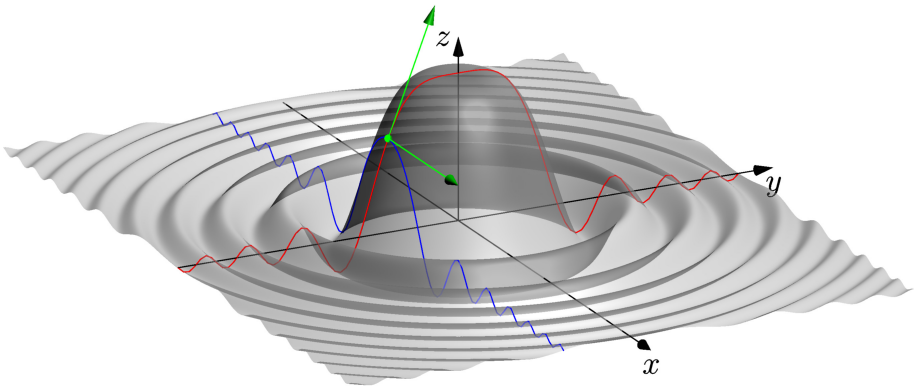
Bruno Colombel

http://www.mathbko.marot.org/index.php?src=asy3d/acc

maurice72@ymail.com

# Asymptote 3D




Version 1.1
28 mai 2011

# **Asymptote** – 3D

Bruno M. Colombel

28 mai 2011

## Remerciements

Je tiens à remercier chaleureusement toutes les personnes qui m'ont aidé pour réaliser cette documentation.

Tout d'abord, **Christophe Grospellier**, alias chellier, qui m'a fournit les environements de sa documentation ce qui m'a permis de ne pas trop m'occuper de la mise en page et de la forme et dont l'aide sur le forum **Asymptote** de Mathematex est toujours précieuse.

**Gaétan Marris** qui m'a autorisé à piller son site d'exemples et dont la disponibilité en cas de besoin n'a jamais été démentie. En quelque sorte mon parrain en **Asymptote**.

**Olivier Guibé**, alias OG, dont les réponses sur le forum **Asymptote** de Mathematex permettent toujours d'y voir plus clair et d'approfondir ses connaissances.

**Philippe Ivaldi** qui m'a autorisé à « emprunter » quelques codes de son site d'exemples.

Et enfin, **Frédéric Marot** qui m'a remis dans le droit chemin en me faisant (re)découvrir LaTeX. Il héberge cette documentation sur son site.

Cette documentation a été créée sous **Fedora 14** avec **TeXlive 2010** avec l'extension **kfonts** de Christophe Caignaert ; l'utilisation du logiciel pygments a permis la coloration des codes. Merci à tous les développeurs et tous les contributeurs de ces logiciels **libres** !

# Table des matières

# Introduction

Cette documentation est la suite logique du travail de Christophe Grospellier et de son Démarrage rapide. Elle ne se prétend ni complète ni très savante. Elle se veut juste une aide, aussi complète que possible, pour mettre le pied à l'étrier et pouvoir travailler et créer un peu avec ce formidable langage qu'est **Asymptote** en trois dimensions.

En deux dimensions, on dessine ce qui est en face de nous. On se place en général perpendiculairement au plan dans lequel on travaille. En trois dimensions, au contraire, la vision que l'on a d'un objet dépend de l'emplacement de l'observateur. Si on veut représenter l'espace en deux dimensions, on peut faire varier le plan frontal et l'angle de fuite si l'on dessine une perspective cavalière, on peut choisir différents points à l'infini dans le cas de géométrie projective. Bref, il va falloir définir notre point de vue.

Démarrer la 3D avec **Asymptote** est difficile ; beaucoup de paramètres vont arriver dont on ne sait pas toujours quoi faire. il va falloir choisir un angle de vue, une lumière, des couleurs pour rendre attrayante et esthétique notre figure dans l'espace. Une sphère projetée sur un plan n'est qu'un cercle, un pavé n'est qu'un rectangle.

L'essentiel des routines en deux dimensions sont adaptée et généralisée en trois dimensions dans le module **three.asy** puis le module **graph3.asy** nous permettra de tracer des courbes et/ou des surfaces définies par des équations ; les grilles seront abordées dans **grid3.asy** et les solides usuels dans **solids.asy**.
L'essentiel de cette documentation, les quatre premières parties, sera donc consacré à **three.asy**. La 1re partie, **Avant le code**, abordera les réglages « visuels » et les options de compilation. Puis viendront les outils géométriques de **three.asy**.
Les trois parties suivantes porteront principalement sur les modules **graph3.asy** et **solids.asy** où seront abordés les repères, les surfaces, les courbes gauches et les solides de révolution.
Nous finirons par quelques compléments avec quelques exemples d'utilisation de modules annexes.

Je suis utilisateur de Linux et n'ai jamais testé **Asymptote** sous Windows. Les lignes de commandes seront donc utilisables uniquement sous Linux. Pour les heureux détenteurs de Windows seven ou Vista, il semble que le moment de rejoindre la communauté du libre soit venu. Un dual boot est si vite installé !
Ce document a été créé sous le système d'exploitation **Fedora 14** et avec **TeXLive 2010**. L'installation se fait facilement sous linux [1] et en particulier sous **Fedora** comme on peut le lire dans la documentation officielle [3] :

```
Fedora users can easily install the most recent version of Asymptote with the command :
yum --enablerepo=rawhide install asymptote
```

Toutes les figures de cette documentation ont été compilées avec la version 2.08 d'**Asymptote**.

On pourra trouver de l'aide :
- Dans la documentation officielle d'**Asymptote** [3].
- Sur le site d'exemples de Philippe Ivaldi où les exemples sont triés par module [4].
- Sur le site d'exemples de Gaétan Marris où les exemples sont triés par thème [5].
- Sur le site de Christophe Grospellier avec son **Démarrage rapide** et ses exemples [1].
- Dans **Asymptote : un survol** d'Olivier Guibé [2].
- Sur le forum officiel (en anglais) d' **Asymptote**
- Sur le forum (en français) consacré à **Asymptote** hébergé par le site Mathematex.

# I – Avant le code

Avant le code « graphique » d'une figure dans l'espace, un fichier **Asymptote** contient un certain nombre de lignes qui définissent quelques réglages importants : point de vue de l'observateur, lumière, options de compilation, ...
Un fichier commence en général par des lignes du type :

```
import three;                                   //ou graph3 ou solids
currentprojection=perspective(5,4,2);          // Projection par défaut
currentlight=Headlamp;                          // Eclairage par défaut
```

La ligne **import three**; permet de charger le module **three** dans lequel sont définies la plupart des routines 3D. Il existe essentiellement deux autres modules pour l'espace : **graph3** et **solids**. À noter que le module **graph3** charge **three** et que le module **solids** charge **three** et **graph3**.
Charger un de ces modules est donc **obligatoire** pour dessiner dans l'espace.

Les deux autres lignes sont optionnelles. Elles définissent la projection, c'est-à-dire, le point de vue de l'observateur et la lumière. En l'absence de ces lignes, les paramètres par défaut sont utilisés. Ce sont ceux donnés ci-dessus.

Peuvent suivre des lignes consacrées à la compilation. On peut les insérer directement dans le fichier .asy pour éviter de surcharger la ligne de commande lors de la compilation. Cela ressemble à :

```
settings.tex="latex";           // Moteur LaTeX utilisé pour la compilation (latex, pdflatex, ...)
settings.outformat="eps";       // Format de sortie ; eps par défaut
settings.prc=true;              // Format PRC de la figure ; vrai par défaut
settings.render=-1;             // Rendu des figures ; -1 par défaut
```

Si les deux premières options ne servent occasionnellement, il est important de connaître les deux suivantes.
La plupart du temps, ces options de compilation peuvent se définir dans le fichier ou en ligne de commande. Seules quelques options n'existent qu'en ligne de commande. Par exemple, si on souhaite une sortie pdf, avec 4 pixels par bp (**render**=4) sans le format **prc**, on peut au choix écrire au début du fichier .asy :

```
settings.outformat="pdf";
settings.prc=false;
settings.render=4;
```

ou écrire en ligne de commande :

```
asy -f pdf -noprc -render=4 mondessin.asy
```

où −f (**format**) désigne le format de sortie (eps par défaut, pdf, ...)

Dans la suite, nous allons détailler les différents points de vue de l'observateur, les différentes lumières disponibles puis les options de compilation.

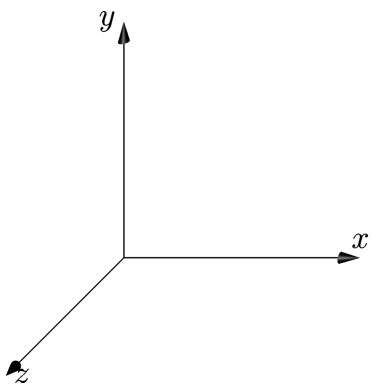
## 1 . Point de vue : perspective et projection

Différents types de projection des figures créées en deux dimensions sont prédéfinies.

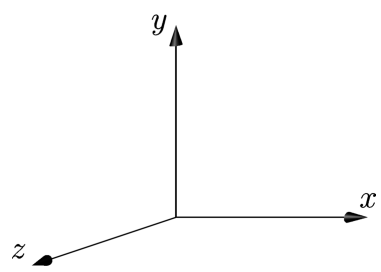
```
oblique(real angle);
```

Le point de coordonnées $(x;y;z)$ est projeté en $(x-0.5z;y-0.5z)$.
Si l'argument optionnel **angle** est donné alors l'axe $(Oz)$ est tracé avec cet angle (en degré).
**oblique** est équivalent à **obliqueZ**.

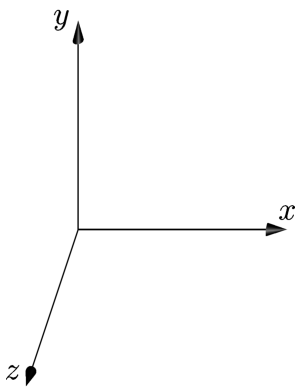
```
currentprojection=oblique;
```




```
currentprojection=oblique(30);
```



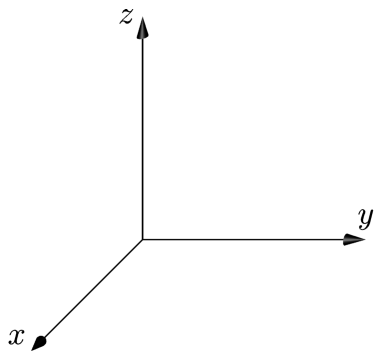

```
currentprojection=oblique(60);
```



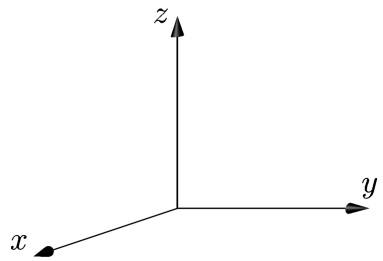
```
obliqueX(real angle);
```

Le point de coordonnées $(x;y;z)$ est projeté en $(y-0.5x;z-0.5x)$.
Si l'argument optionnel **angle** est donné alors l'axe $(Ox)$ est tracé avec cet angle (en degré).
**obliqueX** peut être en particulier utilisé pour se rapprocher de la perspective cavalière. l'angle de fuite peut être régler mais **Asymptote** ne permet pas de gérer pour l'instant le coefficient de perspective. Une solution est apportée par Christophe Grospellier en annexe D de son démarrage rapide.

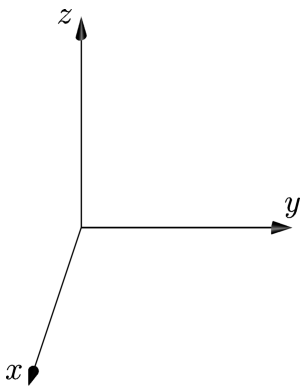
```
currentprojection=obliqueX;
```




```
currentprojection=obliqueX(30);
```



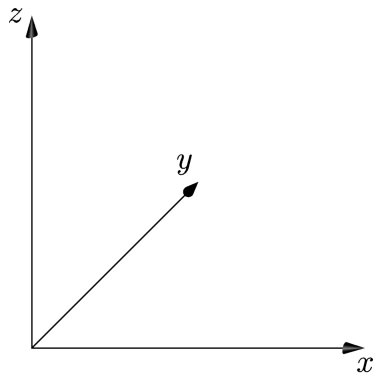

```
currentprojection=obliqueX(60);
```



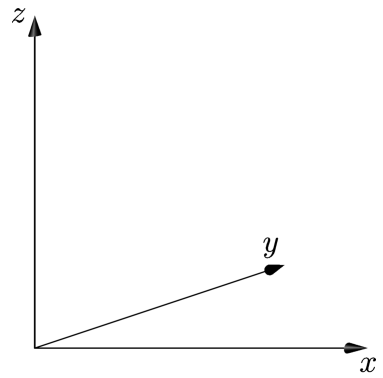

```
obliqueY(real angle);
```

Le point de coordonnées $(x;y;z)$ est projeté en $(x-0.5y;z-0.5y)$.
Si l'argument optionnel **angle** est donné alors l'axe $(Oy)$ est tracé avec cet angle (en degré).

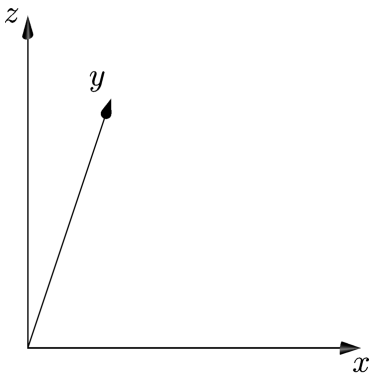
```
currentprojection=obliqueY;
```




```
currentprojection=obliqueY(30);
```




```
currentprojection=obliqueY(60);
```




Deux autres types de projection en deux dimensions existent. Il s'agit de la projection **orthographic** et de **perspective** :

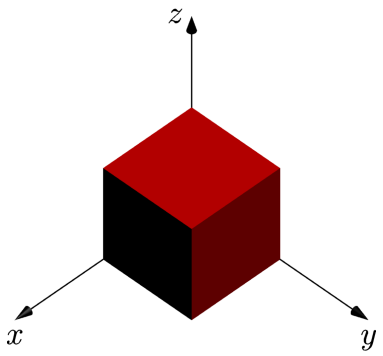
```
orthographic(triple camera,
             triple up=Z,
             triple target=0,
             real zoom=1,
             pair viewportshift=0,
             bool showtarget=true,
             bool center=false)
```

Projette en deux dimensions en utilisant le point de vue d'un observateur situé à l'infini dans la direction **camera** et regardant **target**.
Le projeté de deux droites (ou lignes) parallèles sera également formé de deux droites parallèles.
Si **showtarget=true** le volume englobera le point **target**.
Le paramètre **up=Z** signifie que l'axe des cotes est vertical et dirigé vers le haut.

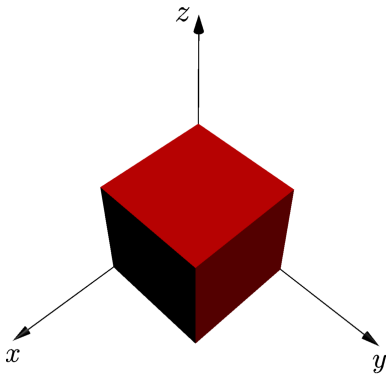
```
perspective(triple camera,
            triple up=Z,
            triple target=0,
            real zoom=1,
            real angle=0,
            pair viewportshift=0,
            bool showtarget=true,
            bool autoadjust=true,
            bool center=autoadjust)
```

Projette en deux dimensions du point de vue d'un observateur positionné en **camera** qui regarde **target**.
Si **autoadjust=true**, **camera** est ajustée de façon à être en dehors du volume représenté.
Si **center=true** la cible **target** est modifiée de façon à être au centre du volume représenté.
Le paramètre **up=Z** signifie que l'axe des cotes est vertical et dirigé vers le haut.

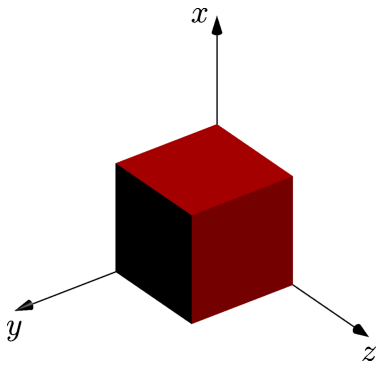
```
currentprojection=orthographic(1.5,1.5,2);
```



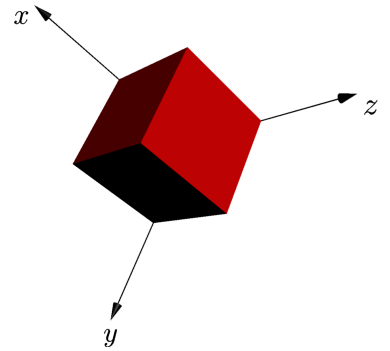

```
currentprojection=perspective(1.5,1.5,2);
```



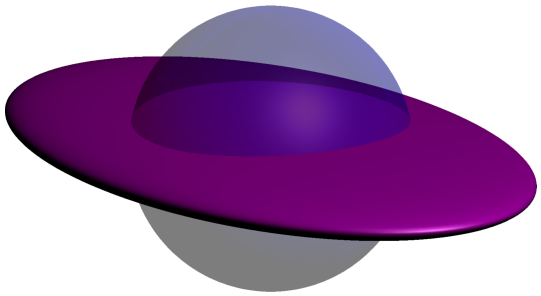

```
currentprojection=orthographic((1.5,1.5,2),up=X);
```



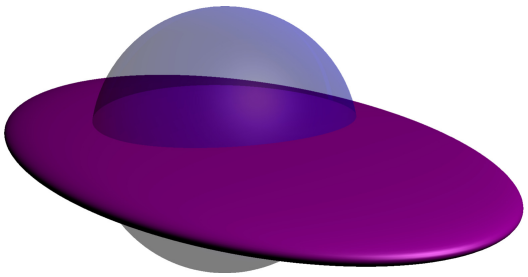

```
currentprojection=perspective((1.5,1.5,2),up=X+Z);
```




```
currentprojection=orthographic(2,3,1);
```



```
currentprojection=perspective(2,3,1);
```



**Remarques :**

1. Les explications de ces deux dernières routines sont succinctes. On se référera à la documentation officielle [3] pour plus de détails.
2. Ces routines utilisent l'argument **camera**. Pour pouvoir choisir le meilleur point de vue possible, on peut utiliser la compilation :

   ```
   asy -V figure.asy
   ```

   On pourra alors « manipuler » la figure et déterminer le meilleur angle de vue. Voir la section consacrée à la compilation.
3. La projection par défaut est :

   ```
   perspective(5,4,2)
   ```

## 2 . Éclairage

**Asymptote** permet aussi de régler l'éclairage des figures en trois dimensions par le biais de la structure **light** :

```
struct light {
  real[][] diffuse;
  real[][] ambient;
```

```
  real[][] specular;
  pen background=nullpen; // Background color of the 3D canvas.
  real specularfactor;
  bool viewport; // Are the lights specified (and fixed) in the viewport frame?
  triple[] position; // Only directional lights are currently implemented.

  transform3 T=identity(4); // Transform to apply to normal vectors.

  bool on() {return position.length > 0;}
}
```

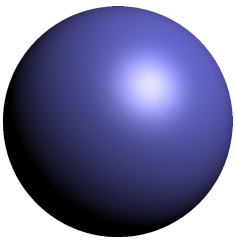
Trois **light** sont prédéfinies :

```
light Viewport=light(ambient=gray(0.1),
                     specularfactor=3,
                     viewport=true,(0.25,-0.25,1));

light White=light(new pen[] {rgb(0.38,0.38,0.45),rgb(0.6,0.6,0.67),rgb(0.5,0.5,0.57)},
                  specularfactor=3,
                  new triple[] {(-2,-1.5,-0.5),(2,1.1,-2.5),(-0.5,0,2)});

light Headlamp=light(gray(0.8),
                     ambient=gray(0.1),
                     specular=gray(0.7),
                     specularfactor=3,
                     viewport=true,dir(42,48));
```

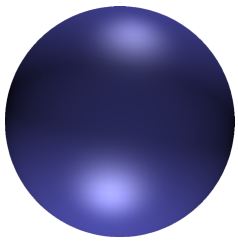
L'éclairage par défaut est

```
currentlight=Headlamp;
```

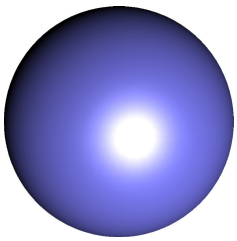
On définit en gros la luminosité générale avant **specularfactor** et l'emplacement de la ou des source(s) lumineuse(s) après.



