
        |     ParametricPlot[
        |         Evaluate[rotation[dir,{a Cos[t],b Sin[t]}] + {x0,y0}],
        |         {t,0,2 Pi}, AspectRatio -> Automatic ] ]

In[] := | t = 5; Timing[Ellipse1[2,1,{1,2},{1,1}]][[1]]
```

(Messages et graphique non montrés)

```
Out[] = | 0.333333 Second

In[] := | t

Out[] = | 5
```

1.3 - Fonction `Ellipse` : quatrième étape (utiliser les options graphiques)

Et si on voulait introduire des options graphiques ?

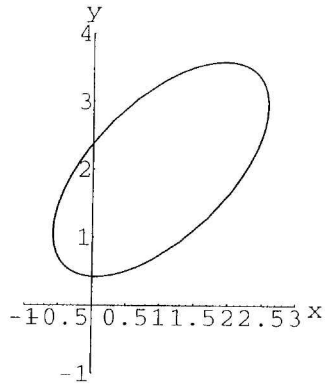
```
In[] := | Ellipse[2, 1, {1, 2}, {1, 1}, PlotRange -> {{-1, 3}, {-1, 3}}]
```

Il faut transmettre les options graphiques de `ParametriPlot` à notre fonction, ce qui se fait à l'aide d'un argument (`opts`) capable de prendre comme valeur une séquence formée d'un ou plusieurs arguments (et éventuellement aucun). On utilise pour cela la "forme blanche triple" (en anglais "triple blanck") : `opts___`:

```
In[] := |
Clear[Ellipse]
Ellipse[a_, b_, opts___] := ParametricPlot[
 {a Cos[t], b Sin[t]},
 {t, 0, 2 Pi},
 opts, AspectRatio -> Automatic ]

Ellipse[a_, b_, {x0_, y0_}, opts___] := ParametricPlot[
 {a Cos[t] + x0, b Sin[t] + y0},
 {t, 0, 2 Pi},
 opts, AspectRatio -> Automatic ]

Ellipse[a_, b_, {x0_, y0_}, dir : {_, _}, opts___] := Module[{t},
 ParametricPlot[
  Evaluate[rotation[dir, {a Cos[t], b Sin[t]}] + {x0, y0}],
  {t, 0, 2 Pi},
  opts, AspectRatio -> Automatic ] ]
```

```
In[] := | Ellipse[2, 1, {1, 2}, {1, 1},
  PlotRange -> {{-1, 3}, {-1, 4}}, AxesLabel -> {"x", "y"}];
```

-Figure 3-



Remarquons que la valeur de l'option `AspectRatio -> Automatic` devient la valeur par défaut en la placant après la séquence des options représentées par `opts` (dans la liste de règles qui sera appliquées, la première règle `AspectRatio -> valeur` rencontrée sera appliquée; ce sera celle-là si on n'en n'a pas introduite une autre dans l'appel de la fonction).

```
In[] := | Ellipse[1, 3, {1, 2}, AspectRatio -> 1];
```

(Figure non montrée)

1.4 - Fonction `Ellipse` : un seul programme pour tous les cas

Pour rendre plus concis le programme ci-dessus, on utilise des formes avec valeurs par défaut : si le troisième argument est omis, il prend la valeur par défaut {0, 0} et si le quatrième est omis, il prend la valeur {1, 0} (rotation identique).

```
In[] :=
Clear[Ellipse]
Ellipse[a_, b_, c_List: {0, 0},
  dir_List: {1, 0}, opts___Rule] := Module[{t},
 ParametricPlot[
  Evaluate[rotation[dir, {a Cos[t], b Sin[t]}] + c],
  {t, 0, 2 Pi},
  opts, AspectRatio -> Automatic
 ] ]
```

```
In[] := Ellipse[5, 2, AxesLabel -> {"x", "y"}]
```

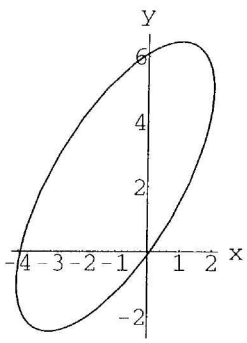
(Figure non montrée)

```
In[] := Ellipse[5, 2, {-1, 2}, AxesLabel -> {"x", "y"}]
```

(Figure non montrée)

```
In[] := Ellipse[5, 2, {-1, 2}, {1, √3}, AxesLabel -> {"x", "y"}];
```

-Figure 4-



Pour que ce programme fonctionne, il faut que les arguments, si on ne les entre pas, ne soient pas confondus avec des options; pour cela, nous introduisons des conditions sur les formes. Pour comprendre, essayer de voir ce que donnerait le programme ci-dessous si le premier member de la règle était :

```
In[] := Ellipse[a_, b_, c_: {0, 0}, dir_: {1, 0}, opts___] :=
```

On pourrait même entrer des conditions plus précises; en cas de non correspondance des formes, la fonction est ainsi retournée inévaluée au lieu de partir dans des exécutions génératrices d'erreurs :

```
In[] :=
Clear[Ellipse]
Ellipse[a_?Positive, b_?Positive, c_List: {0, 0},
 d_List: {1, 0}, opts___Rule] :=
Module[{t},
 ParametricPlot[
  Evaluate[rotation[d, {a Cos[t], b Sin[t]}] + c],
  {t, 0, 2 Pi},
  opts, AspectRatio -> Automatic
 ] ]
```

(Les possesseurs d'une version 2 devront peut-être remplacer le test `u_?Positive` par `u_?(Positive[N[#]]&))`.

```
In[] := | Ellipse[√2, π/4, {-1, 2}]
```

(Résultat et Figure non montrés)

```
In[] := | Ellipse[√2, a, {-1, 2}]

Out[] = | Ellipse[√2, a, {-1, 2}]
```

- Rappelons les différentes façons d'introduire des conditions sur les formes :
  - `u_h` où `h` est un symbole désignant un type (ou une tête); ce peut-être un nom de fonction utilisateur; la correspondance a lieu si `u` est de type `h`
  - `u_?fonc` où `fonc` est une fonction (symbole ou fonction pure); la correspondance a lieu si `fonc[u]` retourne la valeur `True` (si `fonc` est une fonction pure, elle doit être mise entre parenthèses)
  - `u_/;conditions` où `conditions` est une expression contenant `u`; la correspondance a lieu pour une valeur `v` de `u` si l'expression : `conditions/.v->u` s'évalue à True

  Ces différentes formes peuvent se combiner pour donner des structures complexes, parfois un peu ésotériques (et qui ne marchent pas toujours : essayer systématiquement une forme complexe avant de l'introduire dans un programme, comme montré ci-dessous).

```
In[] := | f[x_List?(Length[#] == 2&) /; x[[1]] > 0 && x[[2]] < 0] := {x}

In[] := | f[{2, -1}]

Out[] = | {{2, -1}}

In[] := | f[{1, x}]

Out[] = | f[{1, x}]
```

- Pour terminer ces quelques remarques sur la programmation, attention à l'utilisation des deux points dans une forme :
  - placé devant un symbole, il sert à nommer une forme (ex : `u:{_?NumberQ,_Integer}`)
  - placé après une forme, il sert à indiquer une valeur par défaut (ex : `u_List:{0,0}`)

2 - Représentation des trois coniques

En fait pour construire des figures contenant des coniques, il n'est pas judicieux de demander aux fonctions telle que la fonction `Ellipse`, de dessiner à l'écran l'objet graphique qu'elle a construit.

A partir de là nous aurons un choix à faire pour notre fonction; elle pourra :

– soit produire une liste de primitives graphiques; si `listePrim` est une telle liste, on pourra la tracer à l'écran par la commande `Show[Graphics[listePrim]]`, ou la passer directement dans une option `Prolog` ou `Epilog` d'une autre fonction graphique

– soit produire un objet de type `Graphics`; si `graph` est un tel objet, il pourra être directement tracé par `Show[graph]`, et on accédera à la liste des primitives par `First[graph]`

C'est cette dernière option que nous allons adopter. Cette façon de faire est plus proche de l'esprit de programmation sous *Mathematica*. Les options propres à notre Ellipse se passeront dans la fonction `Ellipse`, et les options générales de dessin, dans la fonction `Show`.

2.1 - Représentation d'une ellipse

2.1.1 - Créer un objet graphique à l'aide de la fonction `Ellipse`

Notre fonction `Ellipse` doit donc retourner l'objet graphique produit par la fonction `ParametricPlot`, sans le tracer. Il suffit pour cela de passer en option dans cette fonction : `DisplayFunction -> Identity`. Cependant, dans l'objet graphique ainsi récupéré, l'option `DisplayFunction -> Identity` va subsister, et empêchera l'affichage ultérieur du graphique par la fonction `Show` (à moins de modifier explicitement cette option chaque fois qu'on appelle `Show`).

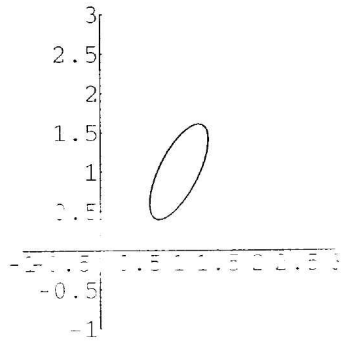
Pour palier à ce problème, on utilise la structure `Block`, qui permet de modifier temporairement la variable globale `$DisplayFunction`. Les variables déclarées dans le premier argument de `Block` (qui est une liste) voient leurs valeurs sauvegardées et restituées en sortie de la structure.

```
In[] := t = 5;

In[] := Block[{t = 2}, Print[t]]
        2

In[] := t

Out[] = 5
```

Mais contrairement aux structures `Module` ou `With`, le symbole représentant cette variable n'est pas modifié; c'est cette propriété que l'on utilise dans le programme ci-dessous; la structure `Block` sert à la fois à protéger *t* et à redéfinir localement `$DisplayFunction`.

```
In[] := Clear[Ellipse]
        Ellipse[
          a_?Positive, b_?Positive, c_List: {0, 0}, d_List: {1, 0}] :=
         Block[{$DisplayFunction = Identity, t},
          ParametricPlot[
           Evaluate[rotation[d, {a Cos[t], b Sin[t]}] + c],
           {t, 0, 2 Pi}, AspectRatio -> Automatic ] ]

In[] := t = 2;

In[] := Ellipse[2/3,1/4,{1,1},{1,2}]

Out[] = - Graphics -

In[] := Show[%, PlotRange -> {{-1,3},{-1,3}}];
```

-Figure 5-




```
In[] := Show[Ellipse[2 / 3, 1 / 4, {0, 0}, {1, 1}]];
```

(Figure non montrée)

2.1.2 - Introduire des options dans notre nouvelle fonction `Ellipse`

Si on voulait par exemple dessiner les axes de l'ellipse à la demande ? On ne peut pas indéfiniment créer des arguments "positionnels" pour une fonction (la règle est de ne pas en mettre plus de 3 ou 4).

Lorsqu'un argument est rarement employé, on l'introduit sous forme d'option, comme celles qui existent déjà pour les fonctions graphiques *Mathematica* (PlotRange, etc...).

Nous décidons d'introduire deux options pour notre fonction :

– l'option `Axes` dessine ou non les axes de la conique (valeur par défaut : `False`)

– l'option `PlotPoints` qui contrôle le nombre de points de subdivisions utilisés au départ pour le tracé (sa valeur par défaut est celle choisi pour l'option `Plotpoints` de `ParametricPlot`)

Notons que l'on a utilisé pour nos options des symboles système; lorsqu'un tel symbole peut convenir, on évite ainsi de multiplier inutilement les noms de symboles.

```
In[] :=
ClearAll[Ellipse]

Options[Ellipse] = {Axes -> False,
 PlotPoints -> (PlotPoints /. Options[ParametricPlot])};

Ellipse[a_?Positive, b_?Positive, c_List: {0, 0}, d_List: {1, 0},
    opts___Rule] := Block[
 {t, gaxe, paxe, objets = {},
   $DisplayFunction = Identity,
   div = PlotPoints /. {opts} /. Options[Ellipse],
 },
 If[Axes /. {opts} /. Options[Ellipse],
   gaxe =
    {rotation[d, {-5 a / 4, 0}] + c, rotation[d, {5 a / 4, 0}] + c};
   paxe =
    {rotation[d, {0, -5 b / 4}] + c, rotation[d, {0, 5 b / 4}] + c};
  objets = {Line[gaxe], Line[paxe]}
 ];
 ParametricPlot[
  Evaluate[rotation[d, {a Cos[t], b Sin[t]}] + c],
  {t, 0, 2 Pi},
  PlotPoints -> div,
  AspectRatio -> Automatic,
  Epilog -> objets
 ]
]
```

Quelques remarques sur le programme ci-dessus :

- Les valeurs courantes des options sont récupérées à l'aide de règles de remplacement qui sont appliquées au symbole représentant le nom de l'option, par exemple :

```
divisions  /. {opts} /. Options[Ellipse]
```

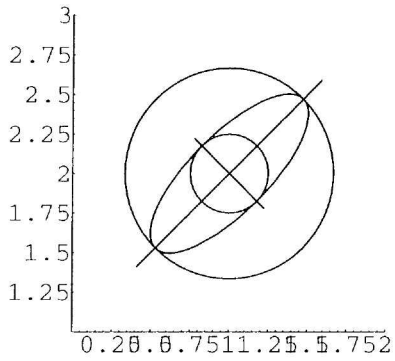
  – Les premières règles appliquées sont celles passées éventuellement dans l'argument `opts`; si on y a placé par exemple "`divisions -> 50`", alors le symbole `divisions` est remplacé par 50, et le processus de remplacement s'arrête (la seconde liste de règles ne peut s'appliquer puisque le symbole divisions ne figure plus).

  – Si les premières règles ne s'appliquent pas (le symbole divisions ne figure pas dans `{opts}`), alors c'est la seconde liste de règles qui s'applique : la valeur prise par `divisions` est la valeur par défaut pour la fonction `Ellipse`.

- Pour construire les objets graphiques auxilliaires, (les axes), on utilise l'option `Epilog`. Nous avons dessiné des axes de dimensions fixes relativement aux axes de l'ellipse; nous verrons plus loin un autre point de vue. Nous ne pouvons plus réutiliser l'option `Epilog`, ce qui effacerait les axes; nous nous servons donc de `Prolog` pour rajouter d'autres éléments (cercles principal et secondaire).

```
In[] := ell = Ellipse[2/3,1/4,{1,2},{1,1},Axes -> True];

In[] := Show[ell,PlotRange -> {{0,2},{1,3}},
        Prolog -> {Circle[{1,2},2/3],Circle[{1,2},1/4]}];
```

-Figure 6-



2.2 - Représentation des coniques à centre

Notre fonction `Ellipse` est pratiquement au point; on pourrait construire des fonctions analogues pour l'hyperbole et la parabole et s'en tenir là.

Dans la version définitive nous avons toutefois voulu rajouter une option permettant d'introduire des éléments graphiques (points, droites) dans les coordonnées relatives aux axes de la conique (option `Placer`); cela nous permettra de tracer facilement des éléments tels que foyers, directrices etc.

Lorsque tout est prêt, il est intéressant de construire un fichier de commandes (package) qui rassemble ces fonctions et permet de les intégrer aux fonctions *Mathematica* déjà existantes, et de profiter de toutes les commodités offertes par l'interface *Mathematica* (principalement l'aide en ligne).

- Ces fonctions vont posséder les options suivantes :
  - `Axes` : inclure ou non les axes de la conique et pour la fonction `Hyperbole`, l'option `Asymptotes`.
  - `PlotPoints` : nombre initial de points de subdivisions
  - `PlotRange` : pour préciser le cadrage de l'objet; si on modifie la valeur de cette option, les droites tracées seront retracées aux dimensions de la boite (donnant l'illusion de "droites illimitées")
  - `Placer` : fonctionne comme les options `Prolog` ou `Epilog`, mais les coordonnées des primitives qu'on y introduit se réfèrent au repère constitué par les axes de la conique.

    Cette option acceptera outre les primitives 2D usuelles, le symbole `Droite` : `Droite[a:{_,_},b:{_,_}]` représentera la droite "illimitée" joignant les points `a` et `b`.
  - `PlotStyle` : permet d'introduire des directives qui s'appliqueront à tous les tracés effectués.

2.2.1 - Fonctions auxilliaires.

Rappelons que nous avons déjà comme fonctions auxilliaires les fonctions `norme` et `rotation`.

Nous ajoutons une troisième fonction qui calcule les intersections d'une droite passant par deux points `a` = $\{x_a, y_a\}$ et `b` = $\{x_b, y_b\}$, avec les bords du rectangle `range` = $\{\{x_{min}, x_{max}\}, \{y_{min}, y_{max}\}\}$:

```
In[] :=
bordsDroite[range_,{a_,b_}] := Module[
    {s,bords,
    α = b[[1]] - a[[1]],β = b[[2]] - a[[2]]
    },
        bords = Sort[
        Select[
            Join[
                If[α != 0,
                    {s,β/α(s - a[[1]]) + a[[2]]} /.
                        ({s -> #}& /@ range[[1]]),
                    {}
                ],
                                    If[β != 0,
                    {α/β(s - a[[2]]) + a[[1]],s} /.
                        ({s -> #}& /@ range[[2]]),
                    {}
                ]
            ],
            range[[1,1]] <= #[[1]] <= range[[1,2]] &&
            range[[2,1]] <= #[[2]] <= range[[2,2]] &
        ],
                (b-a).(#2-#1) > 0 &
    ];
        If[bords === {},{},{First[#],Last[#]}& @ bords]
]
```

- Quelques explications sur ce programme :

  – On calcule les intersections de la droite (ab) avec les quatre droites formant les côtés du rectangle. Si (α, β) est le vecteur $b-a$, les points situés sur les côtés horizontaux, s'ils existent ($\alpha \neq 0$), s'obtiennent en substituant dans le couple $(s, \frac{\beta}{\alpha}(s-x_a)+y_a)$, s par $x_{\min}$ et $x_{\max}$ et ceux situés sur les bords horizontaux ($\beta \neq 0$), en substituant dans $\left(\frac{\alpha}{\beta}(s-y_a)+x_a, s\right)$, s par $y_{\min}$ et $y_{\max}$.

  – On obtient une liste de 2 ou 4 points où l'on sélectionne les éléments appartenant aux segments formant les côtés du rectangle (fonction `Select`) puis que l'on trie selon le critère "être dans le même sens que le vecteur $b-a$ " (fonction `Sort`). On obtient ainsi la liste `bords` qui est vide ou possède de 2 à 4 éléments dont 2 distincts au maximum.

  – Le tri effectué permet, s'il y a deux points distincts, de les séparer (on peut utiliser un autre critère); on ne garde que les points extrêmes; on obtient en fin de compte une liste vide ou deux points distincts ou confondus.

```
In[] :=
bordsDroite[{{-1, 1}, {-1, 1}},
 Array[Random[Real, {-2, 2}]&, {2, 2}]]

Out[] =
{{-1, 0.348054}, {1, 0.564449}}
```

On peut créer une animation :

```
In[] :=
Do[Show[Graphics[{
   Line @ bordsDroite[
      {{-1, 1}, {-1, 1}}, Array[Random[Real, {-2, 2}]&, {2, 2}]]
   }], Frame -> True, PlotRange -> {{-1, 1}, {-1, 1}}], {20}]
```

(Figures non montrées)

2.2.2 - Fonction Ellipse.

- Nous écrivons notre programme `Ellipse` définitif. Nous ferons quelques remarques sur ce programme :
  - Après avoir récupéré l'objet `graph` produit par `ParametricPlot`, nous récupérons la valeur effective de l'option `PlotRange` à l'aide de la fonction `FullOptions`; cette valeur nous servira pour les appels à la fonction `bordsDroite`.
  - Il est inutile ici d'utiliser la fonction `Block` car les options de `graph` sont perdues; on ne récupère que le premier membre pour le concaténer avec les primitives produites par les différentes options.
  - Noter la façon de récupérer la valeur de l'option `PlotStyle` dans une liste simple en utilisant `Flatten`, que la valeur de cette option soit entrée sous forme de liste ou non.
  - Lorsqu'on utilise dans le membre de droite d'une règle ou définition, des noms de formes (ici pour les règles qui construisent la liste `objets`), ne pas utiliser les mêmes noms que ceux des formes du membre de gauche.

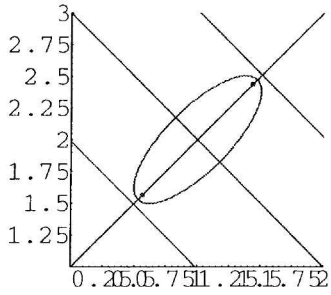
```
In[] :=
ClearAll[Ellipse]

Options[Ellipse] = {Axes -> False, Placer -> {},
 PlotPoints -> 50, PlotRange -> Automatic, PlotStyle -> {} };

Ellipse[a_?Positive, b_?Positive, c_List: {0, 0}, d_List: {1, 0},
    opts___Rule] := Module[ {t, graph, range, lesaxes = {},
  div = PlotPoints /. {opts} /. Options[Ellipse],
  objets = Placer /. {opts} /. Options[Ellipse],
  style = Flatten[{PlotStyle} /. {opts} /. Options[Ellipse]],
  gaxe = {rotation[d, {-1, 0}] + c, rotation[d, {1, 0}] + c},
  paxe = {rotation[d, {0, -1}] + c, rotation[d, {0, 1}] + c} },
 graph = ParametricPlot[
  Evaluate[rotation[d, {a Cos[t], b Sin[t]}] + c],
  {t, 0, 2 Pi},
  PlotPoints -> div,
  AspectRatio -> Automatic,
  PlotRange -> (PlotRange /. {opts} /. Options[Ellipse]),
  DisplayFunction -> Identity  ];
 range = FullOptions[graph, PlotRange];
 If[Axes /. {opts} /. Options[Ellipse],
  lesaxes = {Line[bordsDroite[range, N[gaxe]]],
   Line[bordsDroite[range, N[paxe]]]}  ];
 objets = objets /. {
  Point[u_] :> Point[rotation[d, u] + c],
  Line[u_] :> Line[rotation[d, #] + c & /@ u],
  Polygon[u_] :> Polygon[rotation[d, #] + c & /@ u],
  Circle[u_, rest__] :> Circle[rotation[d, u] + c, rest],
  Disk[u_, rest__] :> Disk[rotation[d, u] + c, rest],
  Rectangle[min_, max_, rest___] :> Rectangle[
   rotation[d, min] + c, rotation[d, max] + c, rest],
  Droite[u_, v_] :> Line[bordsDroite[
   range, {rotation[d, u] + c, rotation[d, v] + c}]],
  Text[st_, u_, rest___] :> Text[st, rotation[d, u] + c, rest] };
 Graphics[Join[style, graph[[1]], lesaxes, objets],
  {AspectRatio -> Automatic, Axes -> True,
  PlotRange -> range}
 ] ]
```

Pour tester notre programme, nous représentons une ellipse avec axes, foyers, directrices.

Les directrices peuvent être passées sous formes de droites "illimitées" (`Droite`) ou de segments (`Line`).

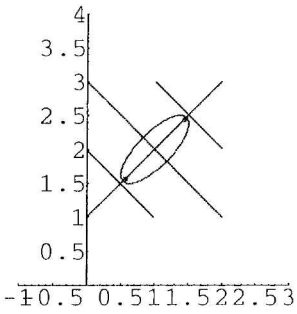
```
In[] := | a = 2/3; b = 1/4; c = Sqrt[a^2 - b^2]; k = a^2/c;

        | ell = Ellipse[a,b,{1,2},{1,1},Axes -> True,
        |     Placer -> {PointSize[1/50],Point[{c,0}],Point[{-c,0}],
In[] := |         Droite[{-k,1/5},{-k,-1/5}],Droite[{k,1/5},{k,-1/5}]},
        |     PlotRange -> {{0,2},{1,3}},
        |     PlotStyle -> RGBColor[1,0,0]  ];

In[] := | Show[ell];
```

-Figure 7-




```
In[] := | Show[ell, PlotRange -> {{-1, 3}, {0, 4}}];
```

-Figure 8-

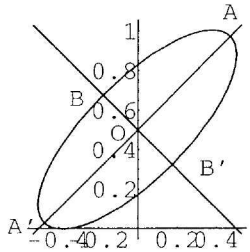


L'option `PlotRange` passée dans `Show` n'a plus d'effet sur les "droites illimitées".

Il est plus facile de placer du texte pour nommer les sommets et le centre de l'ellipse en se rapportant à ses axes. Les "offsets" sont ajustés après quelques essais (voir chapitre précédent) :

```
        | Show[Ellipse[a, b, {0, 1 / 2}, {1, 1}, Axes -> True,
        |  Placer -> {Text["A", {a, 0}, {0, -1.5}],
In[] := |    Text["A'", {-a, 0}, {1, 0}], Text["B", {0, b}, {2, 0}],
        |    Text["B'", {0, -b}, {-2, 0}], Text["O", {0, 0}, {1.5, 0}]}
        |  ], PlotRange -> All ];
```

-Figure 9-




```
In[] := | Clear[a,b,c,k]
```

2.2.3 - Fonction Hyperbole.

Le problème ici est le tracé des branches infinies; nous choisissons un paramétrage trigonométrique et prenons une valeur approchée par défaut de $\frac{\pi}{2}$ pour ne pas provoquer d'erreur d'exécution.

In[] :=

```
ClearAll[Hyperbole]

Options[Hyperbole] =
{Axes -> False, Placer -> {},Asymptotes -> False,
PlotPoints -> 50,PlotRange -> Automatic,PlotStyle -> {}};

Hyperbole[a_?Positive,b_?Positive,c_List:{0,0},d_List:{1,0},
        opts___Rule] := Module[
    {t,graph,range,gaxe,paxe,asymp1,asymp2,lesaxes = {},
        lesasymptotes = {},style = Flatten[{PlotStyle} /.
           {opts} /. Options[Hyperbole]],
        div = PlotPoints /. {opts} /. Options[Hyperbole],
        objets = Placer /. {opts} /. Options[Hyperbole] },
    graph = ParametricPlot[
        Evaluate[{rotation[d,{a/Cos[t],b Tan[t]}] + c,
        rotation[d,{-a/Cos[t],-b Tan[t]}] + c}],
        {t,-1.570796,1.570796},
        PlotPoints -> div,
        AspectRatio -> Automatic,
        PlotRange -> (PlotRange /. {opts} /. Options[Hyperbole]),
        DisplayFunction -> Identity ];
    range = FullOptions[graph,PlotRange];
    If[Axes /. {opts} /. Options[Hyperbole],
        gaxe = {rotation[d,{-1,0}]+c,rotation[d,{1,0}]+c};
        paxe = {rotation[d,{0,-1}]+c,rotation[d,{0,1}]+c};
        lesaxes = {Line[bordsDroite[range,N[gaxe]]],
            Line[bordsDroite[range,N[paxe]]]} ];
    If[Asymptotes /. {opts} /. Options[Hyperbole],
        asymp1 = {rotation[d,{-a,-b}]+c,rotation[d,{a,b}]+c};
        asymp2 = {rotation[d,{-a,b}]+c,rotation[d,{a,-b}]+c};
        lesasymptotes = {Line[bordsDroite[range,N[asymp1]]],
            Line[bordsDroite[range,N[asymp2]]]} ];
    objets = objets /. {
        Point[u_] :> Point[rotation[d,u]+c],
        Line[u_] :> Line[rotation[d,#]+c & /@ u],
        Polygon[u_] :> Polygon[rotation[d,#]+c & /@ u],
        Circle[u_,rest__] :> Circle[rotation[d,u]+c,rest],
        Disk[u_,rest__] :> Disk[rotation[d,u]+c,rest],
        Rectangle[min_,max_,rest___] :> Rectangle[
            rotation[d,min]+c,rotation[d,max]+c,rest],
        Droite[u_,v_] :> Line[bordsDroite[
            range,{rotation[d,u]+c,rotation[d,v]+c}]],
        Text[st_,u_,rest___] :> Text[st,rotation[d,u]+c,rest] };
Graphics[Join[style,graph[[1]],lesaxes,lesasymptotes,objets],
        {AspectRatio -> Automatic, Axes -> True,
        PlotRange -> range} ] ]
```

```
In[] := Show[ Hyperbole[2/3,1/2,{1,1/2},{1,1},
            Axes -> True,Asymptotes -> True,
            PlotRange -> {{-2,4},{-2,3}},
            PlotStyle -> RGBColor[0,0,1]] ];
```

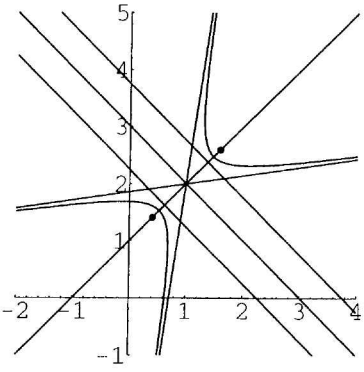
(Figure non montrée)

```
In[] := a = 2/3; b = 1/2; c = Sqrt[a^2 + b^2]; k = a^2/c;
```

```
In[] := Hyperbole[a,b,{1,2},{1,1},Axes -> True,Asymptotes -> True,
            Placer -> {PointSize[1/50],Point[{c,0}],Point[{-c,0}],
                Droite[{-k,2},{-k,-2}],Droite[{k,2},{k,-2}]},
                PlotRange -> {{-2,4},{-1,5}} ];
```

```
In[] := Show[%];
```

-Figure 10-



2.3 - Représentation de la parabole

La parabole est définie par son paramètre p, son sommet s et la direction d de son axe focal.

Le paramètre p est le double de la distance du foyer à la directrice; c'est ausi la moitié de la longueur de la corde passant par le foyer et perpendiculaire à l'axe; il mesure l'écartement de la courbe.

Le repère local pour les éléments de `Placer` est formé par l'axe et la tangente au sommet.

Nous avons choisi un paramétrage trigonométrique de la courbe pour une répartition des points de subdivisions indépendante de x et de y.

Un problème se pose au niveau du cadrage; l'algorithme utilisé par *Mathematica* lorsque l'option `PlotRange` est à `Automatic` ne fonctionne pas correctement (cela peut dépendre d'ailleurs de la version du logiciel). Nous décidons de gérer nous-même cette option; la fonction `transformeRange` prend en arguments une fonction `f`, un rectangle $\{\{x_{min}, x_{max}\}, \{y_{min}, y_{max}\}\}$ et retourne une rectangle de même type contenant son image par `f`.

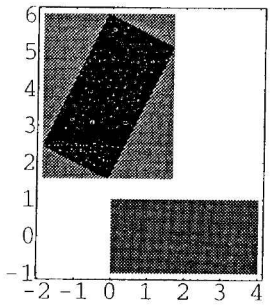
```
In[] := transformeRange[f_, range_] :=
         Map[{Min[#], Max[#]}& ,
          Transpose[
           Map[f,
            Apply[
             Flatten[Outer[List, ##], 1]&,
             range
] ] ] ]
```

Dans l'illustration ci-dessous, la fonction f est une rotation suivie d'une translation :

```
In[] := range1 = {{0, 4}, {-1, 1}};
```

```
In[] := imrange1 = Map[rotation[{1, 2}, #] + {-1, 2}&,
          Apply[ Flatten[Outer[List, ##], 1]&,
           range1 ]][[{2, 1, 3, 4}]];

In[] := range2 = transformeRange[rotation[{1, 2}, #] + {-1, 2}&, range1];

In[] := Show[Graphics[{{RGBColor[1, 0, 0],
          Rectangle @@ Transpose[range1]}, {RGBColor[0, 0, 1],
           Rectangle @@ Transpose[range2]},  Polygon[imrange1]},
         Frame -> True, AspectRatio -> Automatic]];
```

-Figure 11-



On part avec un rectangle $\{\{0, 50\}, \{-10\sqrt{p}, 10\sqrt{p}\}\}$ et la même fonction que celle qui transforme la parabole.

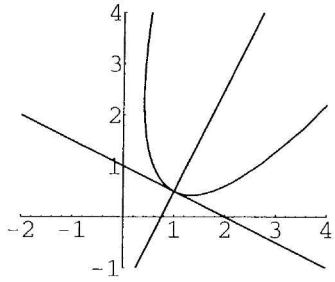
Le programme `Parabole` ci-dessous continue sur la page suivante :

```
In[] := ClearAll[Parabole]

        Options[Parabole] = {Axes -> False, Placer -> {},
        PlotPoints -> 50, PlotRange -> Automatic, PlotStyle -> {}};

In[] := Parabole[p_?Positive,s_List:{0,0},d_List:{1,0},opts___Rule] :=
        Module[{t,graph,lesaxes = {},
                style = Flatten[{PlotStyle}  /. {opts} /.
        Options[Parabole]],
                div = PlotPoints /. {opts} /. Options[Parabole],
                objets = Placer /. {opts} /. Options[Parabole],
                range =  PlotRange /. {opts} /. Options[Parabole] },
            If[range === Automatic,range = {{0,50},Sqrt[p]{-10,10}}];
            graph = ParametricPlot[
                Evaluate[rotation[d,{Tan[t]^2/(2 p),Tan[t]}]+s],
                {t,-1.570796,1.570796},
                PlotPoints -> div,
                AspectRatio -> Automatic,
                PlotRange -> range,
                DisplayFunction -> Identity ];
            If[range === Automatic,
                range = transformeRange[rotation[d,#] + s
        &,{{0,50},Sqrt[p]{-10,10}}]];
            If[Axes /. {opts} /. Options[Parabole],
                gaxe = {rotation[d,{-1,0}]+s,rotation[d,{1,0}]+s};
                paxe = {rotation[d,{0,-1}]+s,rotation[d,{0,1}]+s};
                lesaxes = {Line[bordsDroite[range,N[gaxe]]],
                    Line[bordsDroite[range,N[paxe]]]} ];
```

```
objets = objets /. {
        Point[u_] :> Point[rotation[d,u]+s],
        Line[u_] :> Line[rotation[d,#]+s & /@ u],
        Polygon[u_] :> Polygon[rotation[d,#]+s & /@ u],
        Circle[u_,rest__] :> Circle[rotation[d,u]+s,rest],
        Disk[u_,rest__] :> Disk[rotation[d,u]+s,rest],
        Rectangle[min_,max_,rest___] :> Rectangle[
            rotation[d,min]+s,rotation[d,max]+s,rest],
        Droite[u_,v_] :> Line[bordsDroite[
            range,{rotation[d,u]+s,rotation[d,v]+s}]],
        Text[st_,u_,rest___] :> Text[st,rotation[d,u]+s,rest] };
    Graphics[Join[style,graph[[1]],lesaxes,objets],
        {AspectRatio -> Automatic, Axes -> True,
        PlotRange -> range} ] ]
```

```
In[] := Show[Parabole[2/3,{1,1/2},{1,2},
            Axes -> True, PlotRange -> {{-2,4},{-1,4}}]];
```

-Figure 12-

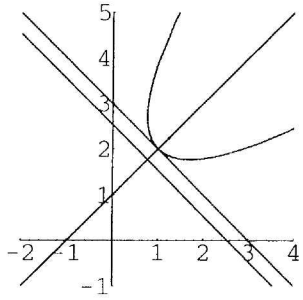



```
In[] := p = 2/3;
```

```
In[] := Parabole[p,{1,2},{1,1},Axes -> True, PlotStyle -> Hue[0.7],
            Placer -> {PointSize[1/100],Point[{p/2,0}],
            Droite[{-p/2,2},{-p/2,-2}]},PlotRange -> {{-2,4},{-1,5}}];
```

```
In[] := Show[%];
```

-Figure 13-



Une animation pour observer l'effet de la variation du paramètre :

```
In[] := Do[Show[Parabole[p = 2^(n/2), {0, 0}, {0, 1},
            PlotRange -> {{-10, 10}, {-15, 15}},
            PlotStyle -> RGBColor[1, 0, 0],
            Placer -> {PointSize[1 / 30], Point[{p / 2, 0}],
              Droite[{-p / 2, 1}, {-p / 2, -1}]}
            ]], {n, -6, 9, 1 / 2}]
```

(Figures non montrées)

3 - Réalisation d'un fichier de commandes

Il s'agit de regrouper les commandes précédentes dans un fichier. Nous allons profiter de cet exemple pour montrer comment réaliser un fichier de commandes à l'aide de *Mathematica* V.3.

On construit pour cela un cahier structuré comme indiqué ci-dessous; les cellules dans lesquelles figurent le code faisant partie du fichier de commande sont marquées cellules d'initialisation (un petit I apparaît dans le crochet). Au moment de l'enregistrement (dans le répertoire AddOns/Applications/Geometrie), un dialogue propose de construire un fichier de commades associé à ce cahier; on confirme. On peut tester le programme en activant le menu "Evaluate Initialization"; on procède alors aux essais proposés dans la section "Exemples d'utilisation"; si on désire procéder à des modifications du programme, on exécute les commandes de la sous-section "Réinitialisation". Le fichier de commandes associé sera automatiquement remis à jour.

Avec une version 2, lorsque le programme est au point, on le recopie dans un fichier comportant une seule cellule d'initialisation; dans chaque programme les instructions sont séparées par des retours à la lignes et les instructions elle-mêmes par des sauts de lignes. Les commentaires sont placés entre (* et *).

Cahier Coniques.nb

Introduction et réferences

On peut indiquer dans cette rubrique des informations d'ordre général; voici quelques rubriques :

Titre

Auteur

Sujet

Les fonctions Ellipse, Hyperbole, Parabole créent des objets graphiques représentant la conique correspondante. On les représente graphiquement à l'aide de la fonction Show.

Options:

– Divisions: pour modifier le nombre de points d'échantillonnage des graphiques.

– Axes: pour tracer ou non les axes de la conique (pour une parabole, l'axe et la tangente au sommet).

– Asymptotes: pour tracer ou non les asymptotes de l'hyperbole.

– Placer: pour introduire des primitives graphiques, dans le repère formé par les axes de la conique (on peut ainsi facilement tracer les foyers, les directrices, etc... ainsi que les axes si les axes tracés par défaut ne conviennent pas).

On peut aussi y introduire des droites "illimitées" par la fonction Droite.

Les autres options graphiques doivent être passées dans la fonction Show.

Si on passe PlotRange dans Show, les axes et les asymptotes (si options), ne sont pas recalculées pour aller aux limites du nouveau graphique.

Copyright

Versions

Mots clés

Sources

Interface

Cette partie déclare les fonctions et autres symboles publics.

La destination de notre fichier sera en version 3, le répertoire AddOns/Applications/Geometrie et en version 2 le répertoire Packages/Geometrie

Isolement des contextes de travail publics pour le fichier

```
In[] := BeginPackage["Geometrie`Coniques`"]
```

Messages d'usage

Le début du message d'usage doit indiquer correctement la syntaxe des fonctions pour être exploitée par les commandes de menus `complete Selection` et `Make Template`.

```
In[] := Ellipse::usage = "Ellipse[a,b,{x0,y0},dir,opts] représente
        une ellipse de demi-axes a et b, de centre {x0,y0}, dir
        indiquant la direction de l'axe focal.  Si on omet dir,
        l'axe focal est l'axe des abscisses et si on omet {x0,y0},
        le centre est à l'origine. L'ellipse peut être tracée en
        utilisant Show. Les options sont Axes, Placer, PlotPoints,
           PlotStyle, PlotRange."
```

```
In[] := Hyperbole::usage = "Hyperbole[a,b,{x0,y0},dir,opts]
        représente une hyperbole de demi-axes a et b, de centre
        {x0,y0},dir indiquant la direction de l'axe focal.
        Si on omet dir, l'axe focal est l'axe des abscisses et si on
        omet {x0,y0}, le centre est à l'origine. L'hyperbole
        peut être tracée en utilisant Show. Les options sont  Axes,
        Asymptotes, Placer, PlotPoints, PlotStyle, PlotRange."
```

```
In[] := Parabole::usage =
         "Parabole[a,b,{x0,y0},dir,opts] représente une parabole
        de paramètre p, de sommet {x0,y0}, dir indiquant la direction
        de l'axe focal. Si on omet dir, l'axe focal est l'axe des
        abscisses et si on omet {x0,y0}, le sommet est à l'origine.
        La parabole peut être tracée en utilisant Show. Les
        options sont Axes, Placer, PlotPoints, PlotStyle, PlotRange."
```

Les messages suivants concernent les options (pour les symboles système, on compléte les messages existants).

```
In[] := Axes::usage = Axes::usage <> "\n Axes est aussi une option
           indiquant de tracer ou non les axes d'une conique."
```

```
In[] := Placer::usage = "Placer est une option permettant d'introduire
           des primitives graphiques dans le tracé d'une conique.
           On peut aussi introduire des droites par la fonction Droite."
```

```
In[] := Droite::usage = "Droite[a,b] représente la droite passant
           par les points a et b à introduire dans le tracé d'
           une conique à l'aide de l'option Placer."
```

Options pour les symboles publics

```
In[] := Options[Ellipse] = {Axes -> False, Placer -> {},
         PlotPoints -> 50, PlotRange -> Automatic,  PlotStyle -> {} };
```

```
In[] := Options[Hyperbole] =
         {Axes -> False, Placer -> {},  Asymptotes -> False,
        PlotStyle -> {}, PlotPoints -> 50, PlotRange -> Automatic};
```

```
In[] := Options[Parabole] = {Axes -> False, Placer -> {},
         PlotPoints -> 50, PlotRange -> Automatic,  PlotStyle -> {} };
```

Messages d'erreurs pour les symboles publics

On n'a pas prévu ici de gérer des erreurs.

Implémentation

Cette partie déclare les fonctions et autres symboles privés et contient le code.

Mise en place d'un contexte courant privé

```
In[] := | Begin["`Private`"]
```

Définitions pour des fonctions auxiliaires

Définitions des fonctions privées; nous y placons celles des fonctions `norme`, `rotation`, `bordsDroite` (les seconds membres à compléter d'après les programmes déjà étudiés) :

```
In[] := | norme[u_?VectorQ] :=

In[] := | rotation[{α_, β_}, m:{_, _}] :=

In[] := | bordsDroite[range_,{a_,b_}] :=
```

Définitions pour les fonctions publiques

Nous y placons celles des fonctions `Ellipse`, `Hyperbole`, `Parabole` (les seconds membres à compléter d'après les programmes déjà étudiés) :

```
In[] := | Ellipse[a_?Positive, b_?Positive, c_List:{0, 0}, d_List:{1, 0},
        |     opts___Rule] :=

In[] := | Hyperbole[a_?Positive, b_?Positive, c_List:{0, 0}, d_List:{1, 0},
        |   opts___Rule] :=

In[] := | Parabole[p_?Positive, s_List:{0, 0}, d_List:{1, 0},
        |   opts___Rule] :=
```

Fin du contexte privé

```
In[] := | End[ ]
```

Fin du programme

Protection des symboles publics

N'évaluer cette cellule que lorsque tout est au point, au moment de construire le fichier de commandes; on la marquera à ce moment cellule d'initialisation.