`currentlight=Headlamp;`

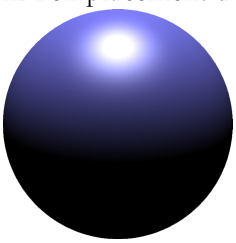


`currentlight=White;`

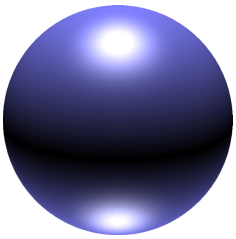


`currentlight=Viewport;`

On peut aussi définir l'emplacement d'une ou de plusieurs sources lumineuses :



`currentlight=((0,0,3));`



`currentlight=light((0,0,3),(0,0,-3));`

On peut bien sur se préparer sa propre lumière. Pour cela, il faut fouiller un peu plus et regarder la signification des listes de réels **ambient**, **diffuse** ou **specular**. On pourra se référer à la routine **material** du fichier **three_light.asy** et surtout faire beaucoup de tests !

## 3 . Compilation

Comme pour la 2D, le dessin créé peut être généré au format eps (par défaut) ou au format pdf. La compilation au format pdf offre la possibilité d'utiliser le format 3D PRC ce qui permer d'« activer » les figures et de les « manipuler ». Il faut dans ce cas utiliser une version d'Adobe Reader supérieure ou égale à 9.
Il existe aussi les options **−render=** de compilation permettant d'obtenir un rendu plus ou moins fin. Malheureusement, on se heurte parfois à la limitation de sa carte graphique et on se retrouve avec des bandes noires sur notre fichier sortie ! Il

faut s'armer de patience et quelquefois revoir ses ambitions à la baisse.
Une autre solution est d'utiliser le rendu de OpenGL. Cette option présente beaucoup d'avantages, en particulier celui de pouvoir déterminer l'angle de vue (**camera**, **target**, ...) et de régler la plupart des problèmes de compilation (bandes noires, etc.).
Regardons d'un peu plus près ces alternatives. Pour plus de détails, rendez-vous sur la documentation officielle. [3]

1. **Les options -render=**
   On peut spécifier l'option **−render=n** où $n$ est un entier lors de la compilation pour définir la résolution à $n$ pixels par **bp** pour le fichier de sortie. Si la valeur de $n$ est négative, elle est interprétée comme étant égale à $|2n|$ pour une sortie en eps ou pdf.

   ```
   asy -render=n mondessin.asy
   ```

   Vous pouvez aussi écrire :

   ```
   settings.render=4;
   ```

   au début de votre fichier .asy.
   La qualité du résultat dépend de la qualité de votre carte graphique. Ajouter l'option **−glOptions=−indirect** peut parfois résoudre vos problèmes ...
2. **Sortie OpenGL** : Pour visualiser vos dessins avec la sortie OpenGL, utiliser la compilation suivante (en ligne de commande uniquement) :

   ```
   asy -V mondessin.asy
   ```

   avec les réglages par défaut **outformat**="" et **render**=−1.
   Si votre carte graphique donne des signes de faiblesse essayez :

   ```
   asy -V -glOptions=-indirect mondessin.asy
   ```

   À l'apparition du résultat (dans une fenêtre « sans-titre ») on peut alors manipuler la figure avec la souris.
   Vous pouvez aussi double-cliquer sur le bouton droit de la souris. Il apparaît alors un menu déroulant composé de :

   - (h) Home
   - (f) Fitscreen
   - (x) Xspin
   - (y) Yspin
   - (z) Zspin
   - (s) Stop
   - (m) Mode
   - (e) Export
   - (c) Camera
   - (p) Play
   - (r) Reverse
   - ( ) Step
   - (q) Quit

   où les lettres entre parenthèses sont les raccourcis clavier.
   Cette méthode s'avère très pratique pour définir les paramètres de **currentprojection**. En effet, lorsque vous choisissez **camera** (**c**) s'affiche alors dans le terminal les informations sur :

   ```
   currentprojection=orthographic(
   camera=(5,4,2),
   up=(-0.00342938575262755,-0.00274350860210204,0.0140604815857729),
   target=(8.88178419700125e-16,8.88178419700125e-16,-4.44089209850063e-16),
   zoom=1);
   ```

   Ce sont les paramètres du **CODE 1**. On peut donc ainsi facilement déterminer puis utiliser le point de vue qui nous convient le mieux avec un simple « copier - coller ».
3. **Format PRC**
   Pour intégrer le format PRC dans votre document pdf, il faut configurer **Asymptote**. Vous pouvez écrire les lignes :

   ```
   settings.prc=true,
   settings.outformat="pdf"
   ```

   au début de votre fichier .asy ce qui est équivalent à la ligne de commande :

```
asy -prc -f pdf mondessin.asy
```

Attention, par défaut **settings.prc=true** et votre dessin ne peut être visionné qu'avec Adode Reader et une version supérieure ou égale à 9. Avec un autre lecteur de pdf vous obtiendrez une page blanche. Pour pallier ce défaut, on peut combiner ces compilations avec un **settings.render=n** afin d'obtenir une image fixe « au-dessus » du format PRC ce qui permet au dessin d'être visionné avec d'autre lecteurs pdf qu'Adobe Reader (qui malgré toutes ses qualités a l'immense défaut d'être propriétaire).
Par exemple, vous pouvez compiler votre fichier avec la ligne :

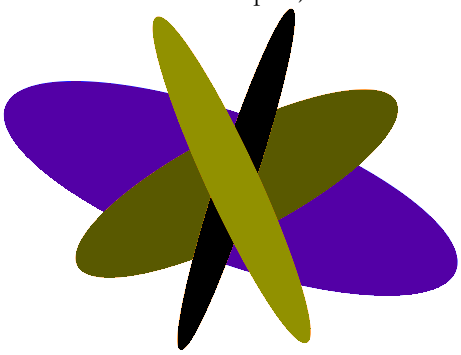
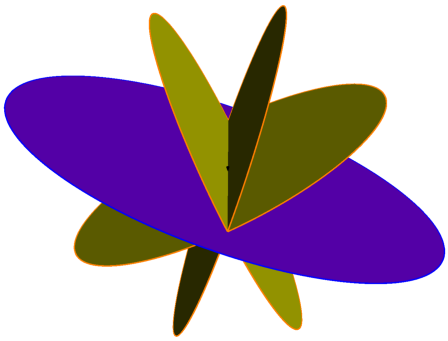
```
asy -prc -f pdf -render=4 mondessin.asy
```

Le fichier .pdf généré par **Asymptote** et visualisé avec Adobe Reader peut alors être manipulé.
Cependant, l'intégration d'une image PRC dans un .pdf généré avec LaTeX ne peut se faire avec un \includegraphics. Il faut se rendre au paragraphe suivant.

**Remarque :**
l'option −**prc** est l'option par défaut. La ligne suivante suffit :

```
asy -f pdf -render=4 mondessin.asy
```

4. **Compilation avec -render=0**
   Cette option projette la figure en deux dimensions (pour les formats .eps ou .pdf). Seules les surfaces « simples »acceptent une telle compilation. En effet, les faces cachées ne sont pas gérées.
   Comparons le même code compilé, l'un avec −**render**=0 et l'autre avec −**render**=4. Je vous laisse deviner lequel.

### En pratique

Dans la pratique, à chacun ses petites habitudes et ses petites préférences. Je ne pourrais donc pas faire mieux que d'expliquer comment je procède. J'utilise essentiellement la ligne de commande qui a l'avantage de signifier les erreurs. J'ai une légère préférence pour la sortie en .eps, sans doute une réminiscence de l'époque où je guerroyais en PSTricks.
Pour la 3D, c'est ce qui nous intéresse, voila comment je procède. Toutes les figures de ce documents ont été compilées de la même façon :

1. Je compile en eps avec une sortie OpenGL :

   ```
   asy -V -render=4 figure.asy
   ```

   C'est le moyen le plus efficace que j'ai trouvé pour éviter les raies noires et autres problèmes dus à la carte graphique. Le cas échéant, je peux modifier l'angle de vue et l'« exporter » dans mon code (touche (c) pour **camera**). Si l'image me convient je l'exporte avec la touche (e) (export) et obtient l'image figure.eps.
   L'option −**render**=4 est préférable sinon l'image peut être de moindre qualité.
2. Si le document LaTeX est compilé avec pdfLaTeX, je recommence le même procédé avec la commande :

   ```
   asy -noprc -f pdf -V -render=4 figure.asy
   ```

   L'option −**noprc** est nécessaire pour que la fenêtre OpenGL s'ouvre, sinon la figure s'affiche directement avec Adobe Reader.

**Remarques :**
- La véritable raison pour laquelle je ne compile pas directement avec :

```
asy -noprc -f pdf -V -render=4 figure.asy
```

est due à l'ouverture systématique d'Adobe Reader après chaque export ce que je trouve un peu lourd. En effet, Adobe reader n'a pas la légèreté d'une visionneuse pdf comme evince par exemple. Pour la sortie eps, je n'ai pas ce problème puisque la visionneuse est ghostview qui n'est pas installé chez moi. Il faudrait que je modifie les réglages par défaut d'**Asymptote** mais j'ai la flemme.
Cela peut se faire avec le fichier **config.asy** ; voir le désormais célèbre démarrage rapide.

- Si jamais vous compilez sans l'option **−V**, et que votre carte graphique vous pose des problèmes (bandes noires, copie d'écran en fond ou autre), essayez de compiler en augmentant le $n$ dans **−render=n** où $n = 2, 4, 6, 8, \ldots$. Dans ce cas, la taille de l'image augmente exponentiellement.

## 4. Obtenir un PRC dans un pdf avec LaTeX

### Avec une compilation extérieure au document

Si pour une raison ou pour une autre, vous préférez créer votre image en dehors du document LaTeX, vous devrez compiler votre fichier figure.asy avec les options suivantes pour une compilation latex :

```
asy -inlineimage figure -render=4
```

ou pour une compilation pdflatex :

```
asy -inlineimage figure -render=4 -tex pdflatex
```

Lors de la compilation sont créés les fichiers :

figure.pre
figure.tex
figure_0.pdf
figure+0.js
figure+0.prc
figure+0.tex
figure+0_0.tex

Ces fichiers sont créés à chaque compilation mais sont habituellement effacés.

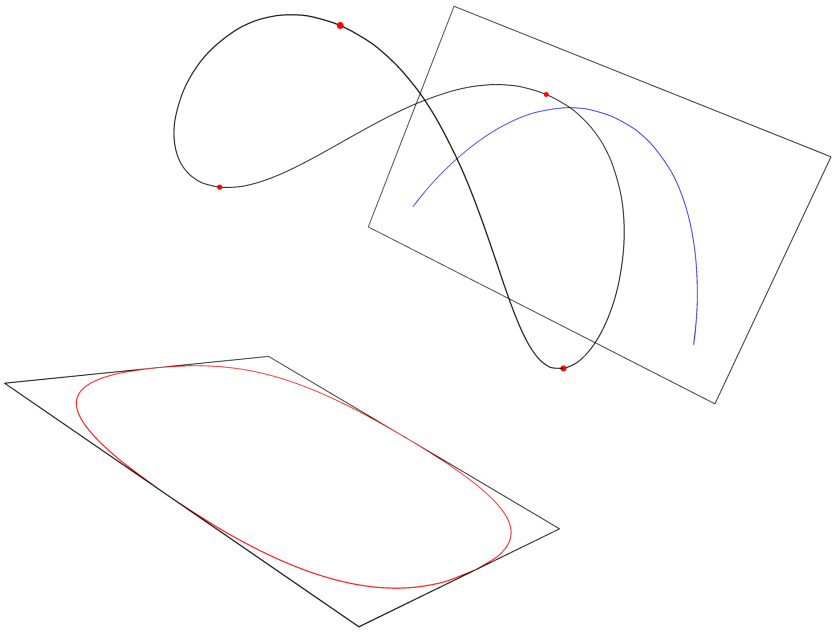
Il faut ensuite ajouter dans le préambule du document LaTeX la ligne :

```
\input figure.pre
```

puis dans le document à l'emplacement où l'on veut insérer le dessin :

```
\input figure.tex
```

Voici un exemple :



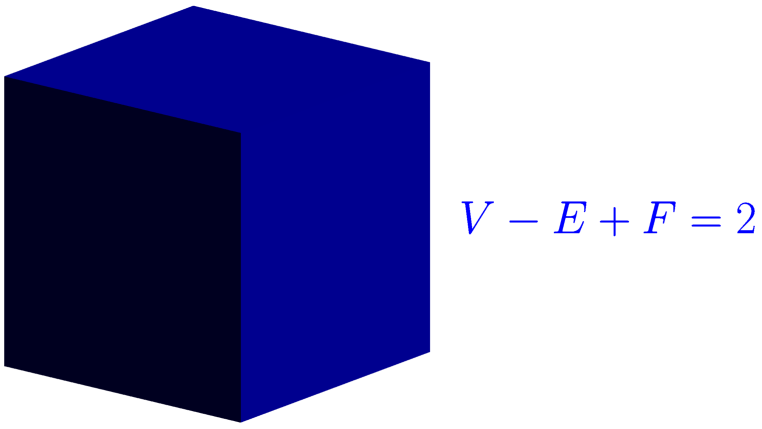
Si vous visionnez ce document avec Adobe Reader dans une version supérieure ou égale à 9, vous pouvez manipuler le dessin après avoir cliqué dessus.

**Remarque :**
Peuvent (res)surgir les problèmes de compilation évoqués plus haut. Pour obtenir cette figure, j'ai dû m'y reprendre à deux fois. Après la première compilation, des raies noires défiguraient le dessin. Une deuxième compilation en augmentant l'argument **−render** a été tentée avec succès :

```
asy -inlineimage figure -render=6 -tex pdflatex
```

Il faut parfois s'armer de patience avant d'obtenir le résultat escompté. La logique n'est pas toujours respectée et il arrive qu'avec exactement les mêmes options, cela fonctionne une fois et pas la suivante !

### Directement dans le document LaTeX




Voici l'exemple de la documentation officielle [3]. Le code est entre les balises :

```
\begin{asy}
...
\end{asy}
```

On peut alors compiler le fichier .asy généré avec :

```
asy -render=4 figure.asy
```

avant de relancer latex ou pdflatex.

On peut aussi utiliser **latexmk**. Dans ce cas, l'option **inline=true** est nécessaire.

```
\begin{asy}[inline=true]
...
\end{asy}
```

**Remarque :**
Sur mon ordinateur, on peut effectivement activer la figure mais il y a comme un gros problème de zoom ...
Il s'agit, si l'on en croit le forum officiel d'**Asymptote**, d'un problème inhérent à certaines distributions linux lié à Adobe Reader. Le problème ne vient pas d'**Asymptote**.
Si le problème survient chez vous, il faut « dézoomer » avec la roulette de la souris ...

### En conclusion

Insérer un PRC dans un fichier LaTeX n'est pas si simple ! Pour le premier exemple, sept fichiers spécifiques (en plus du .asy) ont été créés et je dois les garder. Pour le deuxième exemple, neuf fichiers spécifiques de créés ; ce qui fait 16 pour deux figures. Je me vois mal faire la même chose pour la soixantaine de dessins de ce document ! ! !

Et pour la route :

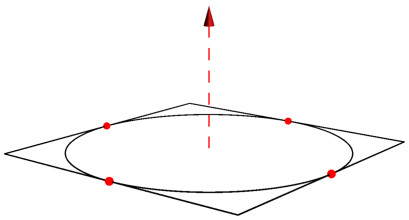


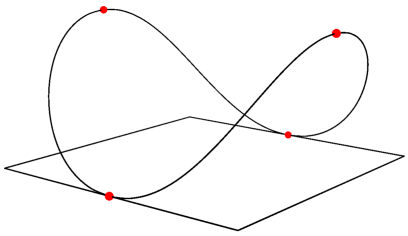
Toujours ce problème de zoom (avec la première méthode cette fois ci !)

# II – Module three

## 1 . guide3, path3, surface et size3

Le module **three** d'**Asymptote** met en place tout ce qui est spécifique à trois dimensions. Premièrement, il généralise les chemins et les guides à la troisième dimension. En outre, il définit les nouveaux types **guide3**, **path3** et **surface**.
Les **guide** en trois dimensions (et donc les **path** en 3 dimensions) acceptent la même syntaxe qu'en deux dimensions. On utilisera néanmoins les **triple** (**x**,**y**,**z**) à la place des **pair** (**x**,**y**).
La documentation officielle nous fournit les deux exemples :





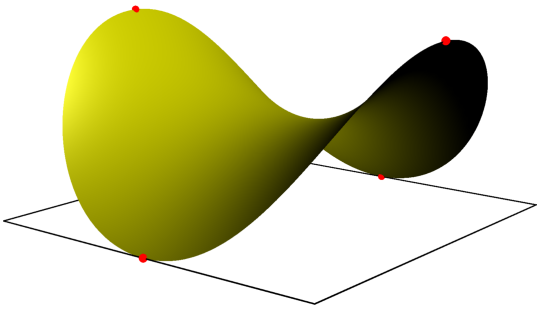
CODE 1

```
import three;

size(150);

path3 g=(1,0,0)..(0,1,0)..(-1,0,0)
                        ..(0,-1,0)..cycle;
draw(g);
draw(O--Z,red+dashed,Arrow3);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)
                        --(-1,1,0)--cycle));
dot(g,red);
```

CODE 2

```
import three;

size(150,0);

path3 g=(1,0,0)..(0,1,1)..(-1,0,0)
                        ..(0,-1,1)..cycle;
draw(g);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)
                        --(-1,1,0)--cycle));
dot(g,red);
```

Si le chemin est fermé, on peut alors dessiner la surface délimitée par ce chemin :



CODE 3

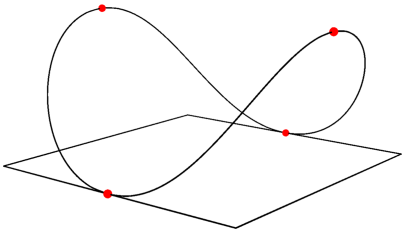
```
import three;

size(200,0);

path3 g=(1,0,0)..(0,1,1)..(-1,0,0)..(0,-1,1)..cycle;
draw(surface(g),yellow);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle));
dot(g,red);
```

grâce à la routine :

```
surface surface(path3 external, triple[] internal=new triple[],
triple[] normals=new triple[], pen[] colors=new pen[],
bool3 planar=default);
```

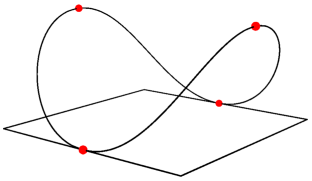
On peut spécifier la « taille » maximale de chaque dimension en utilisant :

```
void size3(picture pic=currentpicture, real x, real y=x, real z=y,
bool keepAspect=pic.keepAspect);
```



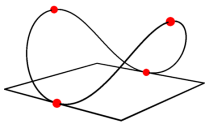
CODE 4

```
import three;

size3(10cm, 10cm, 2cm);

path3 g=(1,0,0)..(0,1,1)..(-1,0,0)
                        ..(0,-1,1)..cycle;
draw(g);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)
                        --(-1,1,0)--cycle));
dot(g,red);
```



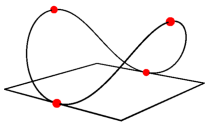
CODE 5

```
import three;

size3(3cm, 3cm, 2cm);

path3 g=(1,0,0)..(0,1,1)..(-1,0,0)
                        ..(0,-1,1)..cycle;
draw(g);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)
                        --(-1,1,0)--cycle));
dot(g,red);
```



CODE 6

```
import three;

size3(2cm, 10cm, 10cm);

path3 g=(1,0,0)..(0,1,1)..(-1,0,0)
                        ..(0,-1,1)..cycle;
draw(g);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)
                        --(-1,1,0)--cycle));
dot(g,red);
```



CODE 7

```
import three;

size3(10cm, 2cm, 10cm);

path3 g=(1,0,0)..(0,1,1)..(-1,0,0)
                        ..(0,-1,1)..cycle;
draw(g);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)
                        --(-1,1,0)--cycle));
dot(g,red);
```

**Remarque :**
On peut aussi utiliser comme en 2D la commande **size**().

## 2. Objets de base : cercle, arc, plan

Mais le module **three** fait bien plus que cela. Tout d'abord, afin de rendre les notations plus naturelles, les points

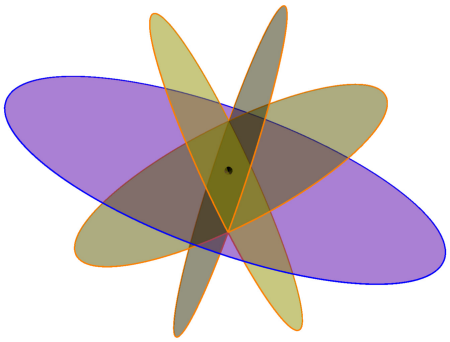
$$\mathbf{O}=(0,0,0),\ \mathbf{X}=(1,0,0),\ \mathbf{Y}=(0,1,0)\ \text{ et }\ \mathbf{Z}=(0,0,1)$$

sont définis dans ce module ainsi que le cercle unitaire du plan XY :

```
path3 unitcircle3=X..Y..-X..-Y..cycle;
```

Plus généralement, un cercle de centre **c**, de rayon **r** et orthogonal au vecteur **normal** est défini par :

```
path3 circle(triple c, real r, triple normal=Z);
```



CODE 8

```
import three;

currentprojection=perspective(
camera=(6.35944335631399,-0.681158669873566,2.02568040477462),
up=(-0.00378023886887436,-0.00368741420586603,0.010682089649285),
target=(-0.00223526244854888,-0.00167200918287946,0.00893217448712846),
zoom=1,
angle=12.6420760030802,
autoadjust=false);

size(6cm);

dot(0);

path3 c1=unitcircle3;
draw(c1, blue);
draw(surface(c1), purple+opacity(0.5));

path3 c2=circle(0, 0.75, Z-Y);
draw(c2, orange);
draw(surface(c2), yellow+opacity(0.5));

path3 c2=circle(0, 0.75, Y);
draw(c2, orange);
draw(surface(c2), yellow+opacity(0.5));

path3 c2=circle(0, 0.75, Z+Y);
draw(c2, orange);
draw(surface(c2), yellow+opacity(0.5));
```

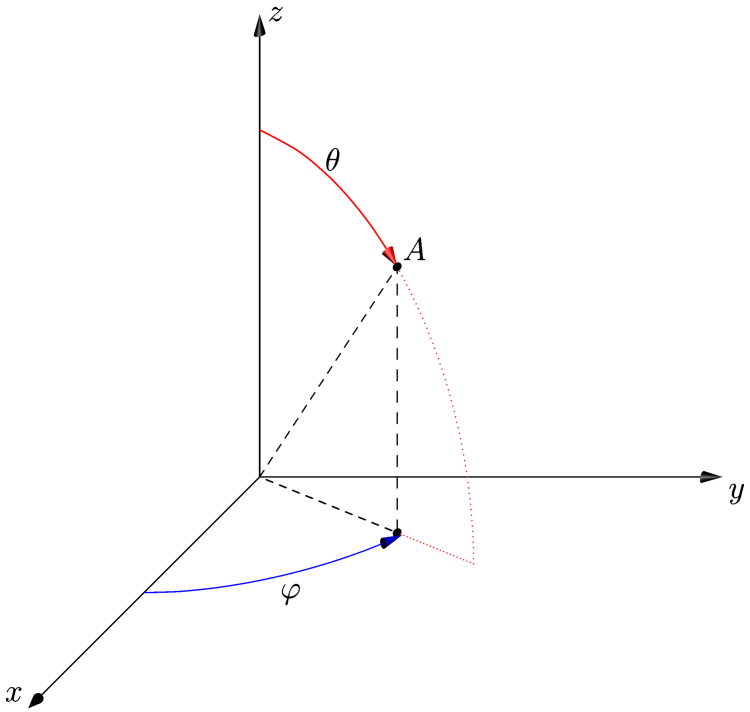
**Remarque :**
Les paramètres de la commande **currentprojection** ont été obtenus grâce à la compilation :

```
asy -V monfichier.asy
```

Suite logique du cercle, l'arc de cercle et plus si affinité :

```
path3 arc(triple c, real r,
          real theta1, real phi1,
          real theta2, real phi2,
          triple normal=0);
```

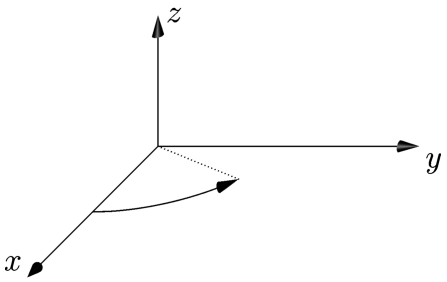
de centre **c**, de rayon **r**, du point **c+r*dir(theta1,phi1)** au point **c+r*dir(theta2,phi2)**, dans le sens des aiguilles d'une montre perpendiculairement à **cross(dir(theta1,phi1),dir(theta2,phi2))** où **cross(u,v)** est le produit vectoriel des vecteurs **u** et **v**. L'angle **theta** correspond à l'angle avec l'axe $(Oz)$ et **phi** à l'angle par rapport à l'axe $(Ox)$ :




CODE 9

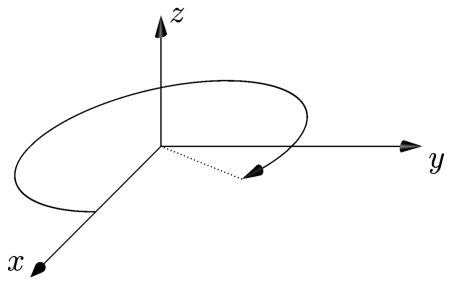
```
import three;
currentprojection=obliqueX;
size(10cm);

draw(O--2*X,Arrow3);      // En attendant
draw(O--2*Y,Arrow3);      // d'avoir les axes
draw(O--2*Z,Arrow3);      // avec graph3.asy
label("$x$", 2*X, NW);
label("$y$", 2*Y, SE);
label("$z$", 2*Z, E);

real theta1=0, theta2=40;
real phi1=0, phi2=60;

path3 arc1=arc(O, 1.5, theta1, phi2, theta2, phi2);
draw(arc1, red, Arrow3);
draw(arc(O, 1.5, theta2, phi2, 90, phi2),red+dotted);

path3 arc2=arc(O, 1, 90, phi1, 90, phi2);
draw(arc2,blue,Arrow3);

triple pA=(1.5*cos(phi2*pi/180)*sin(theta2*pi/180),
           1.5*sin(phi2*pi/180)*sin(theta2*pi/180),
           1.5*cos(theta2*pi/180));
triple projA=(1.5*cos(phi2*pi/180)*cos((90-theta2)*pi/180),
              1.5*sin(phi2*pi/180)*cos((90-theta2)*pi/180),
              0);

dot("$A$", pA, NE);
dot(projA);

draw(O--pA--projA--cycle, dashed);
draw(projA--(1.5*cos(phi2*pi/180),1.5*sin(phi2*pi/180),0),red+dotted);
label("$\varphi$", (cos((phi2/2)*pi/180), sin((phi2/2)*pi/180), 0), SE);
label("$\theta$", (1.5*cos(phi2*pi/180)*sin(theta2*pi/360),
                   1.5*sin(phi2*pi/180)*sin(theta2*pi/360),
                   1.5*cos(theta2*pi/360)), N);
```

On peut spécifier le sens de parcours de l'arc avec les arguments optionnels **CW** (clockwise) et **CCW** (counter-clockwise) :




CODE 10

```
import three;
currentprojection=obliqueX;
size(6cm);

draw(0--2*X,Arrow3);    // En attendant
draw(0--2*Y,Arrow3);    // d'avoir les axes
draw(0--Z,Arrow3);      // avec graph3.asy
label("$x$", 2*X, NW);
label("$y$", 2*Y, SE);
label("$z$", Z, E);

path3 arc1=arc(0,1,
          90,0,
          90,60,
          CCW);          // counter-clockwise
draw(arc1,Arrow3);

draw(0--(cos(pi/3),sin(pi/3),0),dotted);
```



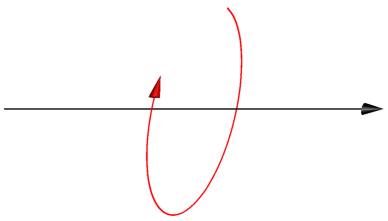

CODE 11

```
import three;
currentprojection=obliqueX;
size(6cm);

draw(0--2*X,Arrow3);    // En attendant
draw(0--2*Y,Arrow3);    // d'avoir les axes
draw(0--Z,Arrow3);      // avec graph3.asy
label("$x$", 2*X, NW);
label("$y$", 2*Y, SE);
label("$z$", Z, E);

path3 arc1=arc(0,1,
          90,0,
          90,60,
          CW);               // clockwise
draw(arc1,Arrow3);

draw(0--(cos(pi/3),sin(pi/3),0),dotted);
```

On peut également construire un arc de cercle de centre **c** passant par les points v1 et v2 en utilisant la routine :

```
path3 arc(triple c, triple v1, triple v2,
          triple normal=0,bool direction=CCW);
```

à condition que $|\,v1 - \mathbf{c}\,| = |\,v2 - \mathbf{c}\,|$.



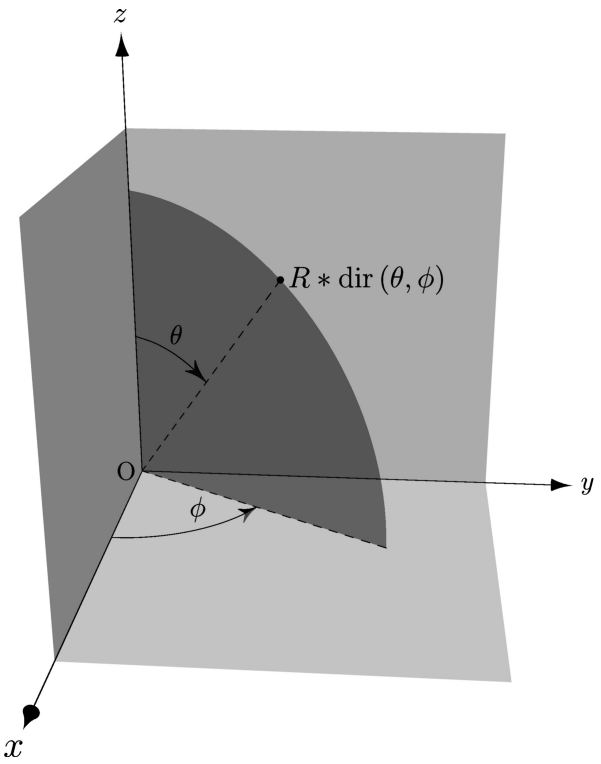
CODE 12

```
import three;
currentprojection=oblique;
size(5cm);

draw(0--2*X,Arrow3);

triple pA, v1, v2;
       pA=(1,0,0);
       v1=(1,sqrt(2)/4,-sqrt(2)/4);
       v2=(1,sqrt(2)/4,sqrt(2)/4);
path3 arc1=arc(pA, v1, v2, CW);
draw(arc1,red,Arrow3);
```

Les coordonnées sphériques utilisées pour les arcs peuvent s'utiliser pour définir des points avec la routine

**dir(phi, theta)**

comme nous montre l'exemple de Philippe Ivaldi [4] :



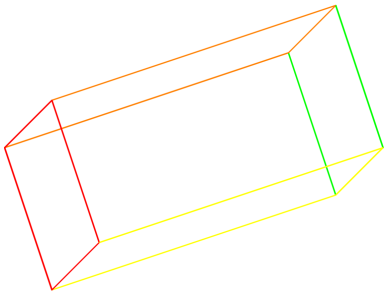

CODE 13

```
import three;
size(8cm,0);
real radius=1, theta=37, phi=60;

currentprojection=perspective(4,1,2);

// Planes
pen bg=gray(0.9)+opacity(0.5);
draw(surface((1.2,0,0)--(1.2,0,1.2)--(0,0,1.2)--(0,0,0)--cycle),bg,bg);
draw(surface((0,1.2,0)--(0,1.2,1.2)--(0,0,1.2)--(0,0,0)--cycle),bg,bg);
draw(surface((1.2,0,0)--(1.2,1.2,0)--(0,1.2,0)--(0,0,0)--cycle),bg,bg);

real r=1.5;
draw(Label("$x$",1), O--r*X, Arrow3(HookHead3));
draw(Label("$y$",1), O--r*Y, Arrow3(HookHead3));
draw(Label("$z$",1), O--r*Z, Arrow3(HookHead3));
label("$\rm O$", (0,0,0), W);

triple pQ=radius*dir(theta,phi); // Point Q
// triple pQ=radius*expi(radians(theta),radians(phi)); // Point Q
draw(O--radius*dir(90,phi)^^O--pQ, dashed);
dot("$R*\mathrm{dir}\left(\theta,\phi\right)$",pQ);

// Arcs
draw("$\theta$", reverse(arc(O,0.5*pQ,0.5*Z)), N+0.3E, Arrow3(HookHead2));
draw("$\phi$", arc(O,0.5*X,0.5*(pQ.x,pQ.y,0)), N+0.3E, Arrow3(HookHead2));

path3 p3=O--arc(O,radius,0,phi,90,phi)--cycle;
draw(surface(p3), blue+opacity(0.5));
```

Autre routine prédéfinie :

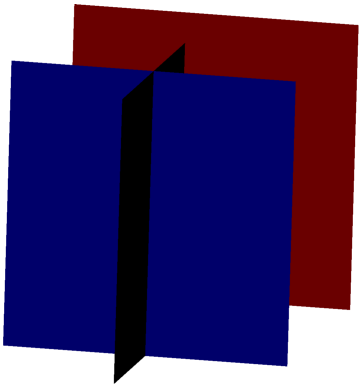
```
path3 plane(triple u, triple v, triple O=0);
```

qui permet de représenter le plan défini par les vecteurs **u** et **v** passant par **O**.
Cette routine définit le chemin **path3 O−−O+u−−O+u+v−−O+v−−cycle**



CODE 14

```
import three;
currentprojection=oblique;
size(5cm);

triple u,v,w;
       u=(1,1,1);
       v=(1,-1,1);
       w=cross(u,v);

path3 pl1=plane(u,v,0);
path3 pl2=plane(u,w,0);
path3 pl3=plane(u,w,v);
path3 pl4=plane(u,v,w);
draw(pl1,red);
draw(pl2,orange);
draw(pl3,yellow);
draw(pl4,green);
```



CODE 15

```
import three;
real a,b,c,d,e,f;
     a=0.0109149612364031;
     b=0.00234242554012969;
     c=0.00233392384594572;
     d=0.00583059272986634;
     e=0.000834265095552367;
     f=0.0152885564527493;
currentprojection=orthographic(
                        camera=(a,b,c),
                        up=(d,e,f),
                        target=(0,0,0),
                        zoom=1);
size(5cm);

path3 pl1=plane(Y,Z,0);
path3 pl2=plane(Y,Z,X);
path3 pl3=plane(X,Z,(X+Y)/2);
draw(surface(pl1),red);
draw(surface(pl2),blue);
draw(surface(pl3),yellow);
```

**Remarque :**
Le vecteur normal d'un plan défini par les vecteurs **u** et **v** (et d'un point) peut être déterminé par

**cross**(**u**, **v**)

qui calcule le produit vectoriel des vecteurs **u** et **v**.