```
In[] := | Protect[Ellipse,Hyperbole,Parabole,Placer,Droite]
```

Retour à l'environnement de travail normal

```
In[] := | EndPackage[ ]
```

Contrôle des contextes et symboles

Etat des contextes

Il peut être intéressant de voir l'état des variables de contextes et celui des symboles créés en cours de session.

La variable `$Context` donne le contexte courant, initalement `Global`, et qui est passé successivement à `Geometrie`Coniques`, `Geometrie`Coniques`Private`, `Geometrie`Coniques` pendant l'évaluation du programme précédent, puis est revenu à `Global` :

```
In[] := | $Context
```

La variable `$ContextPath` donne le chemin des contextes dans lesquels *Mathematica* peut reconnaître les symboles qu'il a créé :

```
In[] := | $ContextPath
```

Les symboles utilisateurs créés pendant la session courante (lorsque le programme ci-dessus a été évalué, le contexte`Global`` a été écarté du chemin de recherche, et ces symboles ont donc été protégés et sans effet).

Notons que le nouveau contexte créé par notre programme est ajouté au chemin de recherche, donnant ainsi accès aux symboles correspondants.

```
In[] := | ?Global`*
```

Les symboles publics de notre programme (les symboles privés s'otiendraient par ?Geometrie`Coniques`Private`*) :

```
In[] := | ?Geometrie`Coniques`*
```

Réinitialisation

On supprime tous les symboles publics et privés créés par notre programme, et on enlève le contexte correspondant du chemin de recherche; il ne reste plus de trace de ce programme, que l'on peut réévaluer sans provoquer de conflits.

```
In[] := | Remove["Geometrie`Coniques`*"]

In[] := | Remove["Geometrie`Coniques`Private`*"]

In[] := | $ContextPath = Rest[$ContextPath]
```

Evaluer l'initialisation.

Exemples d'utilisation

Exemples et tests

Fin du cahier Coniques.nb

4 - Réduction des coniques

Nous pouvons encore enrichir notre fichier de commandes de fonctions permettant par exemple de représenter une conique (et plus généralement d'en donner toutes les caractéristiques géométriques) à partir de son équation cartésienne. Nous allons expliquer un certain nombre de ces commandes, laissant au lecteur le soin de réaliser un fichier de commandes complet en complétant le précédent..

Pour illustrer les calculs qui suivent, on utilisera les équations ci-dessous :

```
In[] := | expr1 = 13x^2-32x y+37y^2-2x+14y-5

Out[] = | -5 - 2 x + 13 x^2 + 14 y - 32 x y + 37 y^2

In[] := | expr2 = (2x+3y)^2+4x-5y+7

Out[] = | 7 + 4 x - 5 y + (2 x + 3 y)^2
```

4.1 - Fonctions préliminaires

```
In[] := normer[u_List] := u/Sqrt[Expand[u.u]]
```

Matrice de la forme quadratique associée:

```
In[] := matrice[p_,{x_,y_}] := With[{ep = Expand[p]},
        {{Coefficient[ep,x^2],Coefficient[ep,x y]/2},
        {Coefficient[ep,x y]/2,Coefficient[ep,y^2]} }]

In[] := matrice[expr1,{x,y}] // MatrixForm

Out[] = ( 13   -16 )
        ( -16   37 )

In[] := matrice[expr2,{x,y}] // MatrixForm

Out[] = ( 4  6 )
        ( 6  9 )
```

Réduisons de cette matrice (toujours diagonalisable avec 2 valeurs propres réelles); la fonction `Eigensystem` donne la liste des valeurs propres et celle des vecteurs propres correspondants :

```
In[] := Eigensystem[%%]

Out[] = {{5, 45}, {{2, 1}, {-1, 2}}}

In[] := Eigensystem[%%]

Out[] = {{0, 13}, {{-3, 2}, {2, 3}}}
```

Si une valeur propre est nulle, ce sera une parabole; on place cette valeur propre nulle en premier pour tomber sur une équation de la forme $a y^2 + b x + c y + d = 0$. Sinon, la conique est à centre, et l'ordre des valeurs propres est indifférent; si un vecteur propre est colineaire au premier vecteur de base, on le place en première position. Voici le programme qui calcule ainsi la matrice orthogonale de changement de base :

```
In[] := Clear[pmatrice]
        pmatrice[sys_] := Module[ {valp, vecp},
            {valp,vecp} = sys;
            If[N[valp[[2]]] == 0.,
                vecp = Reverse[vecp]; valp = Reverse[valp] ];
            If[N[Times @@ valp] != 0. && N[vecp[[2,2]]] == 0.,
                vecp = Reverse[vecp]; valp = Reverse[valp] ];
            Transpose[normer /@ vecp] ]
```

Utilisation:

```
In[] := pmat1 = pmatrice[Eigensystem[matrice[expr1,{x,y}]]]

Out[] = {{2/Sqrt[5], -1/Sqrt[5]}, {1/Sqrt[5], 2/Sqrt[5]}}

In[] := pmat2 = pmatrice[Eigensystem[matrice[expr2,{x,y}]]]

Out[] = {{-3/Sqrt[13], 2/Sqrt[13]}, {2/Sqrt[13], 3/Sqrt[13]}}
```

4.2 - Réduction : étude sur des exemples

Les formules correspondantes :

```
In[] := formules1 = Thread[{x,y} -> pmat1 . {x,y}]
```

$$\text{Out[]} = \left\{x \to \frac{2x}{\sqrt{5}} - \frac{y}{\sqrt{5}},\ y \to \frac{x}{\sqrt{5}} + \frac{2y}{\sqrt{5}}\right\}$$

```
In[] := formules2 = Thread[{x,y} -> pmat2 . {x,y}]
```

$$\text{Out[]} = \left\{x \to -\frac{3x}{\sqrt{13}} + \frac{2y}{\sqrt{13}},\ y \to \frac{2x}{\sqrt{13}} + \frac{3y}{\sqrt{13}}\right\}$$

et la première équation réduite (on distingue la conique à centre de la parabole):

```
In[] := red1 = Expand[expr1 /. formules1]
```

$$\text{Out[]} = -5 + 2\sqrt{5}\,x + 5x^2 + 6\sqrt{5}\,y + 45y^2$$

```
In[] := red2 = Expand[expr2 /. formules2]
```

$$\text{Out[]} = 7 - \frac{22x}{\sqrt{13}} - \frac{7y}{\sqrt{13}} + 13y^2$$

Il reste à centrer la conique; si elle est à centre, celui-ci annule les dérivées partielles du premier membre de l'équation. Si c'est une parabole, il faut envisager les cas de dégénerescence et traiter au cas par cas.

4.3 - Réduction : le programme

Le programme ci-dessous doit être entré dans une seule cellule :

```
In[] := EquationReduite[expr_,{x_,y_}] := Module[
            {sys = Eigensystem[matrice[expr,{x,y}]],
                pmat, centreQ, centre, formules, red,
                cyy,cx,cy,c0,sommet
            },
            pmat = pmatrice[sys];
            centreQ = N[Times @@ sys[[1]]] != 0.;

        If[centreQ,
                (* Si oui: conique à centre *)
                centre = {x,y} /.
                    Solve[{D[expr,x] == 0,D[expr,y] == 0},{x,y}][[1]];
                formules = Thread[{x,y} -> pmat . {x,y} + centre];
                red = Expand[expr /. formules];
                {red,formules},
                (* Si non: famille parabole *)
                red = Expand[expr /. Thread[{x,y} -> pmat . {x,y}]];
                cyy = Coefficient[red,y^2];
                cy = Coefficient[red,y];
                cx = Coefficient[red,x];
                c0 = (red /. {x -> 0, y -> 0});
```

```
        If[N[cx] == 0, (* parabole dégénérée *)
                    sommet = {0,-cy/(2 cyy)};
                    formules = Thread[{x,y} ->
                        Expand[pmat . ({x,y} + sommet)]];
                    red = Expand[expr /. formules];
                    {red,formules},
                    (* Vraie parabole *)
                    sommet = {cy^2/(4 cyy cx) - c0/cx,-cy/(2 cyy)};
                    formules = Thread[{x,y} ->
                        Expand[pmat . ({x,y} + sommet)]];
                    red = Expand[expr /. formules];
                    {red,formules}
                ] ] ]
```

In[] := `expr = (2x - y)^2 + (6x + y)^2  + 2x - y - 3`

Out[] = $-3 + 2x + (2x - y)^2 - y + (6x + y)^2$

In[] := `EquationReduite[expr,{x,y}]`

Out[] =

$$\left\{-\frac{13}{4} + \frac{1885\,x^2}{\frac{377}{8} - \frac{19\sqrt{377}}{8}} - \frac{97\sqrt{377}\,x^2}{\frac{377}{8} - \frac{19\sqrt{377}}{8}} + \frac{1885\,y^2}{\frac{377}{8} + \frac{19\sqrt{377}}{8}} + \frac{97\sqrt{377}\,y^2}{\frac{377}{8} + \frac{19\sqrt{377}}{8}},\right.$$

$$\left\{x \to -\frac{1}{16} + \frac{\left(19 - \sqrt{377}\right)x}{4\sqrt{\frac{377}{8} - \frac{19\sqrt{377}}{8}}} + \frac{\left(19 + \sqrt{377}\right)y}{4\sqrt{\frac{377}{8} + \frac{19\sqrt{377}}{8}}},\right.$$

$$\left.\left.y \to \frac{3}{8} + \frac{x}{\sqrt{\frac{377}{8} - \frac{19\sqrt{377}}{8}}} + \frac{y}{\sqrt{\frac{377}{8} + \frac{19\sqrt{377}}{8}}}\right\}\right\}$$

In[] := `Simplify[%[[1]]]`

Out[] = $\frac{1}{4}\left(-13 - 4\left(-21 + \sqrt{377}\right)x^2 + 4\left(21 + \sqrt{377}\right)y^2\right)$

In[] := `expr /. %%[[2]] // Expand`

Out[] =

$$-\frac{13}{4} + \frac{1885\,x^2}{\frac{377}{8} - \frac{19\sqrt{377}}{8}} - \frac{97\sqrt{377}\,x^2}{\frac{377}{8} - \frac{19\sqrt{377}}{8}} + \frac{1885\,y^2}{\frac{377}{8} + \frac{19\sqrt{377}}{8}} + \frac{97\sqrt{377}\,y^2}{\frac{377}{8} + \frac{19\sqrt{377}}{8}}$$

In[] := `EquationReduite[expr1, {x, y}]`

Out[] = $\left\{-7 + 5x^2 + 45y^2, \left\{x \to -\frac{1}{3} + \frac{2x}{\sqrt{5}} - \frac{y}{\sqrt{5}}, y \to -\frac{1}{3} + \frac{x}{\sqrt{5}} + \frac{2y}{\sqrt{5}}\right\}\right\}$

In[] := `EquationReduite[expr2, {x, y}]`

Out[] = $\left\{-\frac{22x}{\sqrt{13}} + 13y^2, \left\{x \to -\frac{13433}{14872} - \frac{3x}{\sqrt{13}} + \frac{2y}{\sqrt{13}}, y \to \frac{5145}{7436} + \frac{2x}{\sqrt{13}} + \frac{3y}{\sqrt{13}}\right\}\right\}$

Nous pouvons poursuivre et construire une fonction qui calcule toutes les caractéristiques géométriques de la conique puis la dessine. Nous laissons le lecteur intéressé compléter lui-même ce travail.

Chapitre 4

Les graphiques en dimension 3

1 - Les primitives graphiques 3D

1.1 - Retour sur la structure des graphiques 3D

- Rappelons les deux catégories de types graphiques représentant des figures dans l'espace :
  - Les types `SurfaceGraphics`, `ContourGraphics`, `DensityGraphics` qui représentent une surface $z = f(x, y)$ à partir d'une matrice d'altitudes $(z_{i,j})$; nous n'y reviendrons pas (voir chapitre 1).
  - Le type `Graphics3D` qui représente des figures de l'espace à partir de primitives et directives graphiques. Rappelons que les types précédents peuvent se convertir en type `Graphics3D` par l'intermédiaire du type `SurfaceGraphics` (voir chap.1, 3.1).

Un objet graphique 3D est une expression de type `Graphics3D`, à deux arguments qui sont des listes :

```
In[] := | Graphics3D[{ < primitives et directives >}, { < options >}]
```

Les principes régissant l'action des directives graphiques sur les primitives sont analogues à ceux concernant les graphiques 2D :

– Certaines directives n'agissent que sur des primitives déterminées (`PointSize` n'agit que sur la primitive `Point`, `Thickness` sur la primitive `Line`).

– Les directives de couleur agissent sur les primitives `Point`, `Line`, mais leur action sur les primitives `Polygon`, `Cuboid` n'est visible que dans certaines conditions (pas d'éclairage simulé - voir 2.3).

– Une directive n'agit que sur les primitives placées dans la même liste ou dans des sous-listes.

– Une directive agit sur toutes les primitives placées après elle dans la liste où elle figure; son effet est annulé par la présence de la même directive placée après elle dans la liste où elle figure.

Contrairement à la fonction `Graphics`, l'ordre dans lequel sont placées les primitives dans le premier argument de `Graphics3D` n'influence pas le rendu de la figure :

– Les polygones sont opaques; leur rendu dépend de la projection de la figure (point de vue) : ceux situés devant cachent ceux situés derrière (voir 2.2). Lorsque la figure est rendue à l'écran, les polygones situés derrière sont rendus en premier.

– Le texte (primitive `Text`) est toujours visible.

1.2 - Les primitives graphiques 3D

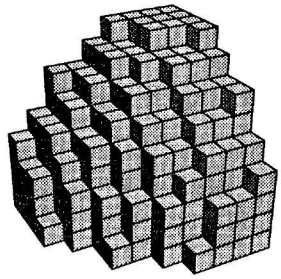
- **Point :** `Point[{x, y, z}]` représente un point de coordonnées $\{x, y, z\}$; ce point est dessiné comme une petite tache.
- **Line :** `Line[{{x1, y1, z1}, {x2, y2, z2}, ..., {xn, yn, zn}}]` représente une ligne brisée joignant les points de coordonnées $\{x_k, y_k, z_k\}$.
- **Polygon :** `Polygon[{{x1, y1, z1}, {x2, y2, z2}, ..., {xn, yn, zn}}]` représente un polygone plein (le dernier sommet est joint au premier). S'il y a plus de trois points, le polygone sera en général gauche : selon le mode d'éclairage et s'il est assez grand, il sera représenté "triangulé".

- **Cuboid** : `Cuboid[{`x_{min}`,` y_{min}`,` z_{min}`}, {`x_{max}`,` y_{max}`,` z_{max}`}]` représente un parallélépipède plein dont les faces sont parallèles aux plans de coordonnées (analogue dans l'espace de la primitive `Rectangle` dans le plan).
- **Text :** `Text["`*texte*`",{`*x*`, `*y*`, `*z*`}]` représente la chaîne de caractères "*texte*" centrée au point de coordonnées {*x*, *y*, *z*}; `Text["`*texte*`",{`*x*`, `*y*`, `*z*`}, {dx, dy}]` décale le texte de {dx, dy} (l'offset est calculé dans le plan de projection). Des options (dont `TextStyle, FormatType`) permettent de choisir le style du texte.

Ci-dessous, on approche par défaut un huitième de sphère par des cubes et on les compte :

```
In[] := cpt = 0;
        gr1 = Show[Graphics3D[Table[If[i^2 + j^2 + k^2 <= 100,
              cpt++; Cuboid[{i - 1, j - 1, k - 1}, {i, j, k}], {}],
            {i, 10}, {j, 10}, {k, 10}], ViewPoint -> {1.4, 2.6, 1.5},
            Boxed -> False]]; cpt
```

-Figure 1-

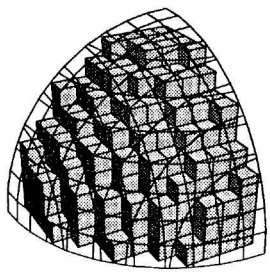


```
Out[] = 410

In[] := N[4/3 π 10^3 / 8]

Out[] = 523.599

In[] := << Graphics`ContourPlot3D`

In[] := gr2 = ContourPlot3D[x^2 + y^2 + z^2 - 100, {x, 0, 10}, {y, 0, 10},
          {z, 0, 10}, ViewPoint -> {1.4, 2.6, 1.5}, Boxed -> False];
```

(Figure non montrée)

```
In[] := Show[gr1, gr2 /. Polygon[s_] :> Line[Append[s, First[s]]]];
```

-Figure 2-



Exercice 1 :

Dessiner un escalier en colimaçon autour de l'axe des z formé de n cubes de côté unité.

1.3 - Les directives graphiques 3D

- On retrouve les mêmes directives de dimensions que dans les graphiques 2D :

PointSize, AbsolutePointSize, Thickness, AbsoluteThickness, Dashing, AbsoluteDashing

- On retrouve également les mêmes directives de couleurs :

GrayLevel, RGBColor, Hue, CMYKColor

On dispose en outre de directives composées permettant de faire agir d'autres directives sur les arêtes et les faces des polygones :

– **EdgeForm** : `EdgeForm[{`d_1`, `d_2`, ...}]` applique les directives $d_1, d_2, \ldots$ aux arêtes des polygones; `EdgeForm[]` efface les arêtes des polygones.

– **FaceForm** : `FaceForm[{`d_1`, `d_2`, ...}, {`s_1`, `s_2`, ...}]` a deux arguments : les directives $d_1, d_2, \ldots$ s'appliqueront aux faces positives des polygones, et les directives $s_1, s_2, \ldots$ aux faces négatives. Ce seront des directives de couleurs ou `SurfaceColor`. L'orientation des faces d'un polygone est déterminée par l'ordre dans lequel sont spécifiés les trois premiers sommets S_1, S_2, S_3 dans `Polygon[{`S_1`, `S_2`, `S_3`, ...}]`.

– **SurfaceColor** : Lorsqu'un polygone est éclairé par des sources de lumière simulée (voir plus loin, 2.3), les directives de couleurs introduites directement n'ont pas d'effet; les faces sont considérées comme blanches et reflètent la lumière incidente. `SurfaceColor[d]` où `d` est une directive de couleur, modifie le pouvoir reflecteur du polygone (en fonction du, ou des arguments de `d`); si `d` est une directive autre que `GrayLevel`, cela a pour effet d'attribuer à la surface une couleur propre. La réflexion paramétrée par le premier argument est une réflection diffuse; deux autres arguments de `SurfaceColor` permettent d'affecter à la surface un pouvoir de réflexion spéculaire, pour lui donner un aspect plus ou moins brillant.

Dans l'expérience ci-dessous, on anticipe sur l'étude des options (voir 2.2 et 2.3).

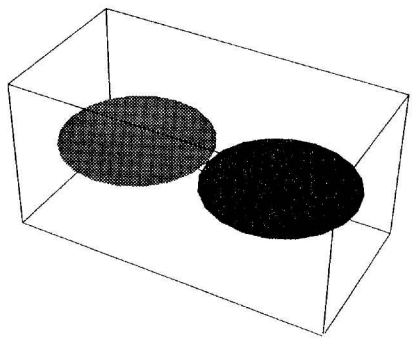
La fonction disque trace un disque dans le plan (xoy).

```
In[] := disque[c : {_, _, 0}, r_, orient_: 1] := Polygon[
          Table[c + r {Cos[orient * t], Sin[orient * t], 0}, {t, 0, 2 π, π/10}]]
```

On montre deux disques orientés différemment, coloriés par FaceForm, sans éclairage simulé :

```
In[] := Show[Graphics3D[{EdgeForm[], FaceForm[RGBColor[1, 0, 0],
          RGBColor[0, 0, 1]], disque[{0, 1, 0}, 1],
          disque[{2, 1, 0}, 1, -1]}], Lighting -> False];
```

-Figure 3-



On dessine maintenant 5 disques auxquels on attache des propriétés réfléchissantes différentes; ces disques sont observés de face, et éclairés à 45 degrés par une source de lumière blanche.

On fait varier les coefficients de reflexions diffuse et spéculaire; le lecteur pourra faire d'autres essais.

```
In[] := Show[Graphics3D[Table[{SurfaceColor[RGBColor[(i - 1)/4, 0, 0]],
          disque[{i, 1/2, 0}, 1/2]}, {i, 5}]],
        Boxed -> False, ViewPoint -> {0, 0, 5},
        LightSources -> {{{0, -1, 1}, RGBColor[1, 1, 1]}}];
```

(Figure non montrée)

```
In[] := Show[Graphics3D[
          Table[{SurfaceColor[GrayLevel[0], RGBColor[1, 0, 0], i],
            disque[{i, 1/2, 0}, 1/2]}, {i, 5}]],
        Boxed -> False, ViewPoint -> {0, 0, 5},
        LightSources -> {{{0, -1, 1}, RGBColor[1, 1, 1]}}];
```

(Figure non montrée)

```
In[] := Show[Graphics3D[Table[
          {SurfaceColor[RGBColor[(i - 1)/4, 0, 0], RGBColor[0, 0, (5 - i)/4], 2],
           disque[{i, 1/2, 0}, 1/2]}, {i, 5}]],
        Boxed -> False, ViewPoint -> {0, 0, 2},
        LightSources -> {{{0, 1, 1}, RGBColor[1, 1, 1]}}];
```

(Figure non montrée)

On montre enfin 5 disques ayant les mêmes propriétés réfléchissantes en faisant varier l'angle d'incidence. Dans le premier cas (100% de reflexion diffuse), la lumière réfléchie est proportionnelle à $\cos(\theta)$ (θ angle d'incidence). Dans le second cas (100% de reflexion spéculaire d'exposant 2), la lumière réfléchie est proportionnelle à $\cos^2(\theta)$. (Sur les figures ci-dessous, la couleur rouge clair est rendue par du blanc; exécuter les commandes à la machine pour pouvoir observer les vraies couleurs).

```
In[] := Show[GraphicsArray[Table[Graphics3D[
          {SurfaceColor[RGBColor[1, 0, 0]], disque[{i, 1/2, 0}, 1/2]},
        Boxed -> False, ViewPoint -> {0, 0, 5},
          LightSources -> {{{0, 1 - i, 1}, GrayLevel[1]}}], {i, 5}]]];
```

-Figure 4-

```
In[] := Show[GraphicsArray[Table[Graphics3D[{SurfaceColor[GrayLevel[0],
            RGBColor[1, 0, 0], 2], disque[{i, 1/2, 0}, 1/2]},
        Boxed -> False, ViewPoint -> {0, 0, 5},
          LightSources -> {{{0, 1 - i, 1}, GrayLevel[1]}}], {i, 5}]]];
```

-Figure 5-

2 - Les options graphiques 3D

2.1 - Les options générales

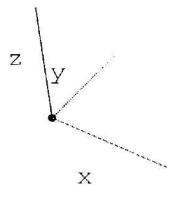
- On retrouve avec les graphiques 3D certaines options déjà rencontrées avec les graphiques 2D :
  - **DisplayFunction** pour préciser la sortie du graphique.
  - **ColorOutput** pour convertir les sorties graphiques (on précise en argument une directive de couleur).
  - **PlotLabel** pour donner un titre au graphique.
  - **AspectRatio** qui s'applique ici aux proportions de la projection du graphique (et non à celles de la boite contenant le graphique); sa valeur par défaut est `Automatic`.
  - **PlotRange** pour préciser la portion de graphique à représenter; les valeurs possibles sont `Automatic`, `All`, $\{z_{min}, z_{max}\}$ et $\{\{x_{min}, x_{max}\}, \{y_{min}, y_{max}\}, \{z_{min}, z_{max}\}\}$.
  - **PlotRegion** pour préciser en coordonnées relatives la portion $\{\{x_{min}, x_{max}\}, \{y_{min}, y_{max}\}\}$ de la zone d'affichage que doit occuper le graphique (par défaut toute le zone : {{0, 1}, {0, 1}}).
  - **DefaultColor**, **Background** pour la couleur par défaut des lignes et du fond.
  - **FormatType**, **TextStyle** pour le format et le style du texte.
  - **Prolog, Epilog** pour introduire des primitives 2D (nécessairement en coordonnées relatives).
- Un graphique 3D peut être encadré d'une boite "fil de fer", et muni d'axes de coordonnées.
  - L'option **Boxed** (valeur par défaut : `True`) montre ou cache la boite; le style des arêtes est précisé par l'option **BoxStyle**, et ses proportions par l'option **BoxRatios**, même si la boite n'est pas tracée : en fait cette option est l'analogue de `AspectRatio` pour les graphiques 2D (pour le type `SurfaceGraphics`, la valeur par défaut est {1, 1, 0.4} et pour `Graphics3D`, elle est `Automatic`, qui correspond à un repère orthonormé). Enfin l'option **FaceGrids** permet de dessiner des quadrillages sur les faces de la boite.
  - L'option **Axes** montre ou non les axes. On peut donner des valeurs séparées pour chaque axe comme `{True, False,True}`. Le style des axes est précisé par **AxesStyle**, et l'option **AxesEdge** permet de choisir sur quelles arêtes de la boite (visible ou non) placer ces axes (par défaut, ils sont placés pour ne pas être cachés par le graphique). Enfin l'option **Ticks** est également disponible.

La syntaxe des options `AxesEdge, FaceGrids, Ticks` est assez compliquée; nous invitons le lecteur à effectuer des essais en s'aidant de la documentation.

Rappelons qu'avec la valeur par défaut du point de vue ({1.3, −2.4, 2}), l'axe des x est horizontal de gauche à droite, l'axe des y horizontal d'avant en arrière, l'axe des z vertical de bas en haut :

```
In[] := Show[Graphics3D[{PointSize[1 / 25], Point[{0, 0, 0}]},
        Boxed -> False, PlotRange -> {{0, 1}, {0, 1}, {0, 1}},
        Axes -> True, AxesEdge -> {{-1, -1}, {-1, -1}, {-1, -1}},
        AxesStyle -> {{RGBColor[1, 0, 0]}, {RGBColor[0, 1, 0]},
        {RGBColor[0, 0, 1]}}, AxesLabel -> {"x", "y", "z"}, Ticks -> None]];
```

-Figure 6-



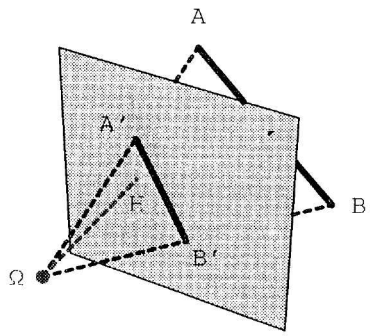
2.2 - Le système de projection de la figure

L'image graphique sur l'écran ou sur la feuille imprimée, est une projection de l'objet spacial que l'on veut représenter. Dans *Mathematica*, il s'agit d'une projection centrale : les points de l'image sont les intersections avec le plan de projection, des droites qui joignent les points de l'objet spacial à un même point appelé centre de la projection, ou point de vue.

```
In[] := a = {0, 7, 2}; b = {3, 9, -3}; y = 4; pv = {0, 0, 0};
        ap = {y a[[1]] / a[[2]], y, y a[[3]] / a[[2]]};
        bp = {y b[[1]] / b[[2]], y, y b[[3]] / b[[2]]};
```

```
In[] := Show[Graphics3D[
          {Thickness[1 / 120], {RGBColor[1, 0, 0], PointSize[0.03],
            Point[pv], Dashing[{1 / 75, 1 / 75}], Line[{pv, {0, y, 0}}],
            Text["H", {0, y, 0}, {0, 1.5}], Text["Ω", pv, {2, 0}]},
          Polygon[{{-2, y, -3}, {4, y, -3}, {4, y, 3}, {-2, y, 3}}],
          {Thickness[1 / 75], Line[{a, b}], Line[{ap, bp}]},
          Text["A", a, {0, -2}], Text["B", b, {-2, 0}],
          {Dashing[{1 / 75, 1 / 75}], Line[{a, pv, b}]},
          Text["A'", ap, {1, -1}], Text["B'", bp, {-1, 1}]
          }], Boxed -> False, PlotRange -> All];
```

-Figure 7-



Le système de projection est déterminé par les deux options **ViewPoint** et **ViewCenter** qui définissent deux points Ω et H; le premier est le centre de la projection, et le plan de projection est le plan passant par H et perpendiculaire à la direction (ΩH).

Les coordonnées de `ViewPoint` sont relatives à un repère centré au centre de la boite, l'unité étant la plus grande des 3 dimensions de la boite.

Les coordonnées de `ViewCenter` sont relatives à un repère centré au coin inférieur gauche de la boite, le coin opposé (supérieur droit) ayant pour coordonnées {1, 1, 1}..

Il existe deux options supplémentaires pour préciser encore la projection de l'image :

ViewVertical, qui permet de faire tourner le plan de projection autour de l'axe (ΩH).

SphericalRegion, qui, lorsque sa valeur est `True`, réduit la figure pour y inclure la projection d'une sphère circonscrite (utile pour des animations faisant varier le point de vue).

- Exercice 2 :

  Dessiner un cube et étudier ses différentes projections en faisant varier le point de vue et le plan de projection (par combinaison des options `ViewPoint` et `ViewCenter`).

Un exemple pour se familiariser avec le sélecteur de point de vue :

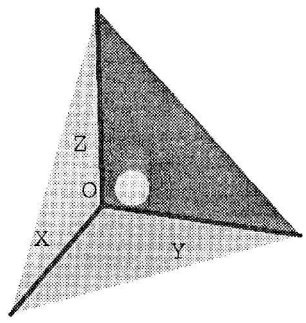
Un exemple pour se familiariser avec le sélecteur de point de vue :

La figure représente un repère dans l'espace (O,X,Y,Z); la boule jaune est placée à l'intérieur du repère, pour le distinguer de l'extérieur et aider à se représenter la position de la figure dans l'espace.

En faisant varier le point de vue à l'aide du sélecteur, on se familiarise avec l'effet produit par les différentes valeurs de l'option.

```
In[] := Show[Graphics3D[
          {EdgeForm[], FaceForm[{}, SurfaceColor[GrayLevel[0.5]]],
           Polygon[{{0, 0, 0}, {1, 0, 0}, {0, 1, 0}}],
           Polygon[{{0, 0, 0}, {0, 0, 1}, {1, 0, 0}}],
           Polygon[{{0, 0, 0}, {0, 1, 0}, {0, 0, 1}}],
           {Thickness[1 / 75], PointSize[1 / 50], RGBColor[1, 0, 0],
            Point[{0, 0, 0}], Line[{{0, 0, 0}, {1, 0, 0}}],
            Line[{{0, 0, 0}, {0, 1, 0}}], Line[{{0, 0, 0}, {0, 0, 1}}]},
           {RGBColor[1, 1, 0], PointSize[1 / 10], Point[{1 / 4, 1 / 4, 1 / 4}]},
           Text["O", {0, 0, 0}, {1, -1}], Text["X", {1 / 2, 0, 0}, {1.5, -1}],
           Text["Y", {0, 1 / 2, 0}, {1.5, 1.5}],
           Text["Z", {0, 0, 1 / 2}, {1.5, 1.5}]
          }, Boxed -> False],
         < contenu du presse - papier >];
```

-Figure 8-



2.3 - L'éclairage des polygones

La couleur des primitives graphiques `Point`, `Line`, `Text` est déteminée par les directives de couleurs présentes dans le premier argument de la fonction `Graphics3D`, comme pour les graphiques en deux dimensions. Pour les primitives `Polygon` et `Cuboid`, par contre, on peut soit les colorier à l'aide de directives comme précédemment, soit les soumettre à un système d'éclairage simulé; c'est l'option `Lighting` qui permet de choisir entre les deux modes.

2.3.1 - Couleurs des polygones sans éclairage simulé

L'option `Lighting` est mise à `False`. Les polygones sont alors coloriés par les directives présentes dans l'expression graphique; par défaut ils sont noirs, comme les autres primitives.

Remarquons que certaines commandes génèrent automatiquement ces directives; par exemple :

– L'option `ColorFunction` des objets de type `SurfaceGraphics` permet de colorier les polygones en fonction de l'altitude (il faut convertir ces objets au type `Graphics3D` pour le voir).

– Les arguments optionnels des fonctions `ListPlot3D`, `Plot3D`, `ParametricPlot3D` permettent d'introduire des directives de couleurs par l'intermédiaire de matrices (pour la première) ou de fonctions (pour les deux suivantes).

2.3.2 - Couleurs des polygones avec éclairage simulé

L'option `Lighting` est mise à `True`. Les directives de couleur présentes dans l'expression n'ont pas d'effet sur les polygones, sauf la directive `SurfaceColor`. Leur couleur est déterminée :

– par la composition spectrale des sources de lumière (options `LightSources` et `AmbientLight`)

– par la directive `SurfaceColor`, qui précise le pouvoir réflecteur des surfaces en fonction des différentes radiations (voir 1.3)

Les coordonnées des sources de lumières sont relatives à un système lié au plan de projection de la figure : abscisses et ordonnées horizontale et verticale dans de plan, cotes perpendiculaires et orientées vers l'avant. Seule la direction des sources de lumière est pertinente.

```
In[] := Options[Graphics3D, LightSources]

Out[] = {LightSources → {{{1., 0., 1.}, RGBColor[1, 0, 0]},
          {{1., 1., 1.}, RGBColor[0, 1, 0]},
          {{0., 1., 1.}, RGBColor[0, 0, 1]}}}

In[] := Options[Graphics3D, AmbientLight]

Out[] = {AmbientLight → GrayLevel[0]}
```

Exercice 3 :

Simuler l'éclairage d'une sphère par les lumières successives d'une rampe circulaire de lumières colorées selon les couleurs de l'arc en ciel.

3 - Manipulation de polyèdres

Comme exemple d'utilisation de fichiers de commandes et de programmation graphique dans l'espace, nous proposons quelques exercices de manipulation de polyèdres réguliers.

3.1 - Description du fichier Geometry`Polytopes`

Le fichier `Geometry`Polytopes`` est une base de données géométriques relative aux polygones réguliers et aux polyèdres réguliers.

```
In[] := << Geometry`Polytopes`
```

Concernant les polyèdres réguliers convexes (solides de Platon), chacun d'eux est désigné par un symbole: `Tetrahedron, Hexahedron, Octahedron, Dodecahedron, Icosahedron.`

On peut obtenir des renseignements sur un polyèdre en demandant des informations :

```
In[] := ?? Icosahedron

       Icosahedron is a polyhedron.

       Area[Icosahedron] ^= Sqrt[3] / 4
       CircumscribedRadius[Icosahedron] ^= Sqrt[10 + 2 * Sqrt[5]] / 4
       Dual[Icosahedron] ^= Dodecahedron
       Faces[Icosahedron] ^= {{1, 3, 2}, {1, 4, 3}, {1, 5, 4},
          {1, 6, 5}, {1, 2, 6}, {2, 3, 7}, {3, 4, 8}, {4, 5, 9},
          {5, 6, 10}, {6, 2, 11}, {7, 3, 8}, {8, 4, 9}, {9, 5, 10},
          {10, 6, 11}, {11, 2, 7}, {7, 8, 12}, {8, 9, 12},
         {9, 10, 12}, {10, 11, 12}, {11, 7, 12}}
       InscribedRadius[Icosahedron] ^= Sqrt[42 + 18 * Sqrt[5]] / 12
       NumberOfEdges[Icosahedron] ^= 30
```

(Suite page suivante)

```
NumberOfFaces[Icosahedron] ^= 20
NumberOfVertices[Icosahedron] ^= 12
Schlafli[Icosahedron] ^= {3, 5}
Vertices[Icosahedron] ^:= Icosahedron /: Vertices[Icosahedron] =
  N[N[Geometry`Polytopes`Private`SphericalToCartesian /@
        {{0, 0}, {ArcTan[2], 0}, {ArcTan[2], (2 * Pi) / 5},
         {ArcTan[2], (4 * Pi) / 5}, {ArcTan[2], (6 * Pi) / 5},
         {ArcTan[2], (8 * Pi) / 5}, {Pi - ArcTan[2], Pi / 5},
               {Pi - ArcTan[2], (3 * Pi) / 5},
         {Pi - ArcTan[2], (5 * Pi) / 5}, {Pi - ArcTan[2], (7 * Pi) / 5},
         {Pi - ArcTan[2], (9 * Pi) / 5}, {Pi, 0}} /
       (1 / 2 + Cos[ArcTan[2]] / 2) ^ (1 / 2), 2 * Precision[1.]]]
Volume[Icosahedron] ^= (5 * (3 + Sqrt[5])) / 12
```

Ces informations peuvent être complétées (d'après le traité de géométrie de E. Rouché et Ch. de Combrousse) :

```
In[] := | Diedre[Tetrahedron] ^= 2 ArcSin[Sqrt[3] / 3];
        | Diedre[Hexahedron] ^= Pi / 2;
        | Diedre[Octahedron] ^ 2 ArcSin[Sqrt[2] / Sqrt[3]];
        | Diedre[Dodecahedron] ^= 2 ArcSin[Sqrt[(5 + Sqrt[5]) / 10]];
        | Diedre[Icosahedron] ^= 2 ArcSin[(1 + Sqrt[5]) / (2 Sqrt[3])];

In[] := | ?? Icosahedron
```

(Sortie non montrée)

Notons que toutes les données ci-dessus, sauf celles fournies par la fonction `Vertices` se rapportent à un polyèdre régulier d'arête unité.

Deux polyèdres duaux (ou conjugués) sont tels que les sommets de l'un sont (à une homothétie près) les centres des faces de l'autre.

Le symbole de Shlafli d'un polyèdre est le couple (n,m) où n (resp. m) est le nombre de côtés de chaque face (resp. d'arêtes issues de chaque sommet); ces couples sont permutés pour deux polyèdres duaux.

3.2 - Représentation graphique des polyèdres réguliers convexes

Le fichier de commandes `Graphics`Polyhedra`` permet de représenter graphiquement les polyèdres réguliers et offre également différents outils de représentations graphiques.

La fonction `Polyhedron` construit un objet graphique à partir des symboles représentant les différents polyèdres; il suffit d'utiliser `Show` pour les tracer à l'écran.

```
In[] := | << Graphics`Polyhedra`
        | Show[Polyhedron[Icosahedron]]
```

(Figure non montrée)

Les quatres polyèdres réguliers d'espèce supérieure à 1 dont nous allons parler plus loin sont également représentés. Nous n'utiliserons pas ce fichier ici, mais conseillons vivement au lecteur d'en étudier la programmation; ce qui suit peut d'ailleurs servir d'introduction à une telle étude.

Revenons à notre fichier `Geometry`Polytopes``. La fonction `Vertices` fournit la liste des coordonnées des sommets du polyèdre dans un repère centré au centre du polyèdre; l'octaèdre et l'icosaèdre ont deux sommets sur l'axe des cotes et leurs duaux (cube, dodécaèdre) deux faces perpendiculaires à cet axe; le tétraèdre est présenté avec un sommet sur cet axe.

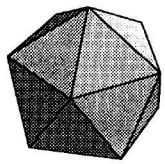
Si *poly* est un polyèdre à n sommets, en supposant que les éléments de la liste `Vertices[`*poly*`]` soient numérotés de 1 à n, `Faces[`*poly*`]` donne la liste des numéros des sommets constituant les faces.