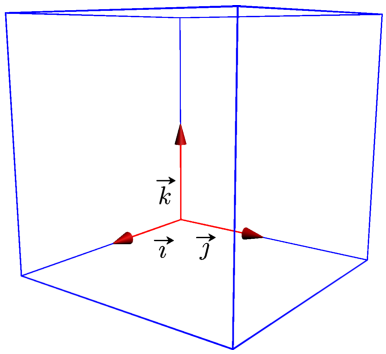
## 3 . Boîtes et unit*

La boîte de sommets opposés v1 et v2 peut être dessinée avec :

```
path3[] box(triple v1, triple v2);
```

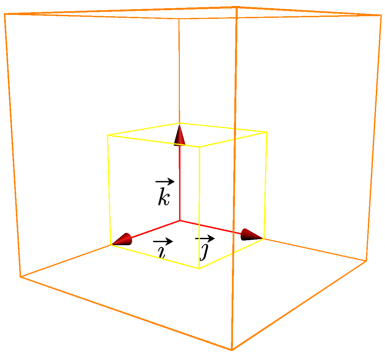
Une application toute simple est la boîte unité :

```
path3[] unitbox=box(0,(1,1,1));
```




CODE 16

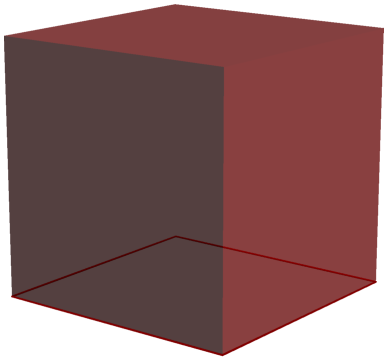
```
import three;
usepackage("esvect");
size(5cm);

// En attendant d'avoir les axes
// avec graph3.asy
draw(O--X, red+linewidth(0.7pt), Arrow3);
draw(O--Y, red+linewidth(0.7pt), Arrow3);
draw(O--Z, red+linewidth(0.7pt), Arrow3);
label("$\vv{\imath}$", 0.5*X, SE);
label("$\vv{\jmath}$", 0.5*Y, SW);
label("$\vv{k}$", 0.5*Z, SW);

draw(box(0,2*(X+Y+Z)),blue);
```




CODE 17

```
import three;
usepackage("esvect");
size(5cm);

// En attendant d'avoir les axes
// avec graph3.asy
draw(O--X, red+linewidth(0.7pt), Arrow3);
draw(O--Y, red+linewidth(0.7pt), Arrow3);
draw(O--Z, red+linewidth(0.7pt), Arrow3);
label("$\vv{\imath}$", 0.5*X, SE);
label("$\vv{\jmath}$", 0.5*Y, SW);
label("$\vv{k}$", 0.5*Z, SW);

draw(unitbox,yellow);
//En attendant d'avoir scale3
draw(box(0,2*(X+Y+Z)),orange);
```

Et dans la famille **unit**, on demande :

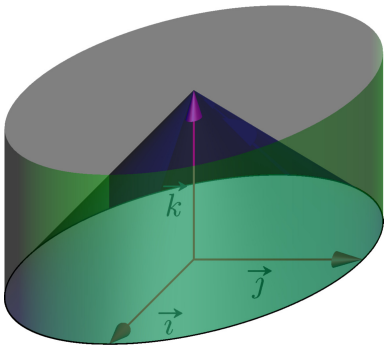


CODE 18

```
import three;
size(5cm);

draw(unitcube,red+opacity(0.5));
draw(unitsquare3,brown+linewidth(0.7pt));
```

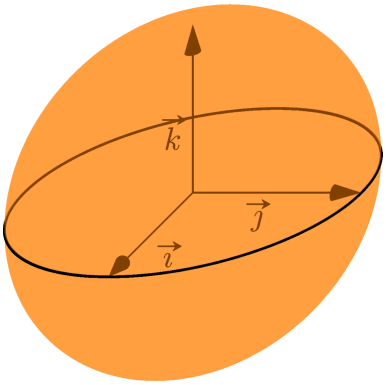
CODE 19

```
import three;
currentprojection=obliqueX;
size(5cm);

patch p1=unitfrustum(0.5,1),
      p2=unitfrustum(2,3),
      p3=unitfrustum(4,4.5);

draw(surface(p1),red);
draw(surface(p2),blue);
draw(surface(p3),yellow);
```



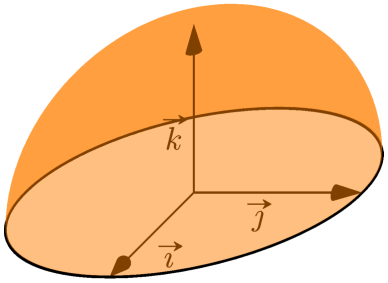

CODE 20

```
import three;
usepackage("esvect");
currentprojection=obliqueX;
size(5cm);
// En attendant d'avoir les axes
// avec graph3.asy
draw(O--X, red+linewidth(0.7pt), Arrow3);
draw(O--Y, red+linewidth(0.7pt), Arrow3);
draw(O--Z, red+linewidth(0.7pt), Arrow3);
label("$\vv{\imath}$", 0.5*X, SE);
label("$\vv{\jmath}$", 0.5*Y, SW);
label("$\vv{k}$", 0.5*Z, SW);
draw(unitcone,blue+opacity(0.5));
draw(unitcylinder,green+opacity(0.5));
draw(unitcircle3);
```



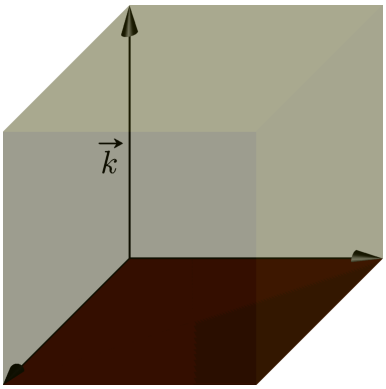

CODE 21

```
import three;
usepackage("esvect");
currentprojection=obliqueX;
currentlight=nolight;
size(5cm);
// En attendant d'avoir les axes
// avec graph3.asy
draw(O--X, black+linewidth(0.7pt), Arrow3);
draw(O--Y, black+linewidth(0.7pt), Arrow3);
draw(O--Z, black+linewidth(0.7pt), Arrow3);
label("$\vv{\imath}$", 0.5*X, SE);
label("$\vv{\jmath}$", 0.5*Y, SW);
label("$\vv{k}$", 0.5*Z, SW);
draw(unitsphere,orange+opacity(0.5));
draw(unitcircle3,black+linewidth(1pt));
```




CODE 22

```
import three;
usepackage("esvect");
currentprojection=obliqueX;
currentlight=nolight;
size(5cm);
// En attendant d'avoir les axes
// avec graph3.asy
draw(O--X, black+linewidth(0.7pt), Arrow3);
draw(O--Y, black+linewidth(0.7pt), Arrow3);
draw(O--Z, black+linewidth(0.7pt), Arrow3);
label("$\vv{\imath}$", 0.5*X, SE);
label("$\vv{\jmath}$", 0.5*Y, SW);
label("$\vv{k}$", 0.5*Z, SW);
draw(unithemisphere,orange+opacity(0.5));
draw(unitcircle3,black+linewidth(1pt));
```

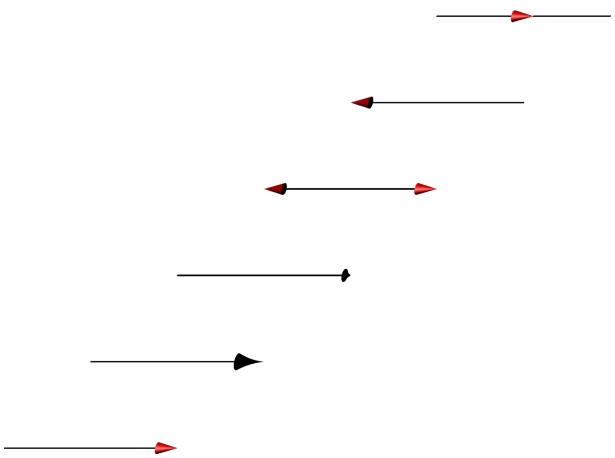



CODE 23

```
import three;
usepackage("esvect");
currentprojection=obliqueX;
size(5cm);
// En attendant d'avoir les axes
// avec graph3.asy
draw(O--X, black+linewidth(0.7pt), Arrow3);
draw(O--Y, black+linewidth(0.7pt), Arrow3);
draw(O--Z, black+linewidth(0.7pt), Arrow3);
label("$\vv{\imath}$", 0.5*X, SE);
label("$\vv{\jmath}$", 0.5*Y, SW);
label("$\vv{k}$", 0.5*Z, SW);

draw(unitplane,brown);
draw(unitcube,yellow+opacity(0.25));
```

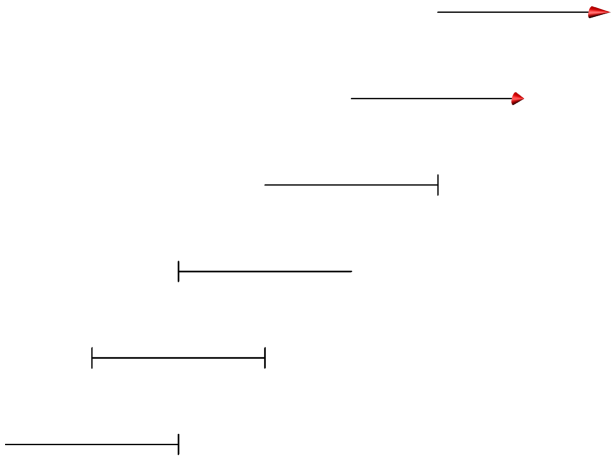
## 4. Flèches, barres

La flèche peut être placée au début, à la fin ou aux deux extrémités de la ligne avec trois « têtes » possibles : **DefaultHead**, **TeXHead** et **HookHead**.



CODE 24

```
import three;
currentprojection=obliqueX;
size(8cm);

draw((2,0,0)--(2,1,0),Arrow3(DefaultHead3, red));
draw((1,0,0)--(1,1,0),Arrow3(HookHead3));
draw((0,0,0)--(0,1,0),Arrow3(TeXHead3));
draw((-1,0,0)--(-1,1,0),Arrows3(red));
draw((-2,0,0)--(-2,1,0),BeginArrow3(red));
draw((-3,0,0)--(-3,1,0),MidArrow3(red));
```

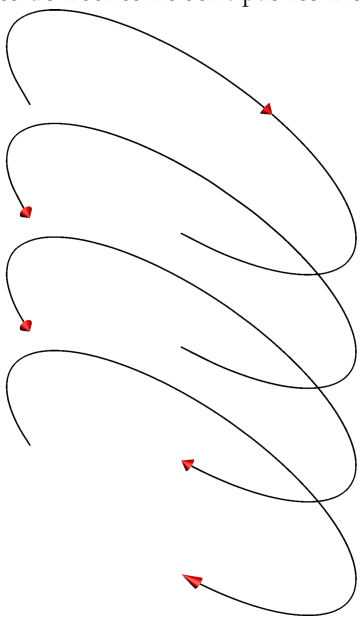


CODE 25

```
import three;
currentprojection=obliqueX;
size(8cm);

draw((2,0,0)--(2,1,0),Bar3);
draw((1,0,0)--(1,1,0),Bars3);
draw((0,0,0)--(0,1,0),BeginBar3);
draw((-1,0,0)--(-1,1,0),EndBar3);
draw((-2,0,0)--(-2,1,0),ArcArrow3(red));
draw((-3,0,0)--(-3,1,0),Arrow3(red));
```

**Remarques :**

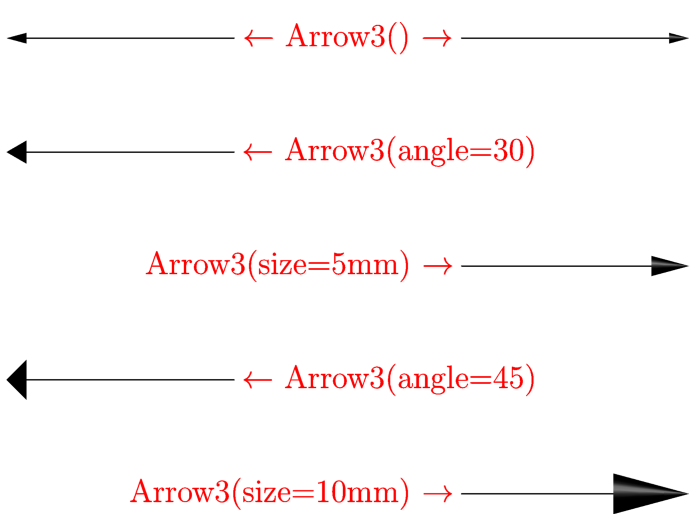
1. On observera que **ArcArrow3** n'est pas spécifique aux arcs et que **Arrow3** peut être utilisé avec un arc mais que les têtes de flèches ne sont pas les mêmes.



CODE 26

```
import three;
currentprojection=obliqueX;
size(8cm);

draw(arc(0,X-0.5*Y+0.75*Z,X+0.5*Y,CW), Arrow3(DefaultHead3, red));
draw(arc(0+0.75*Z,X-0.5*Y+1.5*Z,X+0.5*Y+0.75*Z,CW), ArcArrows3(red));
draw(arc(0+1.5*Z,X-0.5*Y+2.25*Z,X+0.5Y+1.5*Z,CW), BeginArcArrow3(red));
draw(arc(0+2.25*Z,X-0.5*Y+3*Z,X+0.5*Y+2.25*Z,CW), MidArcArrow3(red));
```

2. Comme en deux dimensions, on peut (re)définir la grosseur de la tête et l'angle :




CODE 27

```
import three;
currentprojection=orthographic(0,5,0);
unitsize(3cm);

draw(X--O, Arrow3());
draw(X-Z--O-Z, Arrow3(size=5mm));
draw(X-2*Z--O-2*Z, Arrow3(size=10mm));

draw(2*X--3*X, Arrow3());
draw(2*X-0.5*Z--3*X-0.5*Z, Arrow3(angle=30));
draw(2*X-1.5*Z--3*X-1.5*Z, Arrow3(angle=45));

label("$\leftarrow$ Arrow3() $\rightarrow$", (1.5,0,0), red);
label("Arrow3(size=5mm) $\rightarrow$", (1,0,-1),W, red);
label("Arrow3(size=10mm) $\rightarrow$", (1,0,-2),W, red);

label("$\leftarrow$ Arrow3(angle=30)", (2,0,-0.5), E, red);
label("$\leftarrow$ Arrow3(angle=45)", (2,0,-1.5), E, red);
```

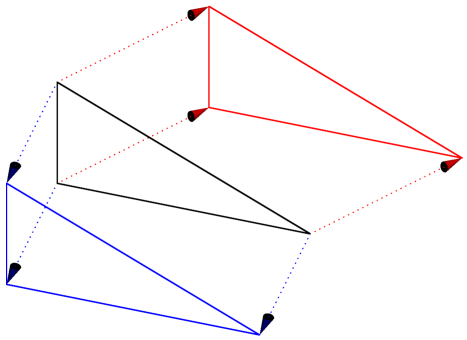
# III – Les transformations en trois dimensions

Les transformations du plan sont naturellement étendues à l'espace avec en prime les projections.
Elles sont regroupées sous le type de **transform3**.

## 1 . les translations

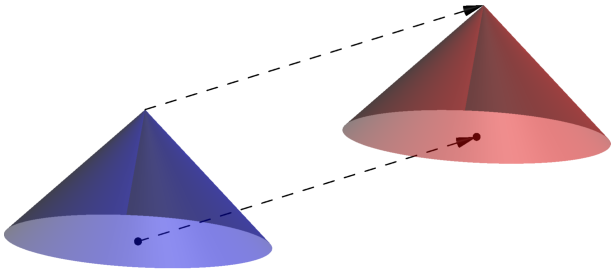
`transform3 shift(triple v)`

définit la translation de vecteur **v**.



CODE 28

```
import three;
currentprojection=obliqueX;
size(6cm);

triple u=(-1.5,0.75,0),
       v=(0,-0.5,-1);
transform3 t1=shift(u);
transform3 t2=shift(v);

triple pA=(0,0,0),
       pB=(0,0,1),
       pC=(1,3,0);
path3 g=pA--pB--pC--cycle,
      g1=t1*g,
      g2=t2*g;

draw(g);
draw(g1,red);
draw(g2,blue);
draw(pA--t1*pA,red+dotted,Arrow3);
draw(pB--t1*pB,red+dotted,Arrow3);
draw(pC--t1*pC,red+dotted,Arrow3);
draw(pA--t2*pA,blue+dotted,Arrow3);
draw(pB--t2*pB,blue+dotted,Arrow3);
draw(pC--t2*pC,blue+dotted,Arrow3);
```



CODE 29

```
import three;
currentprojection=orthographic(
        camera=(0.0252492180883091,
                0.00368201499815729,
                0.00500319427129251),
        up=(0.00668806224327891,
            -0.000569778159228265,
            0.0300934299745645),
        target=(0,0,0),
        zoom=1);
size(8cm);

triple pN=(0,0,1),
       pS=(0,0,0);

triple u=(0,2.5,1);
transform3 t=shift(u);

draw(unitcone,blue+opacity(0.5));
draw(t*unitcone,red+opacity(0.5));
draw(pN--t*pN,dashed,Arrow3);
draw(pS--t*pS,dashed,Arrow3);
dot(pS);
dot(t*pS);
```

## 2 . Agrandissement — Réduction

Les agrandissements/réductions sont définis dans chacune des directions des axes de coordonnées :

`transform3 xscale3(real x)`

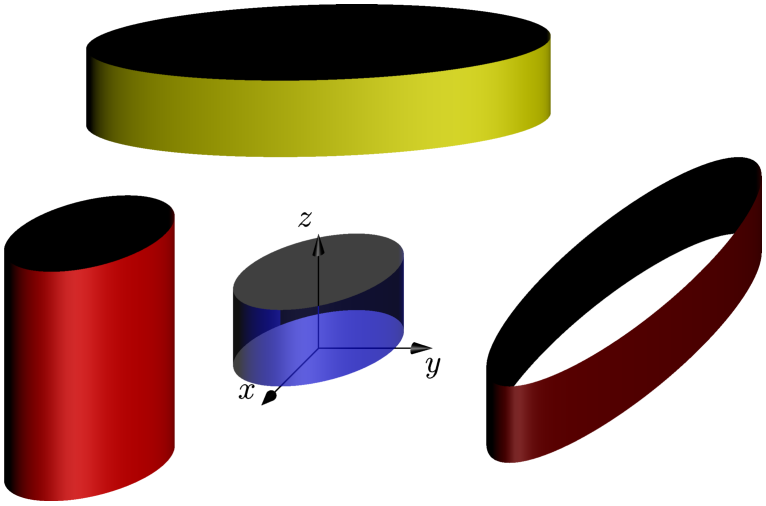
multiplie par **x** dans la direction $(Ox)$.