```
In[] := sommets = Vertices[Icosahedron]

Out[] = {{0, 0, 1.17557}, {1.05146, 0, 0.525731},
  {0.32492, 1., 0.525731}, {-0.850651, 0.618034, 0.525731},
  {-0.850651, -0.618034, 0.525731}, {0.32492, -1., 0.525731},
  {0.850651, 0.618034, -0.525731}, {-0.32492, 1., -0.525731},
  {-1.05146, 0, -0.525731}, {-0.32492, -1., -0.525731},
  {0.850651, -0.618034, -0.525731}, {0, 0, -1.17557}}

In[] := faces = Faces[Icosahedron]

Out[] = {{1, 3, 2}, {1, 4, 3}, {1, 5, 4}, {1, 6, 5}, {1, 2, 6}, {2, 3, 7},
  {3, 4, 8}, {4, 5, 9}, {5, 6, 10}, {6, 2, 11}, {7, 3, 8},
  {8, 4, 9}, {9, 5, 10}, {10, 6, 11}, {11, 2, 7}, {7, 8, 12},
  {8, 9, 12}, {9, 10, 12}, {10, 11, 12}, {11, 7, 12}}
```

Voici comment on construit l'objet graphique représentant l'icosaedre à partir de ces données :

```
In[] := SetOptions[Graphics3D, Boxed -> False];

In[] := ico = Show[Graphics3D[
          Polygon[sommets[[#]]]& /@ faces] ];
```

-Figure 9-



Il peut être commode de voir sur la figure les numéros des sommets représentés :

```
In[] := numsom = MapThread[Text[#1, #2, {1, 1}]&, {Range[12], sommets}];

In[] := iconum1 = Show[ico, Graphics3D[numsom],
          TextStyle -> {FontWeight -> "Bold", FontSize -> 12}];
```

(Figure non montrée)

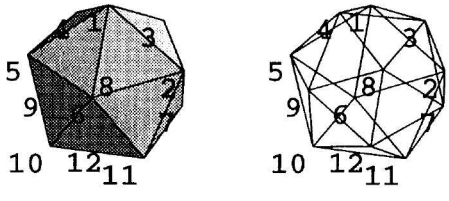
La même figure en "fil de fer" peut être utile :

```
In[] := iconum2 = Show[ico /. Polygon -> Line, Graphics3D[numsom],
          TextStyle -> {FontWeight -> "Bold", FontSize -> 12}];
```

(Figure non montrée)

```
In[] := Show[GraphicsArray[{iconum1, iconum2}]];
```

-Figure 10-



On pourra aussi représenter chaque graphique en modifiant le point de vue si on le désire.

3.3 - Les polyèdres réguliers d'espèces supérieures

Si on abandonne pour les polyèdres réguliers l'hypothèse de convexité, ne gardant que les conditions selon lesquelles les faces sont des polygones réguliers égaux entre eux (convexes ou croisés) faisant entre elles des angles dièdres égaux, alors il existe quatre polyèdres réguliers supplémentaires.

On démontre que ces polyèdres ont nécéssairement les mêmes sommets qu'un polyèdre régulier convexe, qui ne peut être que le dodécaèdre ou l'icosaèdre, les faces étant obtenues en joignant ces sommets de façon différente de celle qui mène au polyèdre convexe.

Lorsqu'on coupe la sphère circonscrite à un polyèdre régulier par les angles polyèdres de sommet le centre de cette sphère et dont les arêtes passent par les sommets des faces, on obtient autant de polygones sphériques réguliers (égaux entre eux) que de faces du polyèdre. Ces polygones sphériques recouvrent (en son entier) la sphère circonscrite un certain nombre de fois appelé l'espèce du polyèdre régulier. Un polyèdre convexe est évidemment de première espèce.

3.3.1 - Les polyèdres d'espèce supérieure à 1 à faces convexes

Les polyèdres d'espèce supérieure à 1 qui ont des faces convexes s'obtiennent tous à partir des sommets de l'icosaèdre régulier convexe. Il y a deux façons d'obtenir ainsi un nouveau polyèdre régulier:

- Le dodécaèdre régulier à faces convexes de troisième espèce.

- L'icosaèdre régulier de septième espèce.

a) - Dodécaèdre régulier de troisième espèce à faces convexes

Les faces de ce polyèdre sont obtenues en joignant les sommets de l'icosaèdre qui sont les extrémités des arêtes issues d'un même sommet. Ce sont des pentagones réguliers convexes.

Par exemple au sommet 6 correspond la face joignant les sommets {1,2,11,10,5}

Voici comment on peut construire la face de ce polyèdre correspondant au sommet *a* de l'icosaèdre :

```
In[] := | a = 3
Out[] = | 3
```

```
In[] := | fa = Select[faces, MemberQ[#, a]&]
Out[] = | {{1, 3, 2}, {1, 4, 3}, {2, 3, 7}, {3, 4, 8}, {7, 3, 8}}
```

On permute circulairement les sommets pour avoir *a* en seconde position :

```
In[] := | far = RotateLeft[#, Position[#, a][[1]] + 1]& /@ fa
Out[] = | {{1, 3, 2}, {4, 3, 1}, {2, 3, 7}, {8, 3, 4}, {7, 3, 8}}
```

Il faut réordonner les faces pour qu'elles soient consécutives (nous avons fait un programme structural) :

```
In[] := | Clear[consec]

        | consec[l_] := Module[{res = {First[l]}, lr = Rest[l]},
        | While[lr =!= {},
In[] := |  Do[If[res[[-1, -1]] == lr[[i, 1]],
        |    AppendTo[res, lr[[i]]]; lr = Drop[lr, {i}]; Break[]],
        |    {i, Length[lr]}] ]; res ]
```

```
In[] := | consec[far]
Out[] = | {{1, 3, 2}, {2, 3, 7}, {7, 3, 8}, {8, 3, 4}, {4, 3, 1}}
```

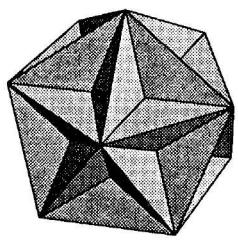
nous obtenons ainsi notre face :

```
In[] := | First /@%
Out[] = | {1, 2, 7, 8, 4}
```

et le programme :

```
In[] := | Clear[extraitFace]

In[] := | extraitFace[fl_, s_] :=
        | Map[First,
        |   consec[
        |    Map[
        |     RotateLeft[#, Position[#, s][[1]] + 1]& ,
        |     Select[faces, MemberQ[#, s]&]
        |     ] ] ]

In[] := | faces1 = extraitFace[faces, #]& /@ Range[12]

Out[] = | {{2, 3, 4, 5, 6}, {3, 1, 6, 11, 7}, {1, 2, 7, 8, 4}, {1, 3, 8, 9, 5},
        |  {1, 4, 9, 10, 6}, {1, 5, 10, 11, 2}, {3, 2, 11, 12, 8},
        |  {4, 3, 7, 12, 9}, {5, 4, 8, 12, 10}, {6, 5, 9, 12, 11},
        |  {2, 6, 10, 12, 7}, {8, 7, 11, 10, 9}}

In[] := | ico1 = Show[Graphics3D[Polygon[sommets[[#]]]& /@ faces1]];
```

-Figure 11-



Le programme est un peu long et il aurait été plus rapide d'écrire les numéros de faces à la main; nous avons cependant voulu soulever quelques problèmes de programmation.

Autre méthode de construction du dodécaèdre régulier de troisième espèce à faces convexes :

Le fichier `Graphics`Polyhedra`` construit ce polygone par une méthode de "stellation" qui consiste à remplacer chaque face par une pyramide dont le sommet peut être dirigé vers l'extérieur ou vers l'intérieur.

- Exercice 4 :

  Programmer une fonction `etoiler[poly,k]` qui remplace dans le polygone `poly` chaque face par une pyramide dont le sommet se déduit du centre de la face par une homothétie de centre O (centre du polygone) et de rapport `k`.

  Le dodécaèdre étoilé s'obtiendra en appliquant cette fonction à l'icosaèdre convexe avec $k = \frac{1}{\sqrt{2}}$.

Remarquons que la construction par la méthode de stellation est plus sûre, car le rendu de figures où des polygones se croisent n'est pas toujours parfaite.

b) - Icosaèdre régulier de septième espèce à faces convexes

L'icosaèdre régulier de septième espèce est obtenu en partant de l'icosaèdre régulier convexe; chaque face est un triangle équilatéral obtenu en considérant chacune des vingt faces de l'icosaèdre convexe, et en joignant les troisièmes sommets des triangles équilatéraux constituant les trois faces adjacentes. (La face de l'icosaèdre convexe et la face correspondante de l'icosaèdre étoilé sont dans des plans parallèles).

Par exemple à la face {1, 3, 2} de l'icosaèdre convexe correspond la face {4, 6, 7} de l'icosaèdre étoilé.

On peut construire facilement la liste des faces de ce polyèdre à la main, puis le dessiner; nous allons cependant écrire un programme qui produit cette liste.

Pour construire la face {4, 6, 7}, on commence par selectionner les faces adjacentes (elles ont 2 sommets communs avec la face {1,3,2}). Remarquons que le polygone étant un triangle, l'ordre des sommets n'a pas d'importance, ce qui simplifie le programme par rapport au précédent :

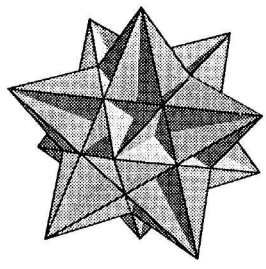
```
In[] := | Select[faces, Length[Intersection[#, {1, 3, 2}]] == 2&]
Out[] = | {{1, 4, 3}, {1, 2, 6}, {2, 3, 7}}
```

On isole le troisième sommet :

```
In[] := | Complement[#, {1, 3, 2}]& /@ %
Out[] = | {{4}, {6}, {7}}

In[] := | Flatten[%]
Out[] = | {4, 6, 7}
```

et notre fonction :

```
In[] := | opface[lf_, f_] :=
        |  Flatten[Map[
        |   Complement[#, f]&,
        |    Select[lf,
        |     Length[Intersection[#, f]] == 2&
        |     ] ] ]

In[] := | faces2 = opface[faces, #]& /@ faces
Out[] = | {{4, 6, 7}, {2, 5, 8}, {3, 6, 9}, {4, 2, 10}, {3, 5, 11},
        |  {1, 8, 11}, {1, 7, 9}, {1, 8, 10}, {1, 9, 11}, {1, 10, 7},
        |  {2, 4, 12}, {3, 5, 12}, {4, 6, 12}, {5, 2, 12}, {3, 6, 12},
        |  {3, 9, 11}, {4, 7, 10}, {5, 8, 11}, {6, 9, 7}, {2, 8, 10}}

In[] := | ico2 = Show[Graphics3D[Polygon[sommets[[#]]]& /@ faces2]];
```



-Figure 12-

3.3.2 - Les polyèdres d'espèce supérieure à 1 à faces croisées

Ces polyèdres s'obtiennent en joignant les sommets d'un polyèdre régulier convexe de façon à former des faces qui soient des polygones réguliers croisés (il s'agira donc toujours de pentagone régulier croisé). Il y a deux façons d'obtenir ainsi un nouveau polyèdre régulier :

- En partant de l'icosaèdre régulier convexe, on obtient le dodécaèdre régulier de troisième espèce à faces étoilées.

- En partant du dodécaèdre régulier convexe, on obtient le dodécaèdre régulier de septième espèce à faces étoilées.

Les mêmes polyèdres peuvent s'obtenir en partant d'un polyèdre régulier convexe (le "noyau"), et en remplacant chaque face par une pyramide basée sur cette face, de sommet l'image du centre de gravité de la face par l'homothétie centrée au centre du polyèdre, et de rapport un nombre k convenablement choisi.

- Pour le dodécaèdre régulier de troisième espèce à faces étoilées (dit de Kepler), le noyau sera un dodécaèdre régulier convexe et le rapport de stellation : $k = \sqrt{5}$.

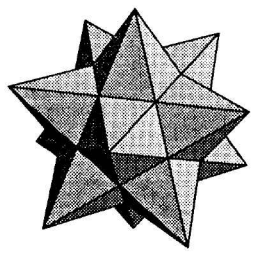
- Pour le dodécaèdre régulier de septième espèce à faces étoilées, le noyau sera un icosaèdre régulier convexe et le rapport de stellation : $k = 3$.

En utilisant la fonction `etoiler` construite dans l'exercice 4, on peut facilement construire ces polyèdres.

Nous partons des dodécaèdre et icosaèdre convexes (construits ici à l'aide du fichier Graphics`Polyhedra`) :

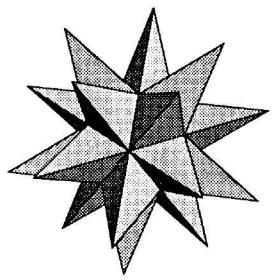
```
In[] := | dode = Polyhedron[Dodecahedron];

In[] := | Show[dode /. Polygon[f_] :> etoiler[f, √5], Boxed -> False];
```

-Figure 13-



```
In[] := | ico = Polyhedron[Icosahedron];

In[] := | Show[ico /. Polygon[f_] :> etoiler[f, 3], Boxed -> False];
```

-Figure 14-



Remarque : Il n'est pas possible ici d'utiliser la méthode consistant à joindre les sommets du polyèdre pour le tracer, comme en 3.3.1; en effet les polygones croisés dans l'espace ne sont pas rendus comme dans le plan (mais comme des polygones convexes que l'on aurait "plié").

Le lecteur pourra par exemple évaluer les cellules ci-dessous qui tentent de construire le dodécaèdre de Kepler :

```
In[] := | faces1x = #[[{1, 3, 5, 2, 4}]]& /@ faces1
```

(Sortie non montrée)

```
In[] := | Show[Graphics3D[Polygon[sommets[[#]]]& /@faces1x]];
```

(Figure non montrée)

```
In[] := | Show[Graphics3D[Polygon[sommets[[#]]]& /@ {faces1x[[6]]}]];
```

(Figure non montrée)

Correction des exercices

Exercice 1 :

La primitive Cuboid ne convient pas car ses faces restent parallèles aux plans de coordonnées; il faut donc construire les cubes entièrement.

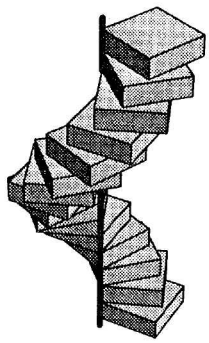
Un cube unité dont un sommet est à la hauteur h sur l'axe des z, une arête confondue avec l'axe des z, une autre horizontale formant l'angle θ avec l'axe des abscisses :

```
In[] := | cube[θ_, h_] := Module[{u = {0, 0, 1},
            v = {Cos[θ], Sin[θ], 0}, w = {-Sin[θ], Cos[θ], 0}, l, l1},
          l = {h u, h u + v, h u + v + w, h u + w};
          l1 = {(h + 1) u, (h + 1) u + v, (h + 1) u + v + w, (h + 1) u + w};
          Join[
           Polygon /@ Transpose[{l, RotateLeft[l], RotateLeft[l1], l1}],
           Polygon /@ {l, l1}] ]
```

```
In[] := | colimacon[n_] :=
          Graphics3D[Join[Table[cube[2 k π / n, k], {k, 0, n - 1}],
            {Thickness[1 / 50], Line[{{0, 0, -1 / 2}, {0, 0, n + 1 / 2}}]}]]
```

```
In[] := | Show[colimacon[16], Boxed -> False, BoxRatios -> {1, 1, 2}];
```

-Figure 15-



Exercice 2 :

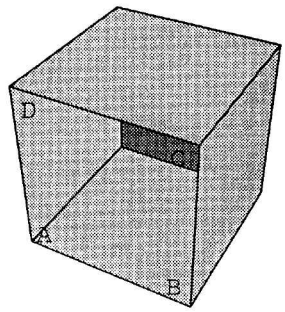
L'objet cube est un cube unité avec la face d'équation $y = 0$ supprimée pour voir l'intérieur; les lettres colorées A, B, C, D sont aux quatre coins de cette face manquante.

```
In[] := cube := With[{10 = {{0, 0, 0}, {1, 0, 0}, {1, 0, 1}, {0, 0, 1}},
          11 = {{0, 1, 0}, {1, 1, 0}, {1, 1, 1}, {0, 1, 1}}},
         Graphics3D[{
        Polygon /@ Transpose[
             {10, RotateLeft[10], RotateLeft[11], 11}], Polygon @ 11,
           PointSize[1 / 50], RGBColor[1, 0, 0], Text["A", {0.1, 0, 0.1}],
          RGBColor[0, 1, 0], Text["B", {0.9, 0, 0.1}],
          RGBColor[0, 0, 1], Text["C", {0.9, 0, 0.9}],
          RGBColor[1, 1, 0], Text["D", {0.1, 0, 0.9}]
        }, Boxed -> False]]
```

La valeur par défaut de `ViewCenter` est `Automatic` : le plan de projection est calculé pour passer par le centre de la boite, de façon que l'image soit la plus grande possible. Cette valeur est le plus souvent {0.5, 0.5, 0.5}.

```
In[] := Show[cube];
```

-Figure 16-




```
In[] := FullOptions[%, ViewCenter]
Out[] = {0.5, 0.5, 0.5}
```

Plus le point de vue est éloigné de l'objet à représenter (c'est à dire de la boite), plus la projection se rapproche d'une projection oblique.

Ci-dessous, on projète frontalement, avec le point de vue devant l'objet et le plan de projection derrière.

```
In[] := Show[cube, ViewPoint -> {0, -10, 0},
         ViewCenter -> {1 / 2, 10, 1 / 2}];
```

(Figure non montrée)

et avec le point de vue entre l'objet et le plan de projection : comme prévisible, l'image est inversée dans ce cas.

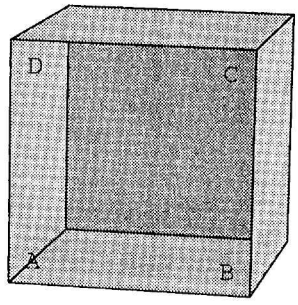
```
In[] := Show[cube, ViewPoint -> {0, -10, 0},
         ViewCenter -> {1 / 2, -30, 1 / 2}];
```

(Figure non montrée)

Avec un point de vue éloigné et incliné à 45 degrés, on approche de la projection dite "perspective cavalière".

```
In[] := | Show[cube, ViewPoint -> {2, -10, 2}];
```

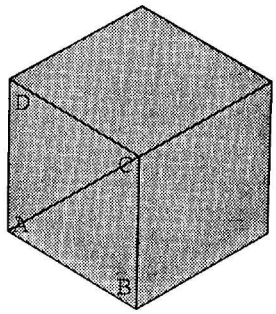
-Figure 17-



Une projection isométrique :

```
In[] := | Show[cube, ViewPoint -> {30, -30, 30}];
```

-Figure 18-



Exercice 3 :

Les sources ci-dessous sont réparties au dessus de l'écran, face à la figure.

Rappelons que les graphiques définis dans l'option `Prolog` sont relatifs à un système de coordonnées écran (comprises entre 0 et 1).

```
In[] := | ampoules =
          Table[Circle[{(3 + 2 Cos[k])/6, (3 + 2 Sin[k])/6}, 0.05], {k, 0, 9 π/5, π/5}];

In[] := | source[k_] := {Hue[k/(2 π)], Disk[{(3 + 2 Cos[k])/6, (3 + 2 Sin[k])/6}, 0.05]}

In[] := | lumiere[k_] := {{Cos[k], Sin[k], 1}, Hue[k/(2 π)]}

In[] := | << Graphics`Shapes`

In[] := | Do[
          Show[Graphics3D[Prepend[Sphere[], EdgeForm[]], Boxed -> False],
          LightSources -> {lumiere[k]}, Prolog -> {source[k], ampoules},
           PlotRange -> {{-2, 2}, {-2, 2}, {-2, 2}}, AspectRatio -> 1],
          {k, 0, 9 π/5, π/5}];
```

(Figures non montrées)

Exercice 4 :

Si $f = \{s_1, s_2, ..., s_p\}$ est une liste de points (sommets d'une face du polygone), le sommet de la pyramide ayant cette face pour base sera obtenue par :

```
In[] := | s = k (Plus @@ f) / Length[f]
```

k étant le rapport de stellation.

Pour construire la liste de triangles Polygon[$\{s, s_k, s_{k+1}\}$] à partir de la face f, on peut procéder ainsi :

```
In[] := | f = {s1, s2, s3, s4, s5};

In[] := | Append[%, First[%]]
Out[] = | {s1, s2, s3, s4, s5, s1}

In[] := | Partition[%, 2, 1]
Out[] = | {{s1, s2}, {s2, s3}, {s3, s4}, {s4, s5}, {s5, s1}}

In[] := | Prepend[#, s]& /@ %
Out[] = | {{s, s1, s2}, {s, s2, s3}, {s, s3, s4}, {s, s4, s5}, {s, s5, s1}}

In[] := | Polygon /@ %
Out[] = | {Polygon[{s, s1, s2}], Polygon[{s, s2, s3}], Polygon[{s, s3, s4}],
          Polygon[{s, s4, s5}], Polygon[{s, s5, s1}]}
```

et le programme :

```
In[] := | etoiler[f_, k_] := Map[Polygon,
          Map[Prepend[#, k (Plus @@ f) / Length[f]]&,
           Partition[
            Append[f, First[f]],
           2, 1]
           ]
          ]

In[] := | Show[ico /. Polygon[f_] :> etoiler[f, 1/Sqrt[2]]];
```

(Figure non montrée)

Chapitre 5

Transformations géométriques

Si nous voulons utiliser des transformations géométriques avec *Mathematica*, nous aurons deux sortes de programmes à envisager :

– Des programmes qui définissent des transformations géométriques ponctuelles; ces transformations seront définies en mode exact et serviront aussi bien à exécuter du calcul géométrique formel, qu'à réaliser des graphiques à partir de calculs approchés. Si nous rassemblons ces programmes dans un fichier de commandes, ce dernier sera à placer dans un répertoire de géométrie.

– Des programmes qui construisent des images d'objets graphiques par des transformations géométriques; ils feront appel naturellement aux fonctions précédentes. Si nous rassemblons ces programmes dans un fichier de commandes, ce dernier sera à placer dans un répertoire de graphiques

Avec la version 3 de *Mathematica*, on peut ainsi prévoir dans le sous-répertoire `Applications` du répertoire `AddOns` des sous-répertoires `Geometrie, Graphiques` (entre autres).

1 - Constitution d'une bibliothèque de transformations géométriques

1.1 - Le fichier « Geometry`Rotations` »

Le fichier standard `Geometry`Rotations`` produit des outils permettant de définir des rotations dans le plan et dans l'espace.

```
In[] := << Geometry`Rotations`
```

La fonction `RotationMatrix2D` donne la matrice de changement de repère qui permet d'obtenir les nouvelles coordonnées en fonction des anciennes; il faudra donc changer le sens de l'angle pour obtenir des matrices de rotations.

```
In[] := RotationMatrix2D[θ] // MatrixForm
Out[] = ( Cos[θ]   Sin[θ] )
        ( -Sin[θ]  Cos[θ] )
```

La fonction `Rotate2D[v, -θ]` [resp. `Rotate2D[m, -θ, c]`] donne l'image d'un vecteur `v` [resp. d'un point `m`] par une rotation d'angle θ [resp. d'angle θ et de centre `c`].

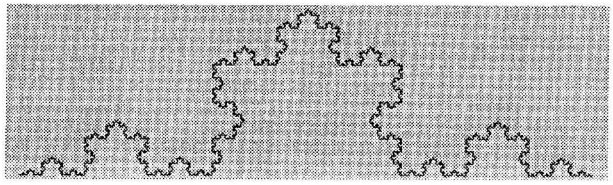
Le programme ci-dessous construit l'ensemble triadique de Cantor :

```
In[] := cantorstep[{a_, b_}] := With[{a1 = a + (b - a)/3, b1 = a + 2 (b - a)/3},
          {a, a1, Rotate2D[a1, π/3, b1], b1, b}]
```

```
In[] := cantor[s : {_, _}, 0] := s;
        cantor[s : {_, _}, n_Integer?(# > 0 &)] :=
        Flatten[cantorstep /@ Partition[cantor[s, n - 1], 2, 1], 1]
```

```
In[] := Show[Graphics[
          Line @ cantor[N[{{0, 0}, {1, 0}}], 5], AspectRatio -> Automatic],
        Background -> RGBColor[0, 1, 1]];
```

-Figure 1-



Le calcul de la la longueur de cette ligne est classique; on peut aussi le programmer :

```
In[] := distance[{u_, v_}] := √((u - v) . (u - v))

In[] := longueur[1 : {{_, _}..}] := Plus @@ distance /@ Partition[1, 2, 1]

In[] := longueur[cantor[{{0, 0}, {1, 0}}, 5]]

Out[] = 1024/243
```

La fonction `RotateMatrix3D` retourne la matrice d'une rotation caractérisée par ses 3 angles d'Euler; si (O,**i**,**j**,**k**) est un repère et (O,**i'**,**j'**,**k'**) son image par la rotation, les angles d'Euler de cette rotation sont :

– ψ (precession) angle (**i**,**u**) autour de (O,**k**), avec (O,**u**) intersection des plans (O,**i**,**j**) et (O,**i'**,**j'**)

– θ (nutation) angle (**k**,**k'**) autour de (O,**u**) ((O,**u**) dite ligne des noeuds)

– φ (rotation propre) angle (u,**i'**) autour de (O,**k'**)

Comme pour les rotations planes, c'est la matrice inverse (ou transposée) qui est retournée :

```
In[] := (m = RotationMatrix3D[-ψ, -θ, -φ]) // MatrixForm

Cos[φ] Cos[ψ] - Cos[θ] Sin[φ] Sin[ψ]    -Cos[θ] Cos[ψ] Sin[φ] - Cos[φ] Sin[ψ]    Sin[θ] Sin[φ]
Cos[ψ] Sin[φ] + Cos[θ] Cos[φ] Sin[ψ]    Cos[θ] Cos[φ] Cos[ψ] - Sin[φ] Sin[ψ]    -Cos[φ] Sin[θ]
          Sin[θ] Sin[ψ]                          Cos[ψ] Sin[θ]                       Cos[θ]
```

On obtient facilement les rotations partielles :

```
In[] := RotationMatrix3D[0, 0, -φ] . RotationMatrix3D[0, -θ, 0] .
         RotationMatrix3D[-ψ, 0, 0] === m

Out[] = True
```

La fonction `Rotate3D[v,-ψ,-θ,-φ]` [resp. `Rotate3D[m,-ψ,-θ,-φ,c]`] donne l'image d'un vecteur v [resp. d'un point m] par une rotation d'angles d'Euler ψ, θ, φ [resp. d'angles d'Euler ψ, θ, φ et dont l'axe passe par c].

1.2 - Construction d'un fichier «Geometrie`Transformations`»

Nous nous proposons de construire un ensemble de fonctions représentant les transformations géométriques les plus courantes; il s'agira de transformations ponctuelles.

Nous adopterons pour ces fonctions la syntaxe suivante : `transfo` étant un symbole désignant une famille de transformations, `transfo[p1, p2, ...]` désignera la transformation de paramètres `p1, p2, ...` et

`transfo[p1, p2, ...] [pt]` l'image du point `pt` par cette transformation. Le nom du symbole est terminé par un nombre de dimensions (2 ou 3) lorsque la transformation n'est pas définie en dimension quelconque.

Le lecteur pourra enrichir le fichier d'autres transformations s'il le désire.

1.2.1 - Construction du fichier

Nous rappelons qu'avec la version 3, il suffit d'enregistrer dans le répertoire `Geometrie` un fichier `Transformations.nb` et de marquer les cellules contenant les programmes "cellules d'initialisation"; le fichier `Transformations.m` se créera et se modifiera automatiquement.

Cahier Transformations.nb

◆ **Contexte public :**

```
In[] := BeginPackage["Geometrie`Transformations`", "Geometry`Rotations`"]
```

◆ **Messages d'usage :**

```
In[] := Translation::usage = "Translation[vec][m] fait correspondre au
        point m = {x1,x2,...,xn} le point vec + m."

In[] := Homothetie::usage = "Homothetie[c,k][m] fait correspondre au
        point m = {x1,x2,...,xn} son image dans l'homothetie de centre
        c, de rapport k."

In[] := Projection::usage = "Projection[e][m] calcule la projection
        orthogonale du point m sur l'espace engendré par les points
        de la liste e.\n Projection[e,dir][m] calcule la projection
        de m sur e, suivant la direction de l'espace engendré par les
        points de la liste dir."

In[] := Symetrie::usage = "Symetrie[e][m] calcule l'image du point m
        par la symetrie orthogonale par rapport à l'espace engendre
        par les points de la liste e.\n Symetrie[e,dir][m] calcule le
        symétrique de m par rapport à e, suivant la direction de
        l'espace engendré par les points de la liste dir."

In[] := Affinite::usage = "Affinite[e,k][m] calcule l'image du point m
        par l'affinité orthogonale par rapport à l'espace engendre par
        les points de la liste e, de rapport k.\n Affinite[e,dir,k][m]
        calcule l'image de m par l'affinité par rapport à e, suivant
        la direction de l'espace engendré par les points de la liste
        dir et de rapport k."

In[] := Reflexion::usage = "Reflexion[p,vn][m] fait correspondre au point
        m = {x1,x2,...,xn} son image dans la reflexion (symetrie
        orthogonale hyperplane) par rapport à l'hyperplan passant
        par p et de vecteur normal vn."

In[] := Rotation2::usage = "Rotation2[c,theta][{x,y}] donne l'image
        du point {x,y} par la rotation plane de centre c."

In[] := Rotation3::usage = "Rotation3[c,theta,direction][{x,y,z}]
        donne l'image du point {x,y,z} par la rotation d'angle theta, d'
           axe la droite passant par c, dirigée par direction.\n
        Rotation3[point,psi,theta,phi][{x,y,z}] donne l'image du point
        {x,y,z} par la rotation de l'espace definie par les angles
        d'Euler psi,theta,phi et dont c est un point invariant."
```

```
In[] := Vissage3::usage =
         "Vissage3[d,theta,transl][m] donne l'image du point
        m = {x1,x2,x3} par le vissage d'angle theta, d'axe la droite
           passant par d, dirigee par transl et de vecteur transl."
```

```
In[] := AffineT::usage = "AffineT[mat,c][m] donne l'image du point m
        par la transformation afine de matrice mat, dont c est un
        point invariant.\n
        AffineT[mat] correspond au cas où c est l'origine du repère."

        ComplexeT2::usage = "ComplexeT2[fonc][m] donne le
           point m' d'affixe fonc[z] où z est l'affixe de m.\n
        ComplexeT2[expr,z][m]donne le même point, la fonction
           complexe étant définie par une expression de z."

        Inversion::usage = "Inversion[c,p][m] donne l'inverse du
           point m dans l'inversion de pôle c et de puissance p."

        AnalytiqueT::usage =
         "AnalytiqueT[{f1,f2,...,fn}][m] donne l'image du point m par
           la transformation définie par les fonctions f1,f2,...,fn.\n
          AnalytiqueT[{e1,e2,...,en},{x1,x2,...,xp}][m] donne
           l'image du point m par la transformation définie
           analytiquement par les expressions ei des variables xj."
```

◆ **Contexte privé :**

```
In[] := Begin["`Private`"]
```

◆ **Messages d'erreurs :**

```
In[] := Projection::dir = "La base et la direction de la projection ne
        définissent pas des sous-espaces supplémentaires."
        RotationMatrice3::dir =
         "Le vecteur `` ne définit pas une direction
        de droite dans l'espace."
        InverseFunction::existe =
         "La transformation `` n'admet pas de réciproque."
```

◆ **Produit vectoriel :**

A définir uniquement avec des versions antérieurs à la version 3 :

```
In[] := Cross[v1_, v2_] := Module[{m = Minors[{v1, v2}, 2][[1]]},
          {m[[3]], -m[[2]], m[[1]]} ]
```

◆ **Transformations générales :**

```
In[] := Translation[vec_][x_] := vec + x

        Homothetie[c_, k_][x_] := c + k (x - c)

        Reflexion[p_, vn_][x_] := x - 2 (x - p) . vn / (vn . vn) vn
```

La projection sur le sous-espace affine e engendré par les points {a,b,c,d} s'obtient ainsi : ve est la direction de ce sous-espace, engendré par {b-a,c-a,d-a}; le sous-espace orthogonal oe est le noyau de la matrice ve.

La projection y de x est à l'intersection des sous-espaces de directions oe et ve, passant respectivement par First[e] et x; leurs équations sont : oe.y = oe.First[e] et ve.y = ve.x.

Même principe pour la projection oblique : la projection y de x est à l'intersection de deux sous-espaces d'équations `m1.y=m1.First[e]` et `m2.y=m2.x`.

```
In[] :=
Projection[e_?MatrixQ][x_] := Module[
 {oe, ve = Map[# - First[e]&, Rest[e]]},
 oe = NullSpace[ve];
 LinearSolve[Join[oe, ve], Join[oe.First[e], ve.x]] ]

Projection[e_?MatrixQ, dir_?MatrixQ][x_] := Module[
 {m1 = NullSpace[Map[# - First[e]&, Rest[e]]],
 m2 = NullSpace[Map[# - First[dir]&, Rest[dir]]], m},
 m = Join[m1, m2];
 If[ NullSpace[m] =!= {},
  Message[Projection::dir]; Return[Null] ];
 LinearSolve[m, Join[m1.First[e], m2.x]] ]

Projection[e__ /; MatrixQ[{e}]][x_] := Projection[{e}][x]
```

Les symétries et affinités se définissent facilement à partir des projections.

```
In[] :=
Symetrie[e__][x_] := With[
 {p = Projection[e][x]},
 If[ p === Null,
  Message[Projection::dir]; Return[Null],
  2 p - x
] ]
```

```
In[] :=
Affinite[e_?MatrixQ, k_][x_] :=
 k x + (1 - k) Projection[e][x]

Affinite[e_?MatrixQ, dir_?MatrixQ, k_][x_] := With[
 {p = Projection[e, dir][x]},
 If[ p === Null,
  Message[Projection::dir]; Return[Null],
  k x + (1 - k) p
] ]
```

L'inversion de pôle c et de puissance p se définit aussi facilement :

```
In[] := Inversion[c_, p_][x_] := With[{y = x - c}, p y / (y.y) + c]
```

Enfin nous définissons des transformations générales : linéaires à partir d'une matrice, puis analytiques quelconques à partir de leurs formules; plus loin, nous définissons aussi des transformations planes à partir d'une fonction complexe.

```
In[] :=
AffineT[mat_][x_] := mat.x
AffineT[mat_, c_][x_] := mat.(x - c) + c
```

```
In[] :=
AnalytiqueT[foncts_List][m_List] :=
 Through[foncts[Sequence @@ m]]
AnalytiqueT[expr_List, x:{_Symbol..}][m_List] /;
  Length[x] == Length[m] :=
 AnalytiqueT[ Function[x, #]& /@ expr][m]
```

◆ **Transformations planes :**

Nous renommons les matrices de rotations afin de changer leurs angles :

```
In[] := RotationMatrice2[theta_] := RotationMatrix2D[-theta]
```

```
In[] := Rotation2[c_, rotVec__][x_] :=
         c + RotationMatrice2[rotVec] . (x - c)
```

```
In[] := ComplexeT2[fonc_][x_] := {Re[#], Im[#]}& @ fonc[x . {1, I}]
        ComplexeT2[expr_, z_Symbol][x_] :=
         ComplexeT2[Function[z, expr]][x]
```

◆ **Transformations dans l'espace :**

Nous renommons également les matrices de rotations; deux syntaxes permettent de définir une rotation par son axe et son angle, ou par ses angles d'Euler :

```
In[] := RotationMatrice3[theta_, dir_List] :=
         Module[{u = {- dir[[2]], dir[[1]], 0}, v, w, p},
          If[u == {0, 0, 0}, u = {1, 0, 0}];
          v = Cross[dir, u];
          u = u / Sqrt[u . u];
          v = v / Sqrt[v . v];
          w = dir / Sqrt[dir . dir];
          p = {u, v, w};
          Inverse[p] . {{Cos[theta], - Sin[theta], 0},
           {Sin[theta], Cos[theta], 0}, {0, 0, 1}} . p
         ] /; Length[dir] == 3 && dir =!= {0, 0, 0}
        RotationMatrice3[psi_, theta_, phi_] :=
         RotationMatrix3D[-psi, -theta, -phi]
        RotationMatrice3[theta_, dir_List] :=
         (Message[RotationMatrice3::dir, dir]; Null)
```

```
In[] := Rotation3[c_, rotVec__][x_] :=
         c + RotationMatrice3[rotVec] . (x - c)

        Vissage3[d_, theta_, transl_][x_] :=
         d + transl + RotationMatrice3[theta, transl] . (x - d)
```

◆ **Fonctions générales :**

Il est aussi possible d'étendre ce fichier en ajoutant des définitions aux fonctions `InverseFunction` et `Composition` pour calculer les réciproques et les composées des transformations précédentes.

```
In[] := InverseFunction[Translation[vec_]] ^:= Translation[- vec]
        InverseFunction[Homothetie[c_, k_]] ^:=
         If[ TrueQ[k != 0],
          Homothetie[c, 1 / k],
          Message[InverseFunction::existe, Homothetie[c, k]]; $Failed ]
        InverseFunction[Reflexion[p_, vn_]] ^:= Reflexion[p, vn]
        InverseFunction[Projection[e__]] ^:=
         Message[InverseFunction::existe, Projection[e]]; $Failed
        InverseFunction[Symetrie[e__]] ^:= Symetrie[e]
        InverseFunction[Affinite[e_, k_]] ^:=
         If[ TrueQ[k != 0], Affinite[e, 1 / k],
          Message[InverseFunction::existe, Affinite[e, k]]; $Failed ]
```

Suite de la cellule d'entrée de la page précédente :

```
InverseFunction[Affinite[e_, dir_, k_]] ^:=
 If[ TrueQ[k != 0], Affinite[e, dir, 1/k],
  Message[InverseFunction::existe, Affinite[e, dir, k]]; $Failed ]
InverseFunction[Rotation2[c_, theta_]] ^:= Rotation[c, - theta]
InverseFunction[Rotation3[c_, theta_, dir_]] ^:=
 Rotation[c, - theta, dir]
InverseFunction[Rotation3[c_, psi_, theta_, phi_]] ^:=
 Rotation[c, - phi, - theta, - psi]
InverseFunction[Vissage3[d_, theta_, transl_]] ^:=
 Vissage[d, - theta, - transl]
InverseFunction[Inversion[c_, p_]] ^:= Inversion[c, p]
InverseFunction[AffineT[mat_, c___]] ^:=
 Check[AffineT[Inverse[mat], c],
  Message[InverseFunction::existe, AffineT[mat, c]],
  Inverse::sing]
```

◆ **Fin du fichier :**

```
In[] := Protect[Translation, Homothetie, Reflexion, Projection,
         Affinite, Symetrie, Inversion, Rotation2, Rotation3,
         Vissage3, ComplexeT2, AffineT, AnalytiqueT]

In[] := End[]

In[] := EndPackage[]
```

Fin du cahier Transformations.nb

1.2.2 - Exemples

```
In[] := << Geometrie`Transformations`

In[] := ?Geometrie`Transformations`*
```

AffineT	Inversion	Symetrie
Affinite	Projection	Transformations
AnalytiqueT	Reflexion	Translation
ComplexeT2	Rotation2	Vissage3
Homothetie	Rotation3	

◆ **Premier exemple :**

Soit à calculer la matrice de la projection de $\mathbb{R}^4$ sur le plan d'équations $\{x - 2y + 3z + t = 0, x + y - z + 2t = 0\}$ suivant la direction du plan engendré par les vecteurs $\{\{1, 2, 0, 3\}, \{-1, 0, 1, 2\}\}$.

```
In[] := Solve[{x - 2 y + 3 z + t == 0, x + y - z + 2 t == 0}]

        Solve::svars :
         Equations may not give solutions for all "solve" variables.

Out[] = {{t -> -3 y + 4 z, x -> 5 y - 7 z}}

In[] := {x, y, z, t} /. %[[1]] /. {{y -> 1, z -> 0}, {y -> 0, z -> 1}}
Out[] = {{5, 1, 0, -3}, {-7, 0, 1, 4}}

In[] := p1 = Prepend[%, {0, 0, 0, 0}]
Out[] = {{0, 0, 0, 0}, {5, 1, 0, -3}, {-7, 0, 1, 4}}
```

```
In[] := p2 = {{0, 0, 0, 0}, {1, 2, 0, 3}, {-1, 0, 1, 2}};

In[] := Transpose[Projection[p1, p2] /@ {{1, 0, 0, 0}, {0, 1, 0, 0},
          {0, 0, 1, 0}, {0, 0, 0, 1}}] // MatrixForm
```

$$\text{Out[]} = \begin{pmatrix} \frac{43}{36} & -\frac{13}{18} & \frac{37}{36} & \frac{1}{12} \\ -\frac{1}{9} & \frac{5}{9} & \frac{5}{9} & -\frac{1}{3} \\ -\frac{1}{4} & \frac{1}{2} & \frac{1}{4} & -\frac{1}{4} \\ -\frac{2}{3} & \frac{1}{3} & -\frac{2}{3} & 0 \end{pmatrix}$$

◆ **Second exemple :**

La fonction standard `Composition` permet de composer des applications.

```
In[] := f = Composition[Homothetie[{1, 1}, 2], Reflexion[{1, 1}, {3, 2}]]
Out[] = Composition[Homothetie[{1, 1}, 2], Reflexion[{1, 1}, {3, 2}]]

In[] := g = Composition[Reflexion[{1, 1}, {3, 2}],
          Homothetie[{1, 1}, 2]];

In[] := Expand[f[{a, b}]] === Expand[g[{a, b}]]
Out[] = True

In[] := Composition[Inversion[{0, 0}, p],
         Inversion[{0, 0}, q]][{a, b}] // Simplify
```

$$\text{Out[]} = \left\{\frac{a\,p}{q}, \frac{b\,p}{q}\right\}$$

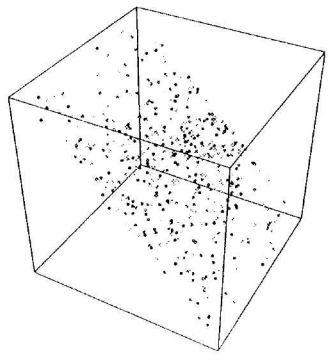
◆ **Troisième exemple :**

On remplit le cube unité de points aléatoires, puis on les projète sur un plan.