`transform3 yscale3(real y)`

multiplie par **y** dans la direction $(Oy)$.

`transform3 zscale3(real z)`

multiplie par **z** dans la direction $(Oz)$.




CODE 30

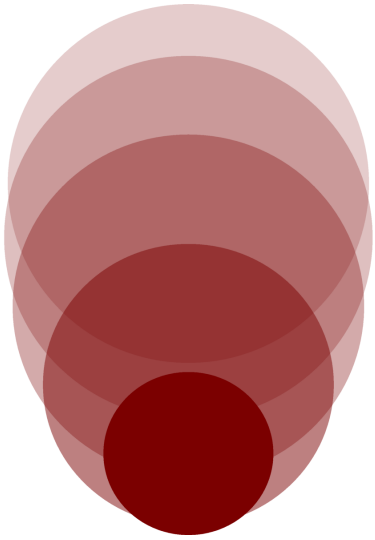
```
import three;
currentprojection=obliqueX;
size(10cm);

draw(O--1.5*X,Arrow3);   // En attendant
draw(O--1.5*Y,Arrow3);   // d'avoir les axes
draw(O--1.5*Z,Arrow3);   // avec graph3.asy
label("$x$", 1.5*X, NW);
label("$y$", 1.5*Y, S);
label("$z$", 1.5*Z, NW);

real k=3;
draw(unitcylinder,blue+opacity(0.75));
draw(shift(3,-1.5,0)*zscale3(k)*unitcylinder,red);
draw(shift(0,0,3)*yscale3(k)*unitcylinder,yellow);
draw(shift(0,4,0)*xscale3(k)*unitcylinder,brown);
```

`transform3 scale3(real s)`

multiplie par **s** dans toutes les directions.

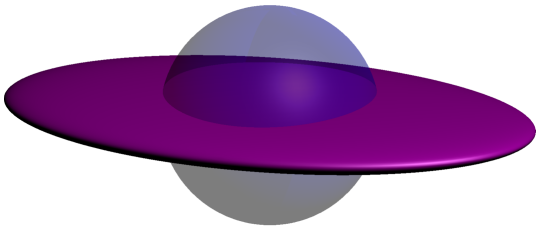
`transform3 scale(real x, y, z)`

multiplie par **x** dans la direction $(Ox)$, par **y** dans la direction $(Oy)$ et par **z** dans la direction $(Oz)$.



CODE 31

```
import three;
currentprojection=perspective(1,1,5);
currentlight=(1,1,5);
size(7cm);

for(real s=1; s<=5; ++s)
        {
        transform3 ag=scale3(s);
        draw(ag*shift((0,0,s))*unitdisk,
             brown+opacity(1/s));
}
```



CODE 32

```
import three;
currentprojection=orthographic(3,1,1);
size(7cm);

transform3 ag=scale(1.5,2.5,0.25);

draw(unitsphere,blue+opacity(0.3));
draw(ag*unitsphere,heavymagenta);
```

### 3 . Rotation

Deux manières de définir une rotation :

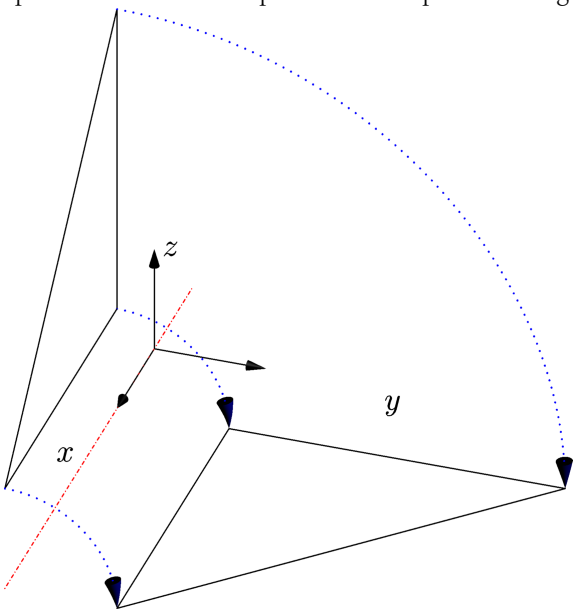
`transform3 rotate(real angle, triple v)`

définit la rotation d'angle **angle** et d'axe (**O**, **v**).

`transform3 rotate(real angle, triple u, triple v)`

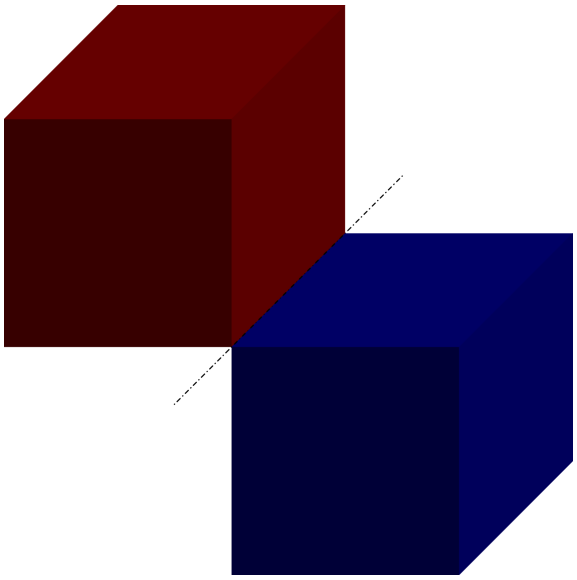
définit la rotation d'angle **angle** et d'axe la droite **u**−−**v**.

Vous pourrez retrouver le premier exemple dans la gallerie de Gaétan Marris [5].




CODE 33

```
import three;
size(7.5cm,0);

currentprojection=orthographic(3,1,2);

triple vectaxe=(1,0,0);
transform3 r=rotate(-90,vectaxe);
triple pA=(1,0,1), pB=(4,0,1), pC=(1,0,4);
path3 tri=pA--pB--pC--cycle;
path3 trip=r*tri;

draw(tri^^trip);

draw(O--X,Arrow3);   // En attendant
draw(O--Y,Arrow3);   // d'avoir les axes
draw(O--Z,Arrow3);   // avec graph3.asy
label("$x$", 2*X, NW);
label("$y$", 2*Y, SE);
label("$z$", Z, E);

pen dotteddash=linetype("0 4 4 4"),
    p2=.8bp+blue+dotted;
draw((-1,0,0)--(4,0,0),red+dotteddash);
draw(arc((pA.x,0,0),pA,r*pA,CCW),p2,Arrow3);
draw(arc((pB.x,0,0),pB,r*pB,CCW),p2,Arrow3);
draw(arc((pC.x,0,0),pC,r*pC,CCW),p2,Arrow3);
```



CODE 34

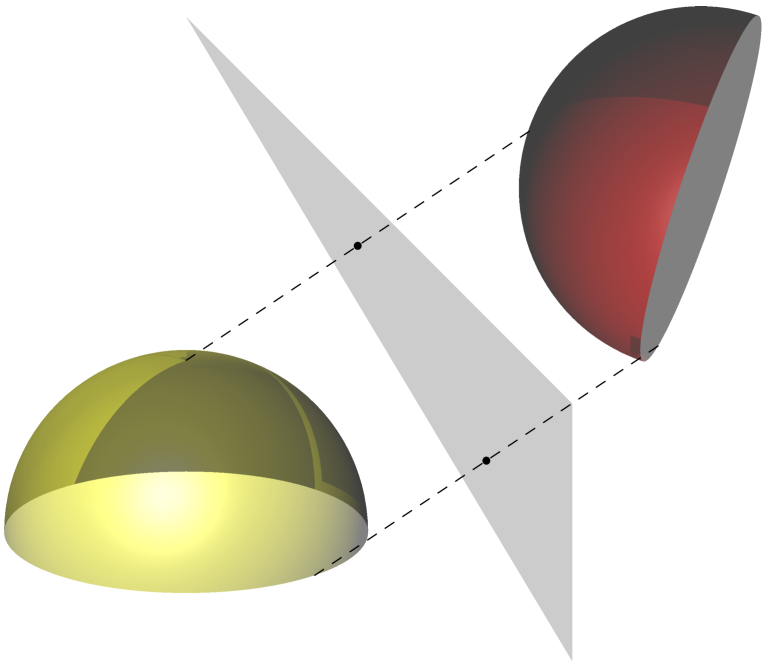
```
import three;
size(7.5cm,0);

currentprojection=obliqueX;

triple point1=(0,1,0),
       point2=(1,1,0);
transform3 r=rotate(180,point1,point2);

draw(unitcube,red);
draw(r*unitcube,blue);

//draw(shift(0,0,-1)*unitcube,
//                green+opacity(0.7));
//draw(r*shift(0,0,-1)*unitcube,
//                yellow+opacity(0.7));

pen dotteddash=linetype("0 4 4 4");
draw((-0.5,1,0)--(1.5,1,0),dotteddash);
```

## 4. Réflexion

La réflexion par rapport à un plan est quand à elle définie par la routine :

`transform3 reflect(triple u, triple v, triple w)`

où le plan de réflexion est défini par les points **u**, **v** et **w**.



CODE 35

```
import three;
currentprojection=orthographic(4,4,2);
currentlight=(4,2,4);
size(10cm);

triple pA=(-3,0,0), pB=(0,3,0), pC=(0,0,3);
transform3 sym=reflect(pA,pB,pC);
path3 tr=pA--pB--pC--cycle;

draw(surface(tr),opacity(0.2));
draw(unithemisphere,yellow+opacity(0.5));
draw(sym*unithemisphere,red+opacity(0.5));

triple pS=(0,0,1), pE=(0,1,0);
draw(pS--sym*pS,dashed);
draw(pE--sym*pE,dashed);
dot((pS+sym*pS)/2);
dot((pE+sym*pE)/2);
```

## 5 . Projections

Pour projeter un chemin sur un plan , on utilise la transformation :

`transform3 planeproject(triple n, triple O=O, triple dir=n);`

qui projette le chemin **p** dans le plan passant par **O** et de vecteur normal **n**.

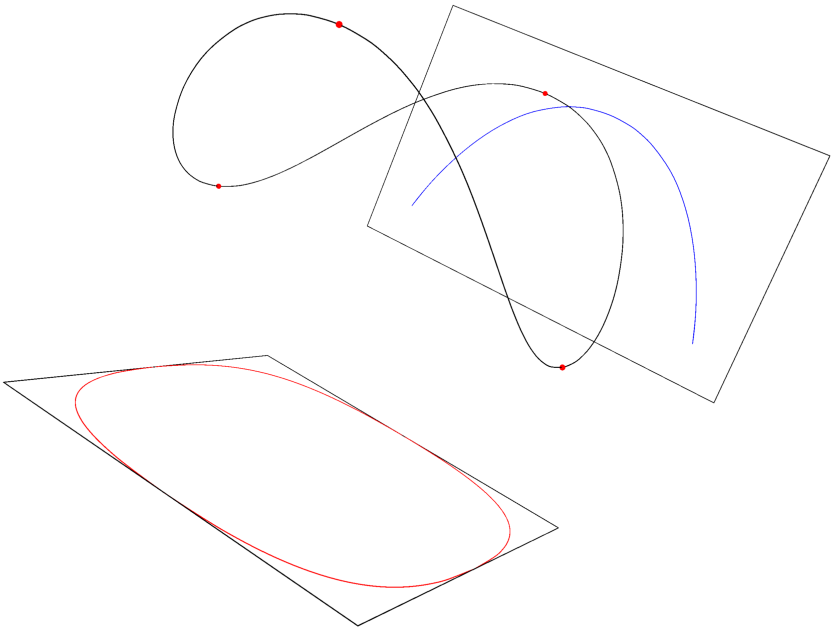
On peut utiliser la routine :

`transform3 planeproject(path3 p, triple dir=normal(p));`

qui projette dans la direction **dir** sur le plan contenant le chemin (plan) **p**.
où :

`triple normal(path3 p);`

détermine le vecteur normal unitaire au plan contenant le chemin (plan) **p**.



CODE 36

```
import three;
//currentprojection=perspective(5,4,2);
currentprojection=perspective(
        camera=(2.60170102889344,-4.91463292460365,2.17489894144093),
        up=(-0.00711675782582678,0.000286046483850576,0.0152098185742004),
        target=(-0.00382168498253588,0.540537739885363,0.853170375594943),
        zoom=0.676839362028687,
        angle=32.1378964699335,
        autoadjust=false);
size(12.5cm);

path3 g=(1,0,1)..(0,1,2)..(-1,0,1)..(0,-1,2)..cycle;
draw(g);
draw(((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle));
dot(g,red);

transform3 pr1=planeproject(Z,0);
draw(pr1*g,red);

path3 h=(1.25,2,0.75)--(-1.25,2,0.75)--(-1.25,2,2.25)--(1.25,2,2.25)--cycle;
draw(h);
triple n=normal(h);
transform3 pr2=planeproject(h,n);
draw(pr2*g,blue);
```

## 6 . De deux à trois dimensions et vice versa ...

Les routines suivantes permettent de faire le lien entre certains éléments définis en deux dimensions et l'espace.

### 6. 1 . les commandes **invert** et **project**

Notions que j'ai eu du mal à comprendre... Je vais essayer. Merci à Gaétan, que je cite presque dans le texte, d'avoir pris le temps de m'expliquer.

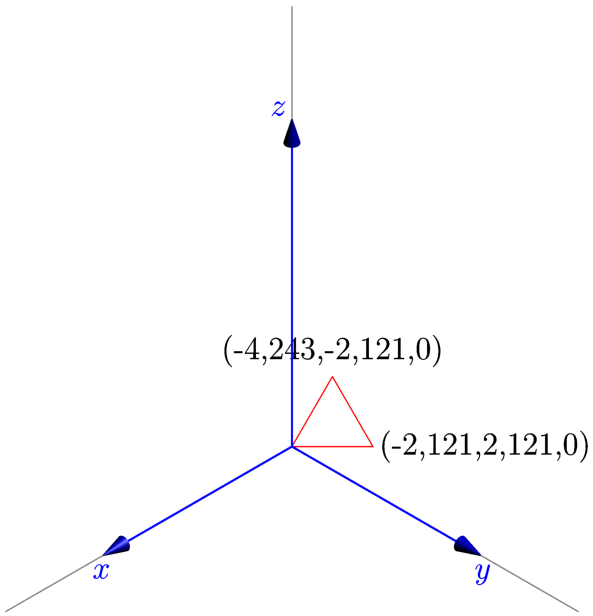
les commandes **invert** et **project** sont là pour faire le lien, entre les coordonnées **triples** d'un point dans l'espace et les coordonnées **doubles** de la représentation de ce **même point** sur une feuille ou sur l'écran de votre ordinateur.
Elles permettent de faire un aller-retour entre des coordonnées différentes d'**un seul et unique point**, respectivement dans un repère 3D et un repère 2D plaqué (sur la projection qui représente la situation 3D).

```
triple invert(pair z, triple normal, triple point, projection P=currentprojection);
```

Cette routine envoie le **pair z** sur le plan orthogonal à **normal** passant par **point**.

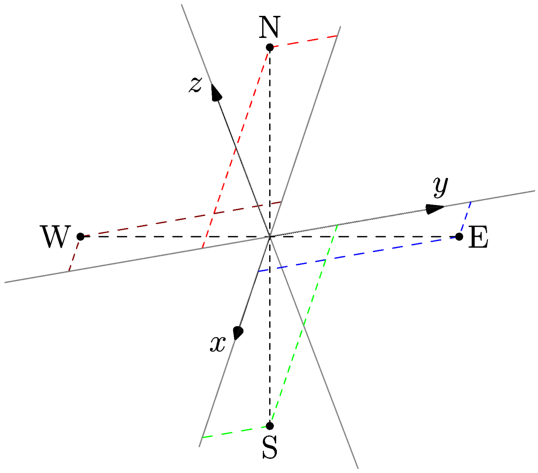
**Remarque :**
**Attention** : il est question ici de projection... mais pas de celles qui transforment un point de l'espace en un autre point de l'espace que l'on a étudié au paragraphe précédent.

Elle peut servir à des problèmes spécifiques.
Par exemple, dessiner un triangle dans le plan $(xOy)$ qui donne l'impression d'être équilatéral :




CODE 37

```
import graph3;
size(8cm);
currentprojection=orthographic(5,5,5);

triple pO=(0,0,0),
pA=invert((0,0),Z,pO),
pB=invert((3,0),Z,pO),
pC=invert(3*dir(60),Z,pO);

draw(pA--pB--pC--cycle,red);

void affichercoordonnees(triple pM,align d) {
label(format("(%.3f,",pM.x)
        +format("%.3f,",pM.y)
        +format("%.3f)",pM.z),
      pM,d);
}
affichercoordonnees(pB,E);
affichercoordonnees(pC,N);

limits(O,15X+15Y+15Z);
xaxis3(gray);
yaxis3(gray);
zaxis3(gray,zmax=20);
limits(O,10X+10Y+10Z);
xaxis3(Label("$x$",1),.8bp+blue,Arrow3());
yaxis3(Label("$y$",1),.8bp+blue,Arrow3());
zaxis3(Label("$z$",1),.8bp+blue,Arrow3());
```




CODE 38

```
import geometry;
import graph3;
currentprojection=orthographic((5,2,3),up=Y+3*Z);

size(7cm);

triple pN=invert((0,1),Z,0),
pE=invert((1,0),Z,0),
pW=invert((-1,0),Z,0),
pS=invert((0,-1),Z,0);

limits((-2,-1.5,-1.5),(2,1.5,1.5));
xaxis3(gray);
yaxis3(gray);
zaxis3(gray,zmin=-1.5,zmax=1.5);
limits((0,0,0),(1,1,1));
xaxis3("$x$",Arrow3(),above=true);
yaxis3(Label("$y$",Z),Arrow3(),above=true);
zaxis3(Label("$z$"),zmin=0,zmax=1,Arrow3(),above=true);

dot("N",pN,N);
dot("E",pE,E);
dot("S",pS,S);
dot("W",pW,W);
draw(pN--pS^^pE--pW,dashed);

void montrecoord(triple P, pen stylo){
draw((P.x,0,0)--P--(0,P.y,0),stylo);
}
montrecoord(pN,red+dashed);
montrecoord(pE,blue+dashed);
montrecoord(pS,green+dashed);
montrecoord(pW,brown+dashed);
```

Les coordonnées (en 3D) des points du 1er exemple n'ont rien à voir avec celle de départ (en 2D).
Sur le 2e exemple, on voit bien que les coordonnées en deux dimensions des points **N**, **W**, **S**, **E** sont bien $(0,1)$, $(-1,0)$,

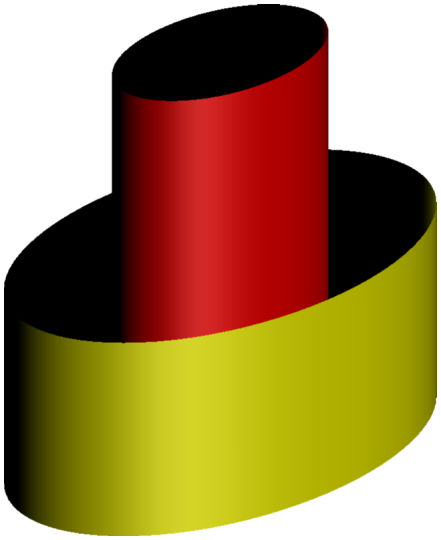
$(-1,-1)$ et $(1,0)$ dans le repère usuel.
Je vous laisse deviner qui est l'auteur de ces exemples.