```
In[] := Show[points = Graphics3D[Table[{Hue[Random[]], Point[{Random[],
          Random[], Random[]}]}, {500}], Boxed -> True]];
```

(figure non montrée)

```
In[] := a = {1, 0, 0}; b = {1, 1, 0}; c = {0, 0, 1};

In[] := Show[points /. Point[m_] :> Point[Projection[{a, b, c}][m]]];
```



-Figure 2-

La règle différée est nécessaire, pour évaluer numériquement `Projection[{a,b,c}][m]`.

2 - Transformation de graphiques

2.1 - Le fichier Graphics`Shapes`

```
In[] := << Graphics`Shapes`
```

Le fichier `Graphics`Shapes`` est une bibliothèque de surfaces usuelles : cylindres, cônes, sphères, tores, hélices, bandes de Mobius.

Les fonctions représentant ces surfaces : `Cylinder`, `Cone`, `Helix`, `MoebiusStrip`, `Sphere`, `Torus` sont des listes de polygones; on les enveloppera par la fonction `Graphics3D` pour en faire des objets graphiques.

Les paramètres de ces fonctions sont de deux sortes :

– les premiers caractérisent géométriquement la surface (rayon de la sphère, rayon et demi-hauteur du cône ou du cylindre, rayons du tore etc.)

– les derniers précisent le nombre de polygones à générer (résolution de la surface)

Les surfaces sont centrées par rapport au repère; pour les représenter en position quelconque, on utilise les fonctions suivantes que l'on peut combiner entre elles :

– `RotateShape` pour effectuer une rotation centrée au centre du repère; on précise les angles d'Euler.

– `TranslateShape` pour effectuer une translation; on précise les coordonnées du vecteur.

– `AffineShape` pour effectuer un changement d'échelle sur chacun des axes; on précise les 3 facteurs correspondants.

Le lecteur utilisera l'aide en ligne ou l'ouvrage de référence (Standard Add-on Packages) pour plus de renseignements.

A titre d'exemple, voici quelques commandes à évaluer :

```
In[] := Show[Graphics3D[
          AffineShape[TranslateShape[Cone[], {0, 0, -1}], {-1, -1, -1}]],
         Axes -> True];
```

(Figure non montrée)

```
In[] := Show[Graphics3D[AffineShape[Sphere[], {3, 1, 1/2}]],
         Boxed -> False];
```

(Figure non montrée)

Nous proposons de colorier en rouge et bleu les faces opposées d'une bande de Mobius puis d'effectuer une animation pour "tourner autour" de la surface :

```
In[] := Show[Graphics3D[
          Join[{EdgeForm[], FaceForm[SurfaceColor[RGBColor[1, 0, 0]],
             SurfaceColor[RGBColor[0, 0, 1]]]},
           MoebiusStrip[3, 2, 50]]],
         Boxed -> False, ViewPoint -> {0.5, -1., 2.},
        LightSources -> {{{2, 1, 0.7}, GrayLevel[1]}}];
```

(Figure non montrée)

```
In[] := SpinShow[%]
```

(Figures non montrées)

2.2 - Utilisation de notre fichier Transformations`

```
In[] := << Geometrie`Transformations`
```

On construit une cardioïde, puis son image dans une inversion de pôle O.

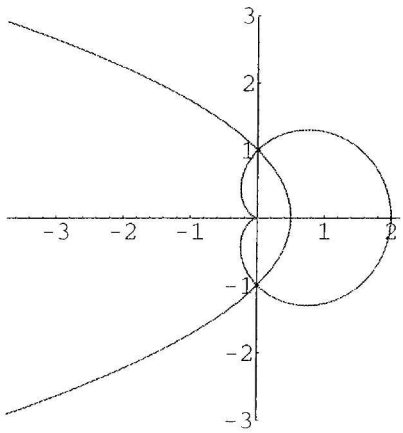
Il est prudent de calculer assez de points dans le graphique de départ (option `PlotPoints`).

```
In[] := gr = ParametricPlot[{(1 + Cos[t]) Cos[t], (1 + Cos[t]) Sin[t]},
          {t, 0, 2 π}, AspectRatio -> Automatic, PlotPoints -> 100,
         PlotStyle -> RGBColor[1, 0, 0]];
```

(Figure non montrée)

```
In[] := Show[gr, gr /. Line[x_] :> Line[Inversion[{0, 0}, 1] /@ x],
        PlotRange -> {-3, 3}];
```

-Figure 3-



On construit maintenant un tore (fichier `Graphics`Shapes``) puis son image par une inversion.

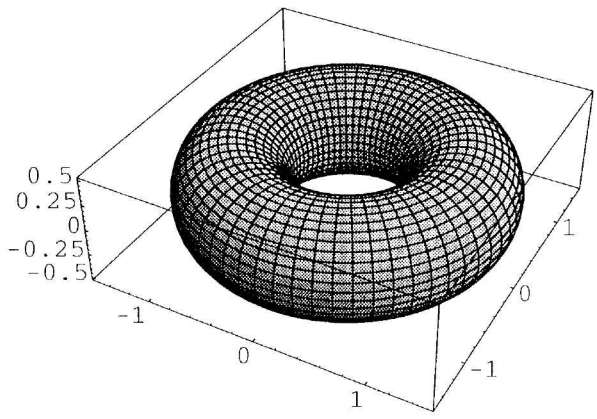
Le lecteur peut varier la position du centre d'inversion; on obtient en général une cyclide de Dupin.

```
In[] := tore = Show[Graphics3D[Torus[1, .5, 60, 40]], Boxed -> False];
```

(Figure non montrée)

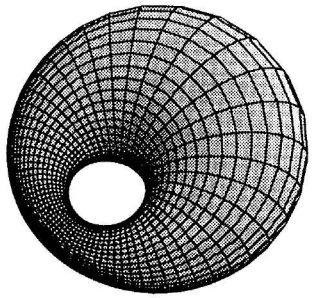
```
In[] := Show[tore, Boxed -> True, Axes -> True];
```

-Figure 4-



```
In[] := Show[tore /. Polygon[x_] :>
         Polygon[Inversion[{0, -3, 0}, 1]/@ x], PlotRange -> All,
         ViewPoint -> {1, -1, 3}];
```

-Figure 5-



2.3 - Construction d'un fichier de transformation de graphiques 2D

Mathematica ne possède pas en standard de fonction permettant de transformer un objet graphique par une transformation géométrique.

Mais des fichiers ont déjà été réalisés par de nombreux auteurs; citons en particulier :

– Des fonctions programmées par Th. W. Gray et J. Glynn dans leur ouvrage Exploring Mathematics with *Mathematica* (Addison-Wesley).

– De nombreux fichiers graphiques réalisés par Xah Lee et disponibles dans le répertoire *MathSource* sur le site internet de Wolfram ou sur CDRom; plus particulièrement pour ce qui nous concerne le fichier `Transform2DPlot.m`.

Nous allons construire quelques programmes simplifiés qui font ce travail pour les graphiques 2D puis pour les graphiques 3D.

Il peut être intéressant de construire l'image d'un graphique non seulement par une transformation géométrique, mais aussi par un ensemble de transformations géométriques : si $F = \{f_1, f_2, \ldots, f_n\}$ est un ensemble de transfomations et G un ensemble de points, l'image de G par F sera l'ensemble : $f_1(G) \cup f_2(G) \cup \ldots \cup f_n(G)$.

2.3.1 - Programmation des fonctions

Un graphique 2D sera principalement à base de primitives `Point` et `Line` et parfois `Cercle`, `Disk`, `Rectangle`, `Polygon`; il n'y aura pas de difficulté pour transformer un point; par contre la primitive `Line` représente un segment de droite ou une ligne brisée et si la transformation qui opère n'est pas affine, un segment de droite ne sera pas transformé en un segment de droite. Le même problème se pose pour les côtés des rectangles et polygones.

Si la ligne est formée de segments de longueurs suffisamment petites, son image sera en général satisfaisante en ne transformant que les extrémités de ces segments; mais si ce n'est pas le cas, il faudra transformer des points supplémentaires des segments en question.

La fonction `subdivise[liste, eps]` transforme la liste de points `liste` en une autre liste de points représentant la même ligne brisée, la distance de deux points consécutifs étant toujours inférieure à `eps` (relativement à l'unité définie par le repère utilisateur).

Une idée qui vient naturellement à l'esprit avec *Mathematica* est d'utiliser une méthode dichotomique programmée de façon récursive :

```
In[] := subdiviseRec[liste_List, eps_] := N[liste] //.
          {a___, b : {_, _}, c : {_, _}, d___} /; (c - b) . (c - b) > eps^2 :>
           {a, b, (b + c)/2, c, d}
```

Malheureusement ce programme très concis est aussi très lent : la condition $(c-b).(c-b) > \text{eps}^2$ est évaluée un trop grand nombre de fois inutilement.

Nous construirons donc un programme structural :

```
In[] := subdivise[ liste_List, eps_ ] :=
        Module[{res = {}, a, b, n},
         Do[
          a = N[liste[[i]]]; b = N[liste[[i + 1]]];
           n = Ceiling[N[Sqrt[(b - a) . (b - a)] / eps]];
          If[
           n > 1,
           res = {res, Table[ a + (b - a) k / n, {k, 0, n - 1}]},
           res = {res, a}
          ],
          {i, 1, Length[liste] - 1}
         ];
         Partition[Flatten[{res, Last[liste]}], 2]
        ]
```

Notons la méthode utilisée dans la boucle pour construire la liste `res` : au lieu d'utiliser tout naturellement les fonctions `Join` ou `AppendTo`, on construit des listes imbriquées : `res = {res, elem}` , puis on reconstruit la structure désirée à l'aide de `Flatten`. Cette méthode est préconisée dans l'ouvrage de référence de Wolfram (2.4.4).

Il est possible de compiler le programme ci-dessus avec la version 3 de *Mathematica* (les versions antérieures ne supportent pas les listes); mais on ne pourra plus imbriquer de listes : la variable `res` est déclarée comme un tenseur d'ordre 2 (type `{_Real,2}`) et doit le rester pendant le calcul. On reviendra donc à `Join` et `AppendTo`; le gain est de 50% :

```
In[] := Clear[subdivise]
```

```
In[] := subdivise = Compile[{{liste, _Real, 2}, {eps, _Real}},
        Module[{res = {{0., 0.}}, a = {0., 0.}, b = {0., 0.}, n = 1},
         Do[
          a = N[liste[[i]]]; b = N[liste[[i + 1]]];
           n = Ceiling[N[Sqrt[(b - a) . (b - a)] / eps]];
          If[
           n > 1,
           res = Join[res, Table[ a + (b - a) k / n, {k, 0, n - 1}]],
          AppendTo[res, a]
          ],
          {i, 1, Length[liste] - 1}
         ];
         Rest[res]
        ]];
```

Voici un exemple d'utilisation de la fonction subdivise :

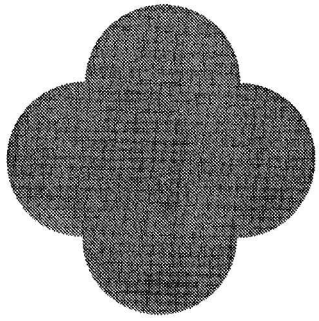
```
In[] := carreplein = Show[Graphics[{GrayLevel[.5],
            Rectangle[{-1, -1}, {1, 1}]}], AspectRatio -> Automatic];
```

(Figure non montrée)

Le rectangle `carreplein` doit être transformé en un polygone et la liste des sommets est fermée pour que la fonction `subdivise` agisse sur tous les côtés :

```
In[] := | Show[carreplein /. Rectangle[min_, max_] :>
          Polygon[Inversion[{0, 0}, 1] /@ subdivise[{min,
          {max[[1]], min[[2]]}, max, {min[[1]], max[[2]]}, min}, .01]]];
```

-Figure 6-



La fonction `transformePrimitives` ci-dessous effectue ce travail pour chacune des primitives qu'on peut rencontrer dans un graphique 2D :

```
In[] := | Clear[transformePrimitives]
```

```
In[] := | transformePrimitives[f_, gr_, eps_] := gr /. {
            Point[m_] :> Point[f[m]],
            Line[l_] :> Line[f /@ subdivise[l, eps]],
            Polygon[l_] :>
             Polygon[f /@ subdivise[Append[l, First[l]], eps]],
            Rectangle[min_, max_] :>
             Polygon[f /@ subdivise[{min, {max[[1]], min[[2]]},
                 max, {min[[1]], max[[2]]}, min}, eps]],
          Circle[c_, r_, a_: {0, 2 Pi}] :> Module[{h, p},
          p = If[Head[r] === List, r, {r, r}];
            h = N[2 Pi / Ceiling[2 Pi Max[r] / eps]];
              Line[Table[f[r {Cos[t], Sin[t]} + c],
                Evaluate[{t, Sequence @@ a, h}]]]],
          Disk[c_, r_, a_: {0, 2 Pi}] :> Module[{h, p},
          p = If[Head[r] === List, r, {r, r}];
            h = N[2 Pi / Ceiling[2 Pi Max[r] / eps]];
              Polygon[Table[f[r {Cos[t], Sin[t]} + c],
                Evaluate[{t, Sequence @@ a, h}]]]],
          Text[st_, m_, rest__] :> Text[st, f[m], rest]
            }
```

Il reste à mettre en place la fonction définitive; elle devra transmettre les options de la fonction `Graphics`; nous ajouterons quelques options propres à la fonction :

- `PlotPoints`; valeur par défaut : 25; la longueur maximum des segments de droites qui ne seront pas subdivisés avant transformation sera le quotient de la plus grande dimension du graphique par ce nombre.

- `Preimage`; valeur par défaut : `False`; avec la valeur `True`, le graphique initial et son transformé sont superposés.

- `PlotStyle`; valeur par défaut : `{{}, {}}`; la première liste peut contenir des directives qui s'appliqueront au graphique transformé, et la seconde liste s'appliquera à la pré-image si l'option précédente est à `True`.

Pour manipuler les options dans notre programme, nous utilisons le petit fichier `FilterOptions.m` de Roman E. Maeder ; `FilterOptions[fonc,opts]` sélectionne dans la liste d'options `opts`, celles attachées à la fonction `fonc` :

```
In[] := << Utilities`FilterOptions`
```

```
In[] := Options[TransformeGraphique] = Join[Options[Graphics],
          { PlotPoints -> 25, Preimage -> False, PlotStyle -> {{}, {}} }];
```

Le second argument peut être un objet de type `Graphics`, ou une simple liste de primitives; nous ramenons le premier cas au second :

```
In[] := Clear[TransformeGraphique]
```

```
In[] := TransformeGraphique[f_, gr_Graphics, opts___] :=
         TransformeGraphique[f, First[gr], opts,
          Sequence @@ Flatten[List @@ Rest[gr]]]
```

Le premier argument peut être une fonction ou une liste de fonctions; on se ramène au premier cas :

```
In[] := TransformeGraphique[f_List, gr_, opts___] := Show[
          Block[{$DisplayFunction = Identity},
           TransformeGraphique[#, gr, opts] & /@ f]]
```

Il reste à traiter le cas d'une fonction et d'une liste de primitives (ou d'une seule primitive) :

```
In[] := TransformeGraphique[f_, gr_, opts___] := Module[
          {eps, objet, image,
           n = PlotPoints /. {opts} /. Options[TransformeGraphique],
          style = PlotStyle /. {opts} /. Options[TransformeGraphique]
           },
        objet = Flatten[{gr}];
        eps = distmax[N[FullOptions[Graphics[objet], PlotRange]]] / n;
        image = transformePrimitives[f, objet, eps];
        If[Preimage /. {opts} /. Options[TransformeGraphique],
        Show[Graphics[{Join[Flatten[{style[[1]]}], image],
              Join[Flatten[{style[[2]]}], objet]},
         FilterOptions[Graphics, opts]]],
        Show[Graphics[Join[Flatten[{style[[1]]}], image],
             FilterOptions[Graphics, opts]]]
        ]
          ]
```

```
In[] := distmax[{{a_, b_}, {c_, d_}}] := √((b - a)^2 + (d - c)^2)
```

Lorsque la ou les transformations sont affines, la subdivision des segments devient inutile et fait perdre du temps; il peut être commode de rajouter une fonction qui construit des images de graphiques par des transformations affines :

```
In[] := Clear[AffineTransforme]
```

```
In[] := AffineTransforme[args__] := Block[{subdivise = (#1&)},
          TransformeGraphique[args]]
```

Bien sûr il faut que la transformation soit affine pour que cette fonction donne une image exacte. Cette fonction sera surtout avantageuse si la figure comporte de nombreux segments de droites ou polygones de longueur assez grande.

2.3.2 - Le fichier

Nous donnons seulement la structure du fichier; il suffira de compléter avec les programmes ci-dessus.

Avec la version 3, on fera un cahier comme expliqué au chapitre 3, qui sera converti automatiquement en fichier de commandes. Nous laissons également le lecteur perfectionner ce programme en introduisant des messages d'erreurs (pour un programme complet, voir la disquette d'accompagnement).

Fichier TransformeGraphique.m

```
In[] :=
BeginPackage[
 "Graphiques`TransformeGraphique`", "Geometrie`Transformations`"]

TransformeGraphique::usage = "TransformeGraphique[f,gr,options]
   dessine l'image du graphique 2D gr par la transformation f.
  gr peut être une liste de
   primitives ou une expression de type Graphics.
   f peut être une fonction ou une liste de fonctions.
  Outre les options de la fonction Graphics, on dispose
   aussi des options PlotPoints, Preimage, PlotStyle."
AffineTransforme::usage =
 "AffineTransforme[f,gr,options] dessine l'image
   du graphique 2D gr par la transformation affine f.
  Les options sont les
   mêmes que pour la fonction TransformeGraphique.
  Si la transformation f n'est
   pas affine, les images peuvent être inexactes."
PlotStyle::usage = PlotStyle::usage <>
  "Pour les fonctions TransformeGraphique et AffineTransforme,
    on entrera une liste {st1,st2} qui spécifie les
    styles de la figure image et de la figure objet."
Preimage::usage = "Preimage est
   une option pour les fonctions TransformeGraphique
   et AffineTransforme qui précise s'il faut tracer
   ou non la figure objet (valeur par défaut : False)."
Options[TransformeGraphique] = ...

Begin[`Private`]
Needs["Utilities`FilterOptions`"(* introduction en contexte caché *)

subdivise = ... (* on choisira la version compilée ou non *)
transformePrimitives[f_, gr_, eps_] := ...
distmax[{{a_, b_}, {c_, d_}}] := ...
TransformeGraphique[f_, gr_Graphics, opts___] := ...
TransformeGraphique[f_List, gr_, opts___] := ...
TransformeGraphique[f_, gr_, opts___] := ...
AffineTransforme[args__] := ...

End[]

Protect[TransformeGraphique, AffineTransforme]
EndPackage[]
```

Fin du fichier TransformeGraphique.m

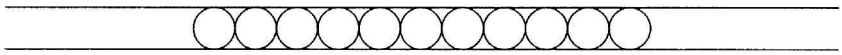
2.3.3 - Quelques exemples d'utilisation

```
In[] := << Graphiques`TransformeGraphique`
```

L'inversion permet de réaliser de belles images à partir de figures simples :

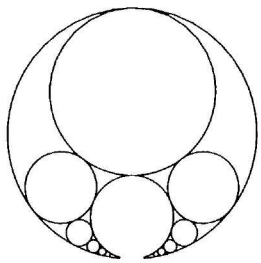
```
In[] := gr = Graphics[{Line[{{-20, -1}, {20, -1}}],
         Line[{{-20, 1}, {20, 1}}], Table[Circle[{i, 0}, 1],
        {i, -10, 10, 2}]}, {AspectRatio -> Automatic}];

In[] := Show[gr];
```

-Figure 7-



```
In[] := TransformeGraphique[Inversion[{0, -2}, 1], gr, PlotPoints -> 500,
         PlotRange -> All];
```

-Figure 8-

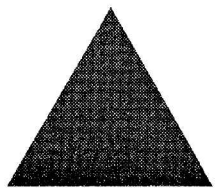


◆ **Le triangle de Sierpinsky :**

Nous allons montrer comment construire simplement le triangle de Sierpinsky.

```
In[] := ptA = {0, 0}; ptB = {1, 0};
        ptC = N[{Cos[Pi / 3], Sin[Pi / 3]}];

In[] := tri1 = Graphics[{GrayLevel[0.2], Polygon[{ptA, ptB, ptC}]},
          AspectRatio -> Automatic];

In[] := Show[tri1];
```

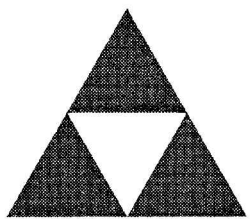
-Figure 9-



Nous transformons par 3 homothéties centrées aux sommets du triangle :

```
In[] := t1 = Homothetie[ptA, 1 / 2];
        t2 = Homothetie[ptB, 1 / 2];
        t3 = Homothetie[ptC, 1 / 2];

In[] := tri2 = AffineTransforme[{t1, t2, t3}, tri1];
```

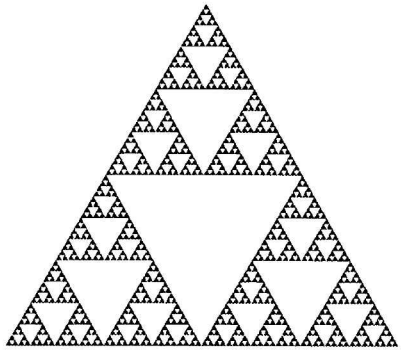
-Figure 10-



On veut itérer le procédé à l'aide de la fonction `Nest`; il faut procéder ainsi pour éviter le tracé des figures intermédiaires :

```
In[] := | Timing[tri6 = Show[Block[{$DisplayFunction = Identity},
        | Nest[AffineTransforme[{t1, t2, t3}, #]&, tri1, 6]]]][[1]]
```

-Figure 11-



```
Out[] = | 8.41667 Second
```

Nous traçons ensuite l'image de cette figure par la transformation suivante :

```
In[] := | transfo = AnalytiqueT[{x^2 - y, x + 2 x y}, {x, y}]
Out[] = | AnalytiqueT[{x^2 - y, x + 2 x y}, {x, y}]
```

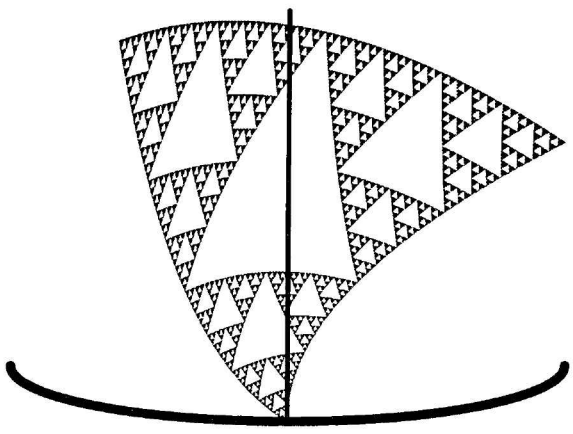
On aurait pu aussi programmer :

```
In[] := | transfo = AnalytiqueT[{#1^2 - #2 &, #1 + 2 #1 #2 &}]
```

Nous obtenons une belle voilure de pirogue; il reste à compléter le dessin :

```
        | TransformeGraphique[transfo, tri6,
        |   Epilog -> {
        |   Thickness[0.01],
In[] := |   Line[{{0, 0}, {0, 1.5}}],
        |   Thickness[0.02],
        |   Circle[{0, 0.2}, {1, 0.2}, {Pi, 2 Pi}]
        |   } ];
```

-Figure 12-



2.4 - Construction d'un fichier de transformation de graphiques 3D

Un graphique3D sera principalement à base de primitives `Point`, `Line`, `Polygon` et parfois `Cuboid`; cette dernière devra être convertie en liste de polygones.

Le principe de subdivision des lignes sera le même que dans le plan; la fonction `subdivise` sera conservée (remnomée `subdiviseLignes`).

Pour subdiviser des polygones, nous avons choisi de procéder en deux étapes :

– d'abord trianguler les polygones (qui, en général sont quadrangulaires)

– subdiviser les triangles par une méthode récursive : chaque triangle est partagé en 4 en introduisant le triangle des milieux.

Ce travail sera fait par la fonction `subdivisePoly`.

Il faudra deux valeurs différentes pour l'option `PlotPoints` : une ligne pourra être subdivisée sans ralentissement notable du programme, tandis que les polygones ne le seront que s'ils sont suffisamment grands; sinon on se contentera de les trianguler.

Nous donnons directement le programme, les fonctions ne néccessitant pas de commentaires particuliers

Le lecteur pourra là aussi construire avec la version 3 un cahier à convertir automatiquement.

2.4.1 - Le fichier

Fichier TransformeGraphique3D.m

In[] :=

```
BeginPackage["Graphiques`TransformeGraphique3D`",
 "Geometrie`Transformations`"]

TransformeGraphique3D::usage =
 "TransformeGraphique3D[f,gr,options] dessine
   l'image du graphique 3D gr par la transformation f.
  gr peut être une liste de
   primitives ou une expression de type Graphics.
   f peut être une fonction ou une liste de fonctions.
  Outre les options de la fonction Graphics3D,
   on dispose aussi des options PlotPoints, PlotStyle."
AffineTransforme3D::usage =
 "AffineTransforme3D[f,gr,options] dessine l'image
   du graphique 3D gr par la transformation affine f.
  Les options sont les
   mêmes que pour la fonction TransformeGraphique3D.
  Si la transformation f n'est
   pas affine, les images peuvent être inexactes."
PlotStyle::usage = PlotStyle::usage <> "Pour les
    fonctions TransformeGraphique3D et AffineTransforme3D,
    on entrera une directive ou une liste de directives
    pour spécifier le style de la figure image ."

Options[TransformeGraphique3D] = Join[Options[Graphics3D],
  { PlotPoints -> {100, 10}, PlotStyle -> {} }]
```

(la même cellule d'entrée précédente continue sur les deux pages suivantes)

```
Begin[`Private`]
Needs["Utilities`FilterOptions`"]

(* Fonctions auxilliaires -> *)
triangule[Polygon[s_ /; Length[s] == 3]] := {Polygon[s]}
triangule[Polygon[s_ /; Length[s] == 4]] :=
 Polygon /@ Partition[Append[s, First[s]], 3, 2]
triangule[Polygon[s_]] := With[{g = (Plus @@ s) / Length[s]},
 Polygon /@
   (Prepend[#, g]& /@ Partition[Append[s, First[s]], 2, 1]) ]

subdivisePoly[p_Polygon, eps_] := Flatten[triangule[p] //.
 Polygon[{a_, b_, c_} /; maxCarLong[{a, b, c}] > eps^2] :>
   Polygon /@ {{(c + a)/2, a, (a + b)/2}, {(a + b)/2, b, (b + c)/2},
      {(b + c)/2, c, (c + a)/2}, {(c + a)/2, (a + b)/2, (b + c)/2}}]

maxCarLong[s_] := Max @ (Apply[Plus, #^2]& /@
    (Apply[Subtract, #]& /@ Partition[Append[s, First[s]], 2, 1]))

subdiviseLignes = Compile[{{liste, _Real, 2}, {eps, _Real}},
Module[{res = {{0., 0., 0.}},
    a = {0., 0., 0.}, b = {0., 0., 0.}, n = 1},
 Do[
  a = N[liste[[i]]]; b = N[liste[[i + 1]]];
   n = Ceiling[N[Sqrt[(b - a) . (b - a)] / eps]];
  If[
   n > 1,
   res = Join[res, Table[a + (b - a) k / n, {k, 0, n - 1}]],
  AppendTo[res, a]
  ], {i, 1, Length[liste] - 1}
 ]; Rest[res]
]] (* on peut remplacer par le programme non compilé *)

transformePrimitives3D[f_, gr_, {epsl_, epsp_}] := gr /. {
   Point[m_] :> Point[f[m]],
   Line[l_] :> Line[f /@ subdiviseLignes[l, epsl]],
   Polygon[s_] :> Map[f, subdivisePoly[Polygon[s], epsp], {3}],
   Cuboid[min_, max_] :> Map[
     Map[f, subdivisePoly[#, epsp], {3}]&,
     cuboidToPolygon[{min, max}]],
   Text[st_, m_, rest__] :> Text[st, f[m], rest] }

cuboidToPolygon[{min_, max_}] := With[
  {faces = {{1, 2, 4, 3}, {3, 4, 8, 7}, {5, 7, 8, 6}, {1, 5, 6, 2},
     {2, 6, 8, 4}, {1, 3, 7, 5}},
  sommets =
    Flatten[Outer[List, ##]& @@ Transpose[{min, max}], 2]},
  Polygon /@ (sommets[[#]]& /@ faces)]

distmax3D[{{a_, b_}, {c_, d_}, {e_, f_}}] :=
 Sqrt[(b - a)^2 + (d - c)^2 + (f - e)^2]
```

```
(* Fonctions publiques *)
TransformeGraphique3D[
  f_, gr_Graphics3D, opts___] := TransformeGraphique3D[f,
  First[gr], opts, Sequence @@ Flatten[List @@ Rest[gr]]]

TransformeGraphique3D[f_List, gr_, opts___] := Show[
  Block[{$DisplayFunction = Identity},
   TransformeGraphique3D[#, gr, opts] & /@ f]]

TransformeGraphique3D[f_, gr_, opts___] := Module[
  {epsl, epsp, dmax, nl, np,
   objet = Flatten[{gr}],
  style = PlotStyle /. {opts} /.
      Options[TransformeGraphique3D] },
  {nl, np} = PlotPoints /. {opts} /.
    Options[TransformeGraphique3D];
dmax = distmax3D[N[FullOptions[Graphics3D[objet], PlotRange]]];
  epsl = dmax / nl; epsp = dmax / np;
Show[Graphics3D[
    {style, transformePrimitives3D[f, objet, {epsl, epsp}]} ,
    FilterOptions[Graphics3D, opts]] ] ]

AffineTransforme3D[args__] := Block[
  {subdiviseLignes = (#1&), subdivisePoly = ({#1}&)},
  TransformeGraphique3D[args] ]

End[]
Protect[TransformeGraphique3D, AffineTransforme3D]
EndPackage[]
```

Fin du fichier TransformeGraphique3D.m

2.4.2 - Quelques exemples d'utilisation

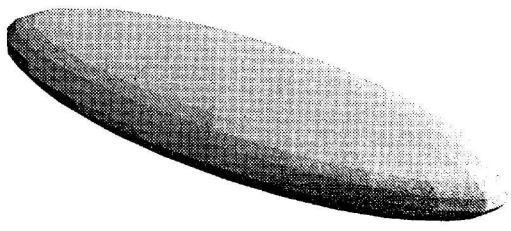
```
In[] := Needs["Graphics`Shapes`"]
```

```
In[] := << Graphiques`TransformeGraphique3D`
```

Nous construisons l'image d'une sphère par une affinité orthogonale d'axe (x'x) :

```
In[] := AffineTransforme3D[Affinite[{{-1, 0, 0}, {1, 0, 0}}, 0.25],
         Sphere[], Boxed -> False, PlotStyle -> EdgeForm[]];
```

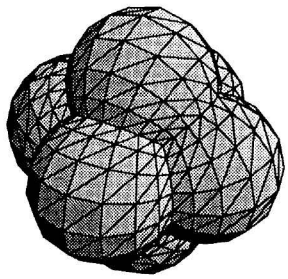
-Figure 13-



Nous transformons un cube par inversion; ce n'est évidemment pas une transformation affine :

```
In[] := TransformeGraphique3D[Inversion[{0, 0, 0}, 1],
         {Cuboid[{-1, -1, -1}, {1, 1, 1}]}, Boxed -> False];
```

-Figure 14-



Avec l'entrée ci-dessous, nous avons successivement le tracé de la courbe de départ, puis de son image :

```
In[] := AffineTransforme3D[Affinite[{{1, 0, 0}, {0, 1, 0}}, 0.5],
         ParametricPlot3D[{Cos[t], Sin[t], Sqrt[t]}, {t, 0, 20}],
         PlotStyle -> RGBColor[1, 0, 0]];
```

(Figure non montrée)

◆ **L'éponge de Sierpinsky :**

On procède comme pour le triangle de Sierpinsky ci-dessus, mais la figure de départ est un cube et on fait agir la transformation définie par 20 homothéties centrées aux 8 sommets et aux milieux des 12 arêtes, de rapport $\frac{1}{3}$.

On définit les coordonnées des sommets, puis, ceux-ci étant numérotés dans cet ordre, on définit par ces numéros, les faces et les arêtes du cube :

```
In[] := sommets = Flatten[Outer[List, {0, 1}, {0, 1}, {0, 1}], 2]

Out[] = {{0, 0, 0}, {0, 0, 1}, {0, 1, 0}, {0, 1, 1}, {1, 0, 0}, {1, 0, 1},
         {1, 1, 0}, {1, 1, 1}}

In[] := faces = {{1, 2, 4, 3}, {3, 4, 8, 7}, {5, 7, 8, 6}, {1, 5, 6, 2},
         {2, 6, 8, 4}, {1, 3, 7, 5}}

Out[] = {{1, 2, 4, 3}, {3, 4, 8, 7}, {5, 7, 8, 6}, {1, 5, 6, 2},
         {2, 6, 8, 4}, {1, 3, 7, 5}}

In[] := aretes = {{1, 2}, {2, 4}, {4, 3}, {3, 1}, {5, 6}, {6, 8},
         {8, 7}, {7, 5}, {1, 5}, {2, 6}, {4, 8}, {3, 7}};
```

On définit le cube de départ :

```
In[] := cube0 =
         Graphics3D[Polygon[sommets[[#]]]& /@ faces, Boxed -> False];

In[] := Show[cube0];
```

(Figure non montrée)

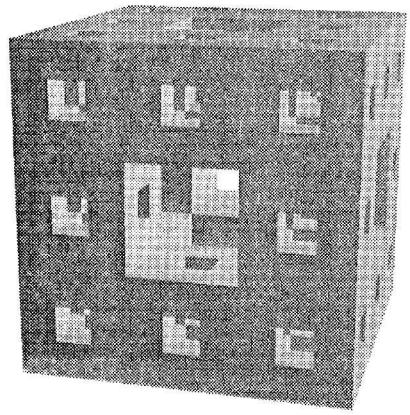
Puis on définit la famille d'homothéties (le lecteur pourra afficher les résultats pour mieux comprendre) :

```
In[] := centres = Join[sommets,
          (Plus @@ #) / 2 & /@ (sommets[[#]]&) /@ aretes];

In[] := homotheties = Homothetie[#, 1/3]& /@ centres;

In[] := cube1 = AffineTransforme3D[homotheties, cube0,
          PlotStyle -> EdgeForm[], ViewPoint -> {0.831, -3.169, 0.845}];
```

(Figure non montrée)

```
In[] := cube2 =
         AffineTransforme3D[homotheties, cube1, PlotStyle -> EdgeForm[]];
```

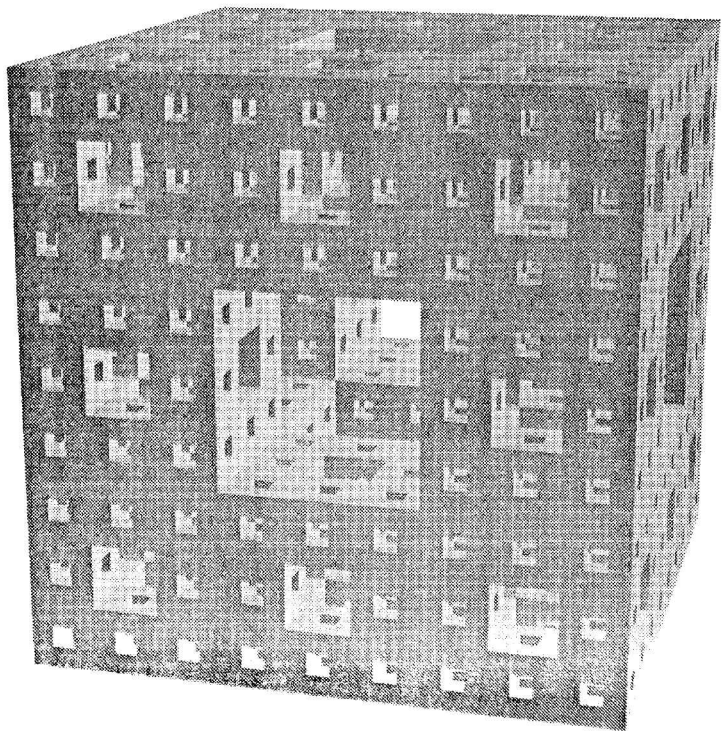
-Figure 15-



Cette dernière itération nécéssite beaucoup de mémoire vive (plus de 100 Mo sur un Macintosh PPC 7500) :

```
In[] := cube3 =
         AffineTransforme3D[homotheties, cube2, PlotStyle -> EdgeForm[]];
```

-Figure 16-



◆ **Une famille de sphères obtenues par inversion :**

Il s'agit de transformer par inversion la figure formée par une sphère, les six sphères tangentes de même rayon, puis six suites de sphères toujours de même rayon, dont les centres sont alignés avec la sphère centrale et les sphères tangentes, enfin les deux plans parallèles tangents situés dessus et dessous.

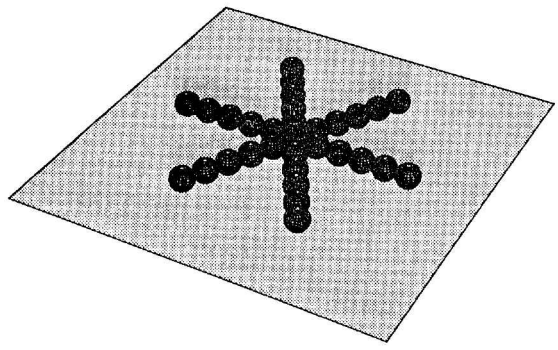
Les deux plans seront transformés en deux sphères tangentes au pôle de l'inversion et les 6 rangées de sphères formeront des colliers entre ces deux dernières.

Le plan supérieur n'est pas tracé (donc la sphère extérieure dans la figure transformée non plus) pour des raisons évidentes.

```
In[] := spheres[n_, rest___] := Graphics3D[{
          Polygon[{{-2 (n + 3), -2 (n + 3), -1}, {2 (n + 3), -2 (n + 3), -1},
            {2 (n + 3), 2 (n + 3), -1}, {-2 (n + 3), 2 (n + 3), -1}}],
          Sphere[rest], Table[TranslateShape[Sphere[],
            {r Cos[k π / 3], r Sin[k π / 3], 0}], {k, 0, 5}, {r, 2, 2 n, 2}]
          }, Boxed -> False]
```

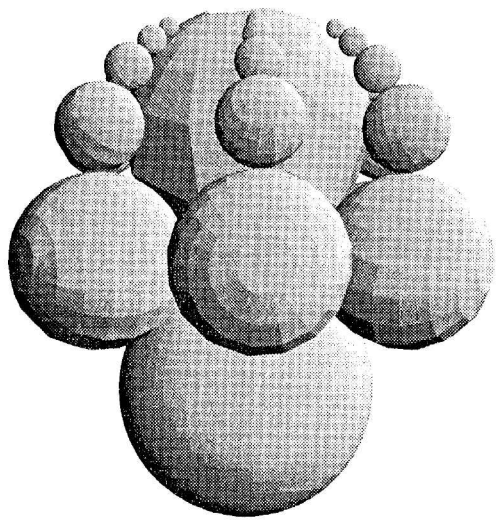
```
In[] := objet = Show[spheres[5, 1, 60, 40]];
```

-Figure 17-



```
In[] := TransformeGraphique3D[Inversion[{0, 0, 3}, 4], objet,
        PlotPoints -> {100, 15}, PlotStyle -> EdgeForm[],
         ViewPoint -> {1.491, -3.037, 0.042}];
```

-Figure 18-



Bibliographie

Ouvrages généraux sur *Mathematica* :

Stephen Wolfram. *The Mathematica Book*. Wolfram Media ; Cambridge University Press (1996)

Wolfram Research. *Standard Add-on Packages*. Wolfram Media ; Cambridge University Press (1996)

Roman. Maeder. *Programming in Mathematica*. Addison-Wesley (1991)

Th. W. Gray, J. Glynn. *Exploring Mathematics with Mathematica*. Addison-Wesley (1991)

Yvon Poitevineau. *Mathematica 3 par la pratique*. Eyrolles (1997)

Ouvrages sur les graphiques dans *Mathematica* :

Tom Wickham-Jones. *Mathematica graphics*. Course notes ; Mathematica conferences (1992)

O. Gloor, B. Amrhein, R. Maeder. Illustrated Mathematics: Visualization *of Mathematical Objects with Mathematica*. Springer-Verlag (1995)

Cameron Smith, Nancy Blachman. *The Mathematica graphics Guidebook*. Addison-Wesley (1995)

Tom Wickham-Jones. *Mathematica graphics: Techniques & Applications*. Springer-Verlag (1994)

Quelques ouvrages de géométrie :

Marcel Berger. *Géométrie*. Cedic/Fernand Nathan (1977)

Robert Deltheil, Daniel Caire. *Compléments de géométrie*. J.B. Baillère et Fils (1951)

Eugène Rouché, Ch. de Comberousse. *Traité de géométrie*. Gauthier-Villars (1900)

Michael F. Barnsley. *Fractals everywhere*. Academic Press (1993)

H. S. M. Coxeter. *Introduction to Geometry*. John Wiley

Index

Les mots en gras sont des noms de symboles (soit des symboles standard *Mathematica*, soit des symboles définis dans des fichiers de commandes standards ou définis dans cet ouvrage).

Auteur : POITEVINEAU Y., Groupe Informatique de l'IREM de Montpellier (travail réalisé avec le soutien de la DISTNB).

Titre : Réaliser des graphiques et faire de la géométrie avec Mathematica.

Editeur : IREM de Montpellier, Université des Sciences et Techniques du Languedoc.

Date : Octobre 1998.

Type de document : Fascicule IREM.

Support : Papier.

Type d'utilisateur : Enseignants du secondaire et des classes préparatoires, universitaires et toute personne voulant utiliser le calcul formel.

Résumé :

Cette brochure passe en revue les possibilités offertes par le logiciel Mathematica dans le domaine du graphisme et étudie quelques applications en géométrie.

L'étude détaillée et thématique des principales commandes graphiques permettra au lecteur de construire des graphiques très élaborés et de les animer. Une initiation à la programmation graphique permet l'élaboration de fichiers de commandes.

Le chapitre sur les coniques permet de tracer ces courbes et de placer leurs axes, leurs directrices et leurs foyers. Il comporte également le moyen d'obtenir l'équation réduite de la conique à partir de son équation cartésienne dans un repère quelconque.

L'auteur a élaboré une bibliothèque de transformations géométriques aussi bien en 2D, qu'en 3D, permettant d'exécuter du calcul géométrique formel ou de réaliser des images d'objets graphiques par transformations géométriques.

Une disquette d'accompagnement contient le texte de l'ouvrage et permet d'effectuer à l'aide du logiciel les manipulations proposées. Elle comporte également l'intégralité des fichiers de commandes étudiées.

Mots clés : calcul formel, classes préparatoires, coniques, dessin 3D, géométrie, graphiques, logiciel Mathematica, lycée, mathématiques, programmation, université.