### 6. 2 . Transformation **extrude**

Les routines

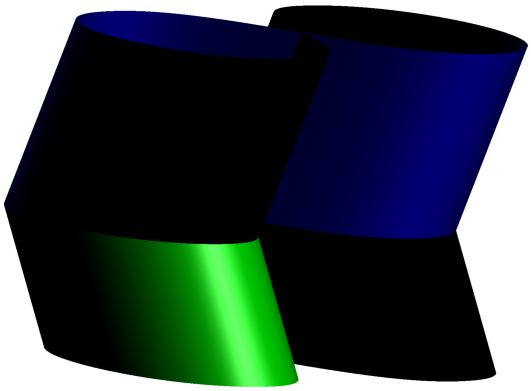
```
surface extrude(path p, triple axis=Z);
surface extrude(Label L, triple axis=Z);
```

dessinent la surface obtenue en faisant « glisser » le chemin **p** ou le label **L** dans la direction du vecteur **axis**.



CODE 39

```
import three;
currentprojection=obliqueX;
size(7cm);

path g=(1,0)..(0,1)..(-1,0)..(0,-1)..cycle;
surface s=extrude(g);
draw(s,yellow);

path g2=scale(0.5)*g;
surface s2=extrude(g2,2*Z);
draw(s2,red);
```



CODE 40

```
import three;
currentprojection=orthographic(
        camera=(0.0131146571808127,
                -0.0133784021031586,
                0.00191475706974314),
        up=(0.00256978814347831,
            -0.00258164193231038,
            0.0251475955218861),
        target=(0,0,0),
        zoom=1);
size(7cm,5cm,IgnoreAspect);

path g=(1,1)..(-1,1)..(1,-1)..(-1,-1)..cycle;
surface s=extrude(g,0.35*(X+Y+3*Z));
surface s2=extrude(g,0.25*(X+Y-3*Z));

draw(s,blue);
draw(s2,green);
```


CODE 41

```
import three;
currentprojection=orthographic(-2,-2,3);
size(8cm);

Label L=Label("\textsc{Asymptote}",(0,0));
Label M=Label("\small et ses potes", (0,0));
surface s=extrude(L,3*Z);
surface t=shift((0,-5,-4))*rotate(45,X)*extrude(M,Z);
draw(s,blue);
draw(t,red);
```

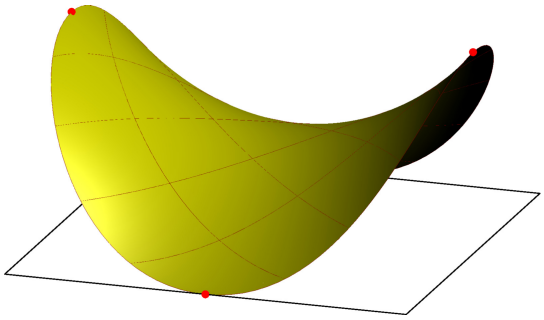
# IV – Compléments sur **draw** et les **path3**

## 1 . les routines **draw**

Pour dessiner une surface, on peut utiliser une des trois routines :

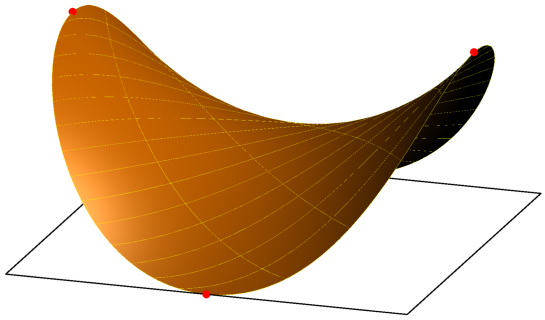
```
void draw(picture pic=currentpicture, surface s, int nu=1, int nv=1,
          material surfacepen=currentpen, pen meshpen=nullpen,
          light light=currentlight, light meshlight=light, string name="",
          render render=defaultrender);
void draw(picture pic=currentpicture, surface s, int nu=1, int nv=1,
          material[] surfacepen, pen meshpen,
          light light=currentlight, light meshlight=light, string name="",
          render render=defaultrender);
void draw(picture pic=currentpicture, surface s, int nu=1, int nv=1,
          material[] surfacepen, pen[] meshpen=nullpens,
          light light=currentlight, light meshlight=light, string name="",
          render render=defaultrender);
```

Les paramètres **nu** et **nv** définissent le nombres de subdivisions dans chaque direction sur la surface pour chaque courbe de Bézier tracée pour éventuellement tracer les **mesh**.



CODE 42

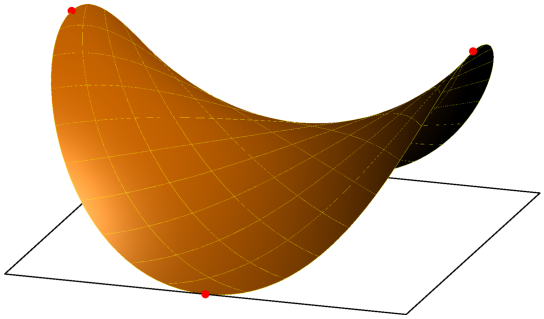
```
import three;
currentprojection=orthographic(3,1,1);
size(200,0);

path3 g=(1,0,0)..(0,1,1)..(-1,0,0)..(0,-1,1)..cycle;
draw(surface(g),nu=5, nv=5, yellow,brown);
draw((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle);
dot(g,red);
```



CODE 43

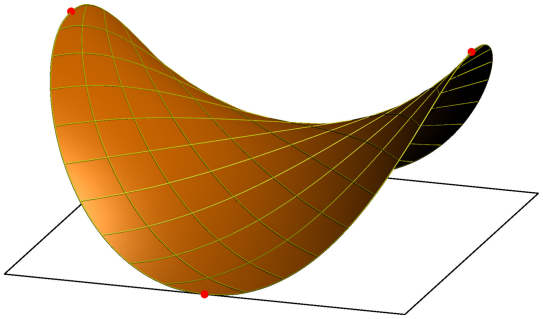
```
import three;
currentprojection=orthographic(3,1,1);
size(200,0);

path3 g=(1,0,0)..(0,1,1)..(-1,0,0)..(0,-1,1)..cycle;
draw(surface(g),nu=5, nv=15, orange,yellow);
draw((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle);
dot(g,red);
```

On pourra s'attarder sur les pinceaux **thin**() et **thick**() pour **meshpen** :



CODE 44

```
import three;
currentprojection=orthographic(3,1,1);
size(200,0);

path3 g=(1,0,0)..(0,1,1)..(-1,0,0)..(0,-1,1)..cycle;
draw(surface(g),nu=10, nv=10, orange,yellow+thin());
draw((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle);
dot(g,red);
```

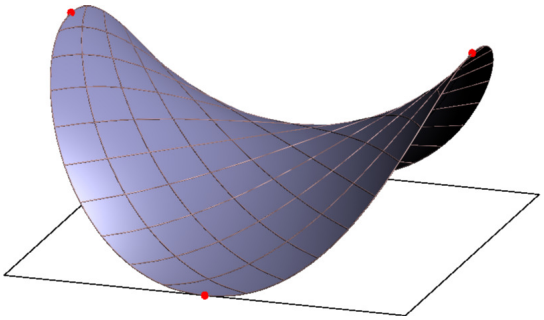


CODE 45

```
import three;
currentprojection=orthographic(3,1,1);
size(200,0);

path3 g=(1,0,0)..(0,1,1)..(-1,0,0)..(0,-1,1)..cycle;
draw(surface(g),nu=10, nv=10, orange,yellow+thick());
draw((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle);
dot(g,red);
```

Le paramètre **thin**() est toujours employé par défaut pour dessiner les **mesh** et les contours. Pour plus d'informations sur les paramètres possibles de ces deux routines, il faudra aller fouiller dans le fichier plain_pens.asy où elles sont définies.

Si l'on choisit la troisième routine de **draw**, on a la d'utiliser plusieurs couleurs pour tracer les **mesh** :



CODE 46

```
import three;
currentprojection=orthographic(3,1,1);
size(200,0);

pen[] p1={paleblue,darkblue},
      p2={palered, red, brown, darkbrown};

path3 g=(1,0,0)..(0,1,1)..(-1,0,0)..(0,-1,1)..cycle;
draw(surface(g),nu=10, nv=10, p1,p2+thick());
draw((-1,-1,0)--(1,-1,0)--(1,1,0)--(-1,1,0)--cycle);
dot(g,red);
```

En combinant l'utilisation de cette routine avec l'emploie du module **pallette.asy**, on peut obtenir des dégradés de couleurs. On se reportera au paragraphe consacré à **pallette.asy**.

On ne s'étendra pas sur **material** :

```
struct material {
                pen[] p; // diffusepen,ambientpen,emissivepen,specularpen
                real opacity;
                real shininess;
                ...
}
```

celui qui voudra approfondir devra fouiller le fichier three_light.asy.

## 2. Autour des **path3**

Les fonctions définies en 2D pour les **path** ont, pour la plupart, une versions 3D. Voici une revue (non-exhaustive) d'effectif :

- `int length(path3 p);`

  retourne le nombre de segment (linéaire ou cubique) du chemin **p**.
- `int size(path3 p);`

  retourne le nombre de points (noeuds) du chemin **p**. Si **p** est cyclique, **size**(**p**) et **length**(**p**) sont identiques.
- `triple point(path3 p, int t);`
  `triple point(path3 p, real t);`

  retourne, dans le premier cas, les coordonnées du $t^e$ nœud modulo **length**(**p**) et, dans le deuxième cas, les coordonnées du point correspondant au paramètre du spline cubique $t - \mathbf{E}(t)$ où $\mathbf{E}(t)$ est la partie entière de t (ce point se situe entre les nœuds $\mathbf{E}(t)$ et $\mathbf{E}(t)+1$).
- `triple dir(path3 p, int t, int sign=0, bool normalize=true);`
  `triple dir(path3 p, real t, bool normalize=true);`

  retournent la direction du vecteur tangent au point t comme expliqué précédemment. Si le paramètre **sign**>0, il s'agit de la direction du vecteur « entrant » si **sign**>0, il s'agit de la direction du vecteur « sortant » et la moyenne des deux si **sign**=0.
  Á noter que **dir**(**path3 p**) est équivalant à **dir**(**p**, **length**(**p**)).
- `triple accel(path3 p, int t, int sign=0);`
  `triple accel(path3 p, real t);`

  travaillent comme les deux fonctions précédentes mais retournent l'accélération.
- `real radius(path3 p, real t);`

  retourne la courbure du chemin **p** au point t.
- `triple precontrol(path3 p, int t);`
  `triple precontrol(path3 p, real t);`

  retourne les coordonnées du point de pré-controle de **p** au nœud t (respectivement au point t).
- `triple postcontrol(path3 p, int t);`
  `triple postcontrol(path3 p, real t);`

  retourne les coordonnées du point de post-controle de **p** au nœud t (respectivement au point t).
- `real arclength(path3 p);`

  retourne la longueur du chemin **p** (en unités de l'utilisateur).
- `real arctime(path3 p, real L);`

  retourne un nombre réel compris entre 0 et la longueur du chemin **p** au sens **point**(**path3 p**, **real** t) tel que la longueur d'arc depuis l'origine de **p** et le point acrtime soit égale à **L**.
- `path reverse(path3 p);`

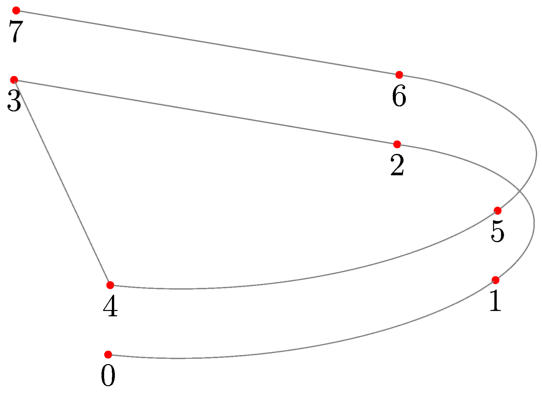
  retourne le chemin **p** mais parcouru dans l'autre sens.
- `path subpath(path3 p, int a, int b);`
  `path subpath(path3 p, real a, real b);`

  retournent le sous-chemin de **p** compris entre les nœuds a et b (respectivement le point a et le point b).
- `triple min(path3 p);`
  `triple max(path3 p);`

  retournent les **triple** (**xmin**,**ymin**,**zmin**) et (**xmax**,**ymax**,**zmax**) correspondants à la boîte nécessaire pour englober le chemin **p**.
- `bool cyclic(path3 p);`

  est un booléen qui retourne **true** si le chemin **p** est cyclique.
- `bool straight(path3 p, int i);`

  est un booléen qui retourne **true** si la partie du chemin **p** comprise entre les nœuds i et i+1 est un segment de droite.

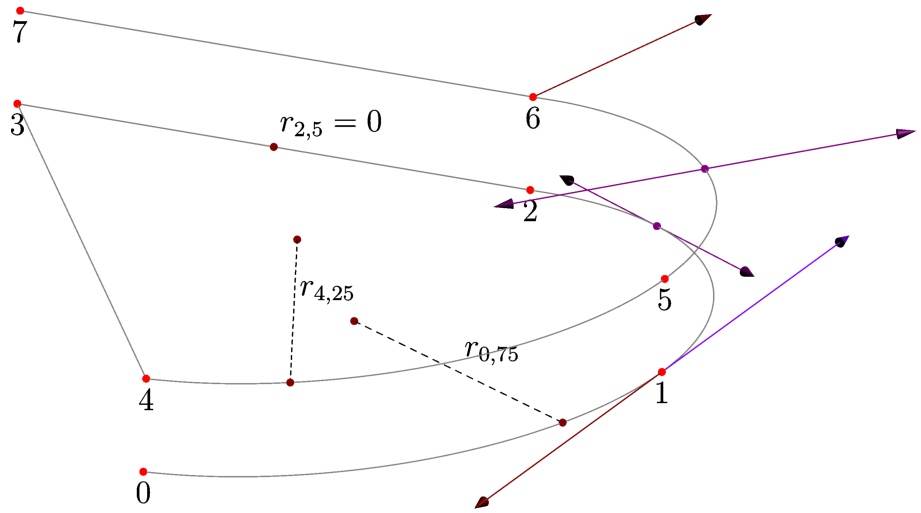



lenght(p)= 7
size(p)= 8
arclength(p)= 10,73
arctime(p,5)= 3,26

chemin non cyclique
p[2]–p[3] est un segment
p[5]–p[6] n'est pas un segment

CODE 47

```
import three;
currentprojection=orthographic(
        camera=(0.00723065282059533,0.00419049670835119,0.00139125395474534),
        up=(0.00223758785379906,0.00101840335331956,0.0083674288407205),
        target=(0,0,0),
        zoom=1);
size(12cm);

triple[] z;
         z[0]=(1,0,0);
         z[1]=(0,1,0.25);
         z[2]=(-1,0,0.5);
         z[3]=(0,-1,0.75);

for(int n=4; n <= 7; ++n) z[n]=z[n-4]+0.25*Z;

path3 p=z[0]..z[1]..z[2]--z[3]--z[4]..z[5]..z[6]--z[7];

draw(p,grey);
dot(z,red);

for(int n=0; n<=7; ++n) label(format("$%i$",n),z[n],S);

real h=-0.125;
label(format("lenght(p)= %i",length(p)), (1,-1.25,2*h),E);
label(format("size(p)= %i",size(p)), (1,-1.25,3*h),E);
label(format("arclength(p)= %.2f",arclength(p)), (1,-1.25,4*h),E);
label(format("arctime(p,5)= %.2f",arctime(p,5)), (1,-1.25,5*h),E);

if(cyclic(p)==true)
          label("chemin cyclique",(1,0.75,h),E);
        else
          label("chemin non cyclique",(1,0.75,h),E);

void segment(path3 p, int i, int n) {
        if(straight(p,i)==true)
          label(format("p[%i]",i)+format("--p[%i] est un segment",i+1),(1,0.75,n*h),E);
        else
          label(format("p[%i]",i)+format("--p[%i] n'est pas un segment",i+1),(1,0.75,n*h),E);
}
segment(p,2,2);
segment(p,5,3);
```




CODE 48

```
import three;
currentprojection=orthographic(
        camera=(0.00723065282059533,0.00419049670835119,0.00139125395474534),
        up=(0.00223758785379906,0.00101840335331956,0.0083674288407205),
        target=(0,0,0),
        zoom=1);
size(12cm);

triple[] z;
         z[0]=(1,0,0);
         z[1]=(0,1,0.25);
         z[2]=(-1,0,0.5);
         z[3]=(0,-1,0.75);

for(int n=4; n <= 7; ++n) z[n]=z[n-4]+0.25*Z;

path3 p=z[0]..z[1]..z[2]--z[3]--z[4]..z[5]..z[6]--z[7];

draw(p,grey);
dot(z,red);
for(int n=0; n<=7; ++n) label(format("$%i$",n),z[n],S);

triple v1=dir(p,1,1),
       v2=dir(p,1,-1),
       v3=dir(p,1.7),
       p3=point(p,1.7);

draw(z[1]--z[1]+v1,purple,Arrow3);
draw(z[1]--z[1]-v2,brown,Arrow3);
dot(p3,deepmagenta);
draw(p3-0.5*v3--p3+0.5*v3,deepmagenta,Arrows3);

triple v4=accel(p,6,-1),
       v5=accel(p,5.5),
       p5=point(p,5.5);

draw(z[6]--z[6]-0.25*v4,brown,Arrow3);
dot(p5,deepmagenta);
draw(p5-0.25*v5--p5+0.25*v5,deepmagenta,Arrows3);
```

```
void rayon(path3 p, real t)
        {
        if(straight(p,floor(t))==false) {
        real r=radius(p,t);
        triple M=point(p,t),
               v=dir(p,t),
               a=accel(p,t)/length(accel(p,t)),
               c=M+r*a;
        path3 g=c--M;
        draw(g,dashed);
        dot(g,brown);
        label(format("$r_{%f}$",t),(M+c)/2,NE);
        }
        else {
        dot(point(p,t),brown);
        label(format("$r_{%f}=0$",t),point(p,t),NE);
        }
}

rayon(p,4.25);
rayon(p,0.75);
rayon(p,2.5);
```

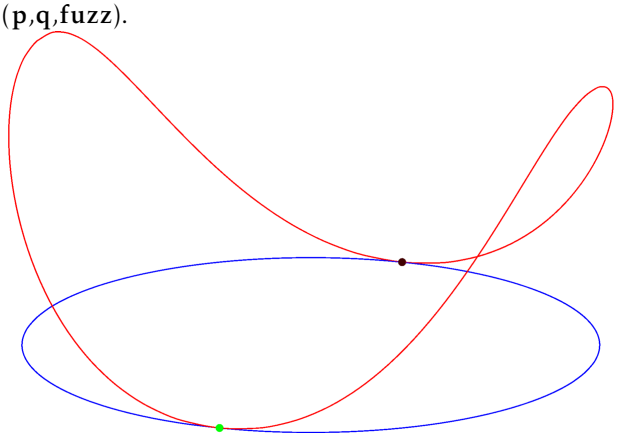
Il nous reste à étudier les fonctions relatives à l'intersection de deux chemins ou d'un chemin et d'un surface :

- `real[] intersect(path3 p, path3 q, real fuzz=-1);`

  Si les chemin **p** et **q** ont au moins un point d'intersection, cette routine retourne une liste de réels de longueur 2 représentant la situation du point d'intersection, au sens **point**(**path3**, **real**), sur chacun des chemins **p** et **q**. Si **p** et **q** ne se coupent pas, la routine renvoie une liste de longueur 0.
  Le paramètre **fuzz** permet de définir l'erreur absolue lors des calculs. Une valeur négative renvoie à la précision de votre ordinateur.

  `triple intersectionpoint(path3 p, path3 q, real fuzz=-1);`

  est équivalant à **point**(**p**, intersec(**p**,**q**,**fuzz**).



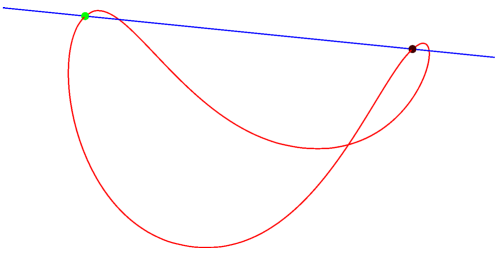
CODE 49

```
import three;
import graph3;
size(8cm,0);
currentprojection=orthographic(3,1,1);

path3 ch1=(0,-1,1)..(1,0,0)..(0,1,1)..(-1,0,0)..cycle;
path3 ch2=unitcircle3;
draw(ch1,red);
draw(ch2,blue);

real[] inter=intersect(ch1,ch2);
triple p1=point(ch1,inter[0]);
dot(p1, green);
triple p2=intersectionpoint(reverse(ch1),ch2);
dot(p2, darkbrown);
```

- D'autres routines retournent la totalité des points d'intersection sous forme de listes ou de tableau :

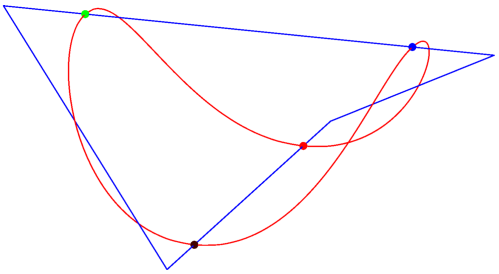
```
real[][] intersections(path3 p, path3 q, real fuzz=-1);
triple[] intersectionpoints(path3 p, path3 q, real fuzz=-1);
```

marchent comme **intersect** et **intersectionpoint** mais retournent tous les points d'intersection entre **p** et **q**.



CODE 50

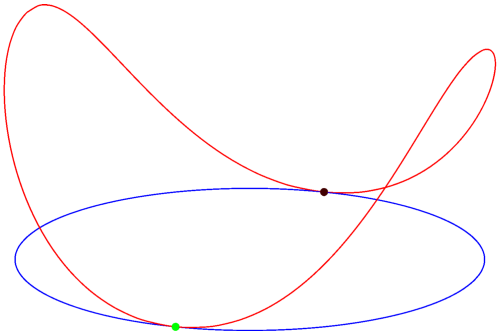
```
import three;
size(6.5cm,0);
currentprojection=orthographic(3,1,1);

path3 ch1=(0,-1,1)..(1,0,0)..
          (0,1,1)..(-1,0,0)..
          cycle;
path3 ch2=(1,-1,0)--(1,1,0)--
          (-1,1,0)--(-1,-1,0)--
          cycle;

draw(ch1,red);
draw(ch2,blue);

real[][] inter=intersections(ch1,ch2);
triple p1=point(ch1,inter[0][0]);
triple p2=point(ch2,inter[1][1]);
dot(p1, green);
dot(p2, darkbrown);

write(sort(inter));
```



CODE 51

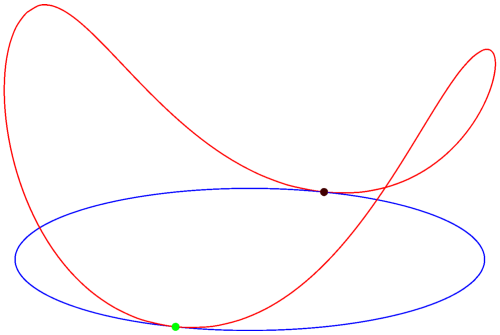
```
import three;
size(6.5cm,0);
currentprojection=orthographic(3,1,1);

path3 ch1=(0,-1,1)..(1,0,0)..
          (0,1,1)..(-1,0,0)..
          cycle;
path3 ch2=(0,1.5,1)--(0,-1.5,1)..
          (1.5,0,0)--(-1.5,0,0)..
          cycle;

draw(ch1,red);
draw(ch2,blue);

real[][] inter=intersections(ch1,ch2);
triple p1=point(ch1,inter[0][0]);
triple p2=point(ch2,inter[1][1]);
triple p3=point(ch1,inter[2][0]);
triple p4=point(ch2,inter[3][1]);
dot(p1, green);
dot(p2, darkbrown);
dot(p3, blue);
dot(p4, red);
write(sort(inter));
```

Si les racines sont « multiples » comme dans notre premier exemple où les chemins sont tangents aux points d'intersection, les dimensions de la matrice seront multipliées. Ici, la racine est double, ma matrice sera donc de dimension $4 \times 2$, les deux premières lignes représentant la première racine double et les deux suivantes la deuxième :



CODE 52

```
import three;
size(6.5cm,0);
currentprojection=orthographic(3,1,1);

path3 ch1=(0,-1,1)..(1,0,0)..(0,1,1)
                ..(-1,0,0)..cycle;
path3 ch2=unitcircle3;
draw(ch1,red);
draw(ch2,blue);

real[][] inter=intersections(ch1,ch2);
triple p1=point(ch1,inter[0][0]);
triple p2=point(ch2,inter[2][1]);
dot(p1, green);
dot(p2, darkbrown);
```



CODE 53

```
import three;
size(6.5cm,0);
currentprojection=orthographic(3,1,1);

path3 ch1=(0,-1,1)..(1,0,0)..(0,1,1)
                ..(-1,0,0)..cycle;
path3 ch2=unitcircle3;
draw(ch1,red);
draw(ch2,blue);

triple[] inter=intersectionpoints(ch1,ch2);
triple p1=inter[0];
triple p2=inter[2];
dot(p1, green);
dot(p2, darkbrown);
```

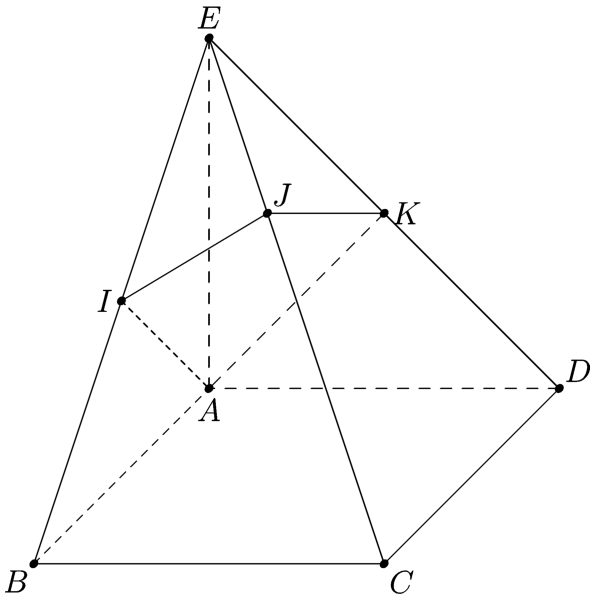
- On peut également utiliser le module **math** qui étend les routine **intersect** et **intersectionpoint** :

  `real intersect(triple P, triple Q, triple n, triple Z);`

  renvoie le réel correspondant au point d'intersection, relativement au segment [PQ], du segment [PQ] avec le plan passant par **Z** et de vecteur normal **n**.

  `triple intersectionpoint(triple n0, triple P0, triple n1, triple P1);`

  renvoie les coordonnées d'un des points d'intersection des plans de vecteurs normaux **n0** et **n1** passant respectivement par les points **P0** et **P1**.




CODE 54

```
import three;
import math;
size(8cm,0);
currentprojection=obliqueX;

triple pA, pB, pC, pD, pE;
       pA=(0,0,0);
       pB=(1,0,0);
       pC=(1,1,0);
       pD=(0,1,0);
       pE=(0,0,1);

dot("$A$",pA,S);
dot("$B$",pB,SW);
dot("$C$",pC,SE);
dot("$D$",pD,NE);
dot("$E$",pE,N);

draw(pB--pC--pD--pE--pC);
draw(pE--pB);
draw(pB--pA--pD,dashed);
draw(pA--pE,dashed);

real pouri=intersect(pB, pE, (pE-pC), pA);
triple pI = ((1-pouri)*pB+pouri*pE);
dot("$I$",pI,W);

real pourj=intersect(pC, pE, (pE-pC), pA);
triple pJ = ((1-pourj)*pC+pourj*pE);
dot("$J$",pJ,NE);

real pourk=intersect(pD, pE, (pE-pC), pA);
triple pK = ((1-pourk)*pD+pourk*pE);
dot("$K$",pK,E);

draw(pI--pJ--pK);
draw(pI--pA--pK, dashed);
```

# V – Repères et grilles

Le module **graph3** est le pendant du module **graph** en trois dimensions : il permet de définir et de tracer des axes et des repères et de dessiner des surfaces définies par des fonctions à deux variables ou des équations. Nous nous bornerons dans ce paragraphe aux repères. Les surfaces et les courbes gauches seront abordées plus loin.

## 1 . Axes et repères

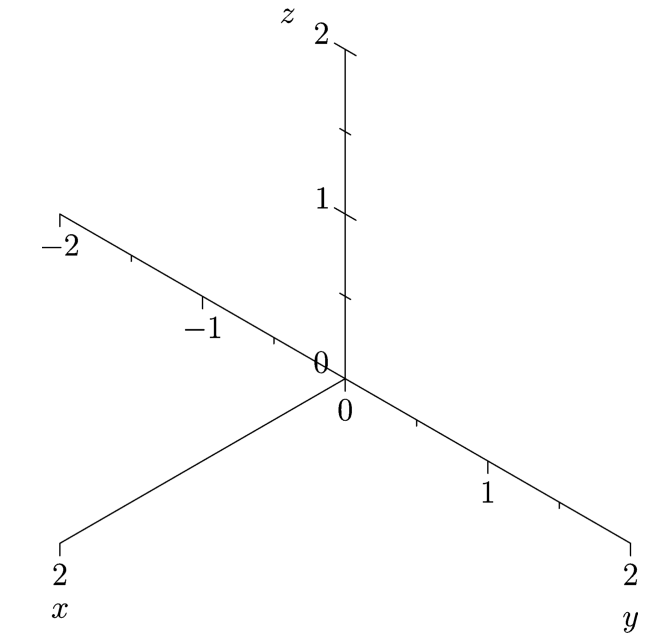
Pour tracer l'axe $(Ox)$ en trois dimensions, on peut utiliser la routine :

```
void xaxis3(picture pic=currentpicture, Label L="", axis axis=YZZero,
            real xmin=-infinity, real xmax=infinity, pen p=currentpen,
            ticks3 ticks=NoTicks3, arrowbar3 arrow=None, bool above=false);
```

où les paramètres sont similaires à ceux de **axis**() en deux dimensions :

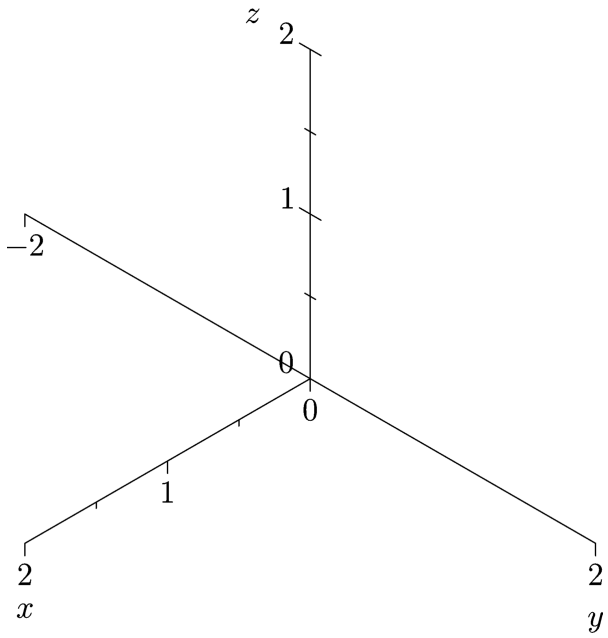
- **YZZero** : l'axe a pour équation $y = 0$ (ou 1 si l'échelle de $(Oy)$ est logarithmique) et $z = 0$ (ou 1 si l'échelle de $(Oz)$ est logarithmique).
  Pour changer ce paramètre on pourra utiliser YZequals (voir plus loin).
- **ticks3** : peut prendre les valeurs **NoTicks3**, **InTicks**, **OutTicks**, et **InOutTicks**.
- **xmin** et **xmax** sont automatiquement déterminés par les dimensions de la figure s'ils ne sont pas spécifiés.

Les routines **yaxis3** et **zaxis3** sont définies de la même façon.




CODE 55

```
import graph3;

size(8cm,0);
currentprojection=orthographic(1,1,1);

xaxis3("$x$", xmin=0, xmax=2, OutTicks());
yaxis3("$y$", ymin=-2, ymax=2, OutTicks());
zaxis3("$z$", zmin=0, zmax=2, InOutTicks());
```




CODE 56

```
import graph3;

size(8cm,0);
currentprojection=orthographic(1,1,1);

yaxis3("$y$", ymin=-2, ymax=2, OutTicks());
xaxis3("$x$", xmin=0, xmax=2, OutTicks());
zaxis3("$z$", zmin=0, zmax=2, InOutTicks());
```

Comme aura pu le remarquer le lecteur assidu, il manque les graduations sur le premier axe tracé. Ce phénomène est du à la définition de **axis3** et que le booléen pic.userSetx est **false**. Pour contourner ce problème, il suffit d'utiliser :

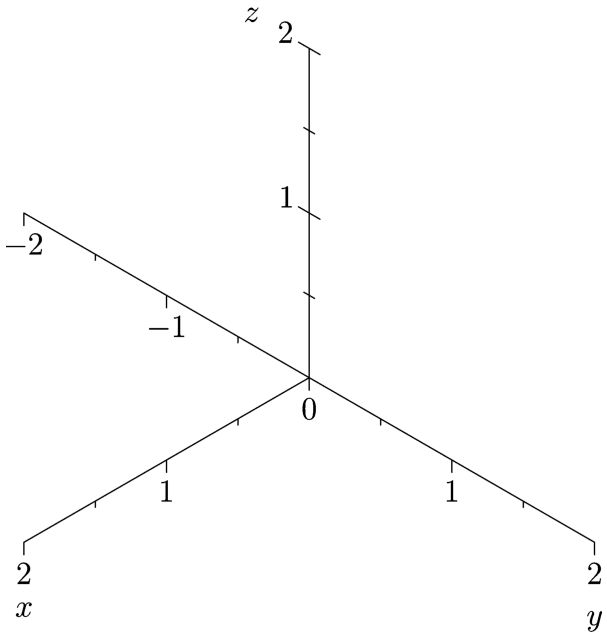
```
void limits(picture pic=currentpicture, triple min, triple max, bool crop=NoCrop);
```

où **min** et **max** détermine les sommets opposés de la boîte dans laquelle la figure est tracée. On ne peut fixer le minimum et le maximum que d'une seule coordonnée en utilisant :

```
void xlimits(picture pic=currentpicture, real min=-infinity,
                        real max=+infinity, bool crop=NoCrop);
```

ou un des ses jumeaux **ylimits** ou **zlimits**.

Dans tous les cas, le booléen **crop=Crop** sert à couper les parties de la figure qui dépassent les limites.




CODE 57

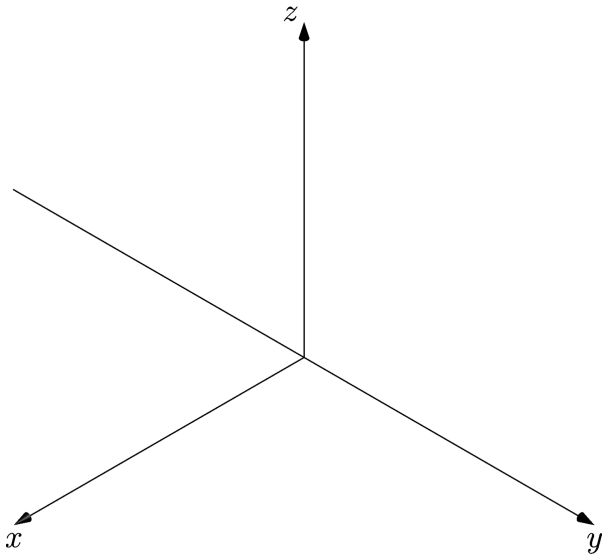
```
import graph3;
size(8cm,0);
currentprojection=orthographic(1,1,1);

limits((0,-2,0),(2,2,2));

xaxis3("$x$", OutTicks());
yaxis3("$y$", OutTicks());
zaxis3("$z$", InOutTicks(NoZero));
```

Si l'on veut tracer les trois axes en même temps, on peut utiliser :

```
void axes3(picture pic=currentpicture,
           Label xlabel="", Label ylabel="", Label zlabel="",
           triple min=(-infinity,-infinity,-infinity),
           triple max=(infinity,infinity,infinity),
           pen p=currentpen, arrowbar3 arrow=None);
```

CODE 58

```
import graph3;

size(8cm,0);
currentprojection=orthographic(1,1,1);

axes3("$x$", "$y$", "$z$",
      min=(0,-2,0),
      max=(2,2,2),
      Arrow3);
```



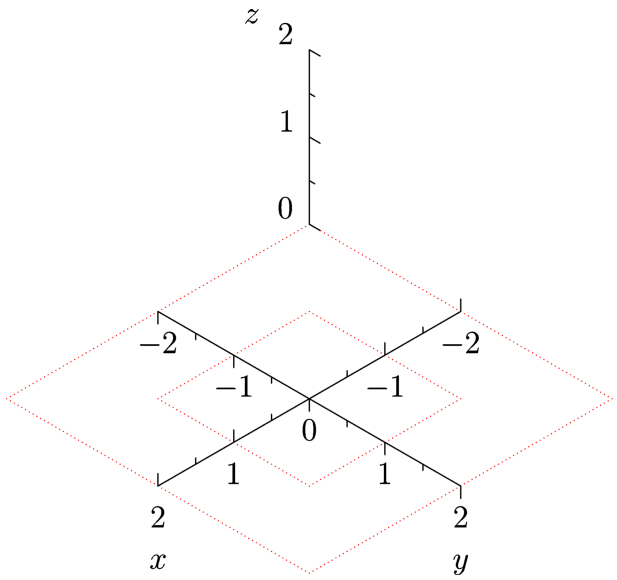
Les routines

```
axis YZZero(triple align=0, bool extend=false);
axis XZZero(triple align=0, bool extend=false);
axis XYZero(triple align=0, bool extend=false);
```

sont des cas particuliers de :

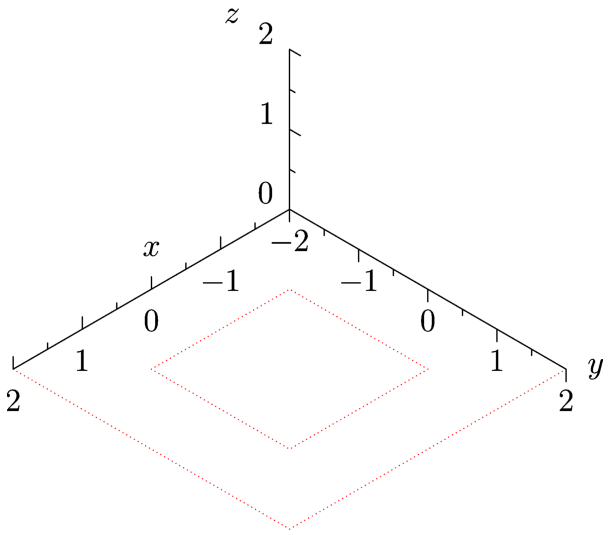
```
axis YZEquals(real y, real z, triple align=0, bool extend=false);
axis XZEquals(real x, real z, triple align=0, bool extend=false);
axis XYEquals(real x, real y, triple align=0, bool extend=false);
```

qui permettent de placer les axes où l'on veut.




CODE 59

```
import graph3;

size(8cm,0);
currentprojection=orthographic(1,1,1);
limits((-2,-2,0),(2,2,2));

xaxis3("$x$", InTicks(NoZero));
yaxis3("$y$", OutTicks());
zaxis3("$z$", XYEquals(-2,-2), InTicks());

path3 g=(2,-2,0)--(-2,-2,0)--(-2,2,0)
        --(2,2,0)--cycle;
draw(g, red+dotted);
draw(scale3(0.5)*g, red+dotted);
```




CODE 60

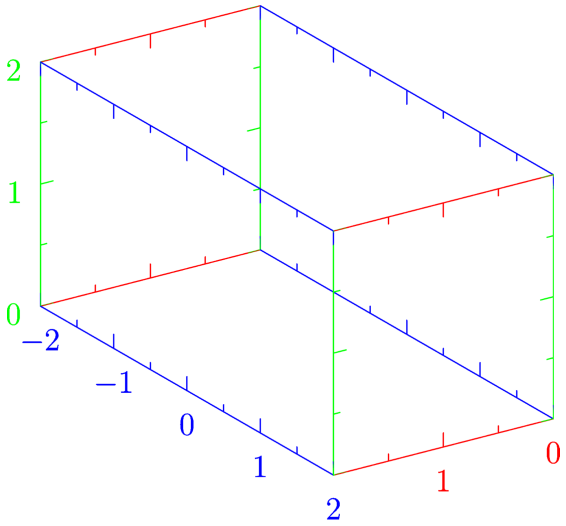
```
import graph3;

size(8cm,0);
currentprojection=orthographic(1,1,1);
limits((-2,-2,0),(2,2,2));

xaxis3(Label("$x$", position=MidPoint, align=N),
       YZEquals(-2,0), InTicks());
yaxis3(Label("$y$", position=EndPoint, align=E),
       XZEquals(-2,0), OutTicks());
zaxis3("$z$",
       XYEquals(-2,-2), InTicks());

path3 g=(2,-2,0)--(-2,-2,0)--(-2,2,0)
        --(2,2,0)--cycle;
draw(g, red+dotted);
draw(scale3(0.5)*g, red+dotted);
```

**Remarque :**
Comme on peut le voir dans l'exemple précédent, les paramètres optionnels **position** et **align** peuvent servir à définir la position du **label** et les alignements des **ticks**.
Le paramètre **position** peut être égal à **BeginPoint**, **MidPoint** ou **EndPoint**.

Dernière routine pour les axes en trois dimensions :

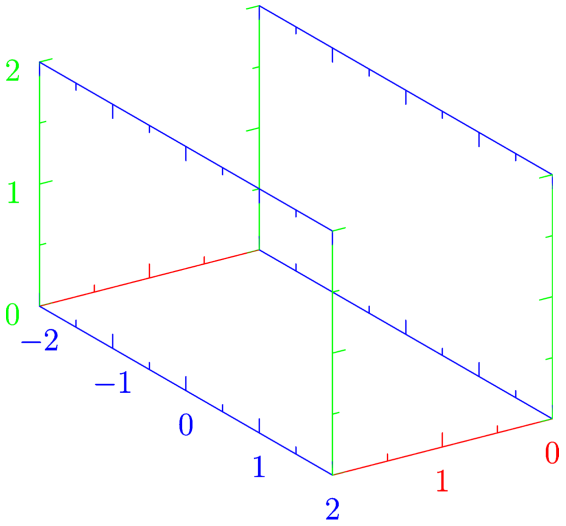
```
axis Bounds(int type=Both, int type2=Both, triple align=0, bool extend=false);
```

où les paramètres type et type2 peuvent prendre comme valeur **Min**, **Max** ou **Both**.
Ils spécifient les quatre directions possibles pour les arêtes de la boîtes



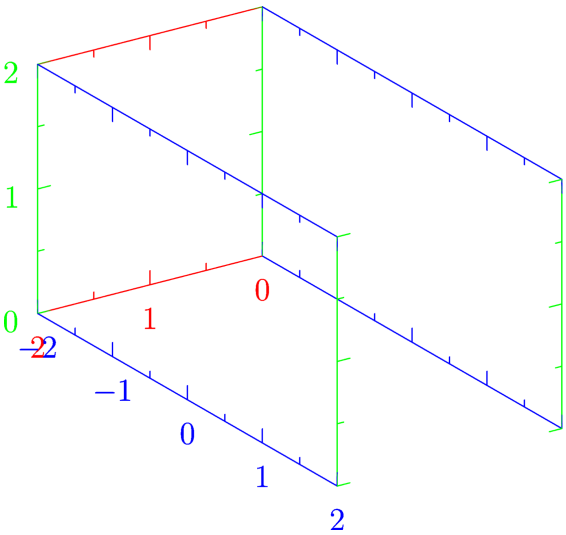

CODE 61

```
import graph3;
currentprojection=orthographic(4,6,3);
size(7.5cm);
limits((0,-2,0),(2,2,2));

xaxis3(Bounds(),red,InTicks(Label,2,2));
yaxis3(Bounds(),blue,InTicks());
zaxis3(Bounds(),green,InTicks(Label,2,2));
```



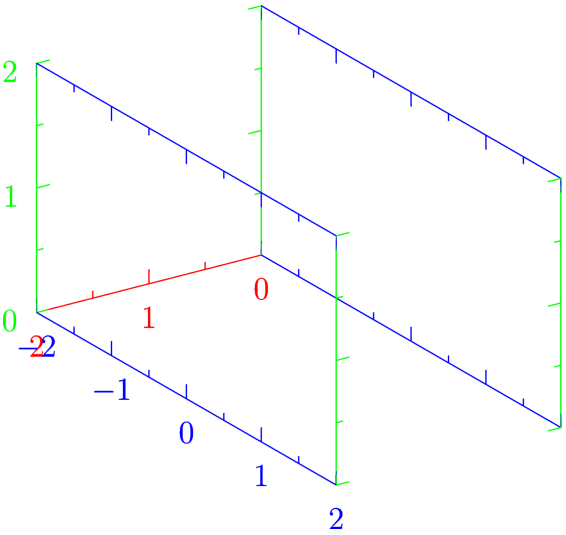

CODE 62

```
import graph3;
currentprojection=orthographic(4,6,3);
size(7.5cm);
limits((0,-2,0),(2,2,2));

xaxis3(Bounds(Both, Min),red,InTicks(Label,2,2));
yaxis3(Bounds(),blue,InTicks());
zaxis3(Bounds(),green,InTicks(Label,2,2));
```



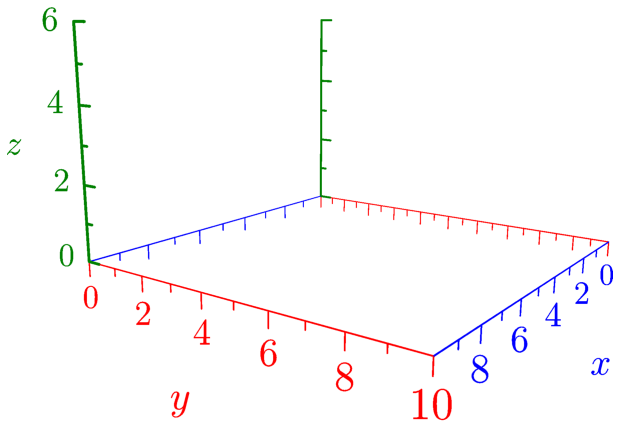

CODE 63

```
import graph3;
currentprojection=orthographic(4,6,3);
size(7.5cm);
limits((0,-2,0),(2,2,2));

xaxis3(Bounds(Min, Both),red,InTicks(Label,2,2));
yaxis3(Bounds(),blue,InTicks());
zaxis3(Bounds(),green,InTicks(Label,2,2));
```




CODE 64

```
import graph3;
currentprojection=orthographic(4,6,3);
size(7.5cm);
limits((0,-2,0),(2,2,2));

xaxis3(Bounds(Min, Min),red,InTicks(Label,2,2));
yaxis3(Bounds(),blue,InTicks());
zaxis3(Bounds(),green,InTicks(Label,2,2));
```

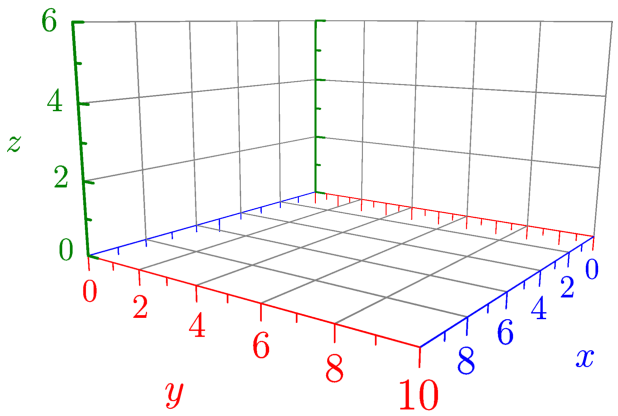



CODE 65

```
import graph3;
size(8cm);
limits((0,0,0),(10,10,6));
currentprojection=perspective(camera=(20,16,7));

xaxis3(Label("$x$",MidPoint,align=Y-Z),
       Bounds(Both,Min),OutTicks(endlabel=false),p=blue);
yaxis3(Label("$y$",MidPoint,align=X-Z),
       Bounds(Both,Min),OutTicks(),p=red);
zaxis3(Label("$z$",MidPoint,align=X-Y),
       Bounds(Both,Min),InTicks(),p=1bp+.5green);
```

Je vous entends déjà, c'est pas mal mais avec une grille ça aurait meilleure allure ...
Pas de problème ! ! !

## 2 . Grille




CODE 66

```
import grid3;
size(8cm);
limits((0,0,0),(10,10,6));
currentprojection=perspective(camera=(20,16,7));

grid3(XYZgrid,Step=2);

xaxis3(Label("$x$",MidPoint,align=Y-Z),
       Bounds(Both,Min),OutTicks(endlabel=false),p=blue);
yaxis3(Label("$y$",MidPoint,align=X-Z),
       Bounds(Both,Min),OutTicks(),p=red);
zaxis3(Label("$z$",MidPoint,align=X-Y),
       Bounds(Both,Min),InTicks(),p=1bp+.5green);
```

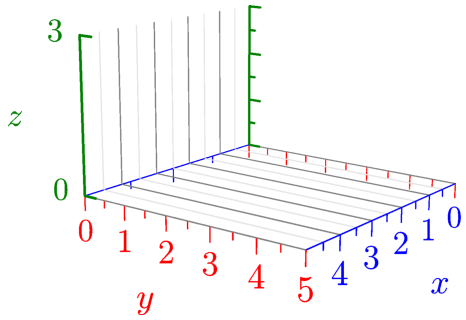
L'exemple précédent a été pris sur le site de Gaétan Marris [5] mais nous devons le module **grid3.asy** à Philippe Ivaldi [4]. Voila comment appeler la routine **grid3** :

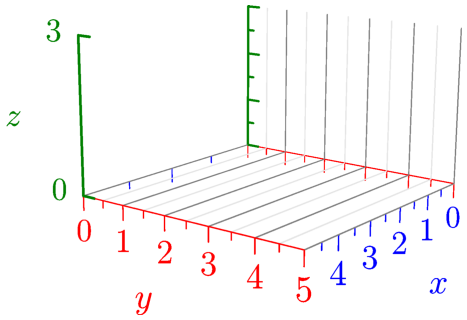
```
void grid3(picture pic=currentpicture,
           grid3routinetype gridroutine=XYZgrid(
           real pos=Relative(0)),
           int N=0,
           int n=0,
           real Step=0,
           real step=0,
           bool begin=true,
           bool end=true,
           pen pGrid=grey,
           pen pgrid=lightgrey,
           bool above=false)
```

où gridroutine est la routine pour dessiner les grilles :

- **XYgrid** dessine une grille sur le plan $(xOy)$ dans la direction $(Oy)$ ;
  **YXgrid** dessine une grille sur le plan $(xOy)$ dans la direction $(Ox)$ ;
  etc.
- **XYXgrid** dessine les grilles **XYgrid** et **YXgrid** ;
  **YXYgrid** dessine les grilles **XYgrid** et **YXgrid** ;
  **ZXZgrid** dessine les grilles **ZXgrid** et **XZgrid** ;
  etc.
- **YX_YZgrid** dessine les grilles **YXgrid** et **YZgrid** ;
  **XY_XZgrid** dessine les grilles **XYgrid** et **XZgrid** ;
  **ZX_ZYgrid** dessine les grilles **ZXgrid** et **ZYgrid**.

- **XYZgrid** dessine **XYXgrid**, **ZYZgrid** et **XZXgrid**.



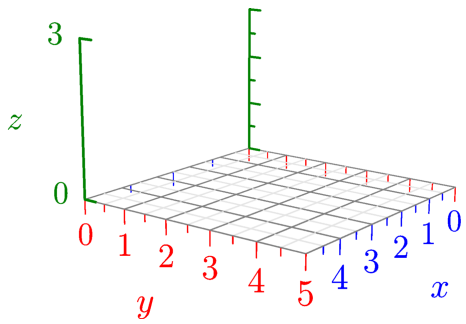




**CODE 67**

```
import grid3;
size(6cm);
limits((0,0,0),(5,5,3));
currentprojection=perspective(camera=(20,16,7));

xaxis3(Label("$x$",MidPoint,align=Y-Z),
       Bounds(Both,Min),
       OutTicks(endlabel=false),p=blue);
yaxis3(Label("$y$",MidPoint,align=X-Z),
       Bounds(Both,Min),
       OutTicks(),p=red);
zaxis3(Label("$z$",MidPoint,align=X-Y),
       Bounds(Both,Min),
       InTicks(),p=1bp+.5green);

grid3(XYgrid);
grid3(XZgrid);
//grid3(YZgrid);
```

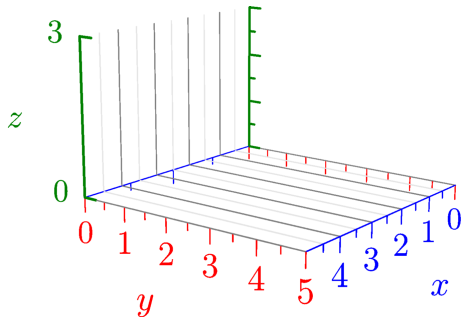
**CODE 68**

```
import grid3;
size(6cm);
limits((0,0,0),(5,5,3));
currentprojection=perspective(camera=(20,16,7));

xaxis3(Label("$x$",MidPoint,align=Y-Z),
       Bounds(Both,Min),
       OutTicks(endlabel=false),p=blue);
yaxis3(Label("$y$",MidPoint,align=X-Z),
       Bounds(Both,Min),
       OutTicks(),p=red);
zaxis3(Label("$z$",MidPoint,align=X-Y),
       Bounds(Both,Min),
       InTicks(),p=1bp+.5green);

grid3(YXgrid);
//grid3(XZgrid);
grid3(YZgrid);
```




CODE 69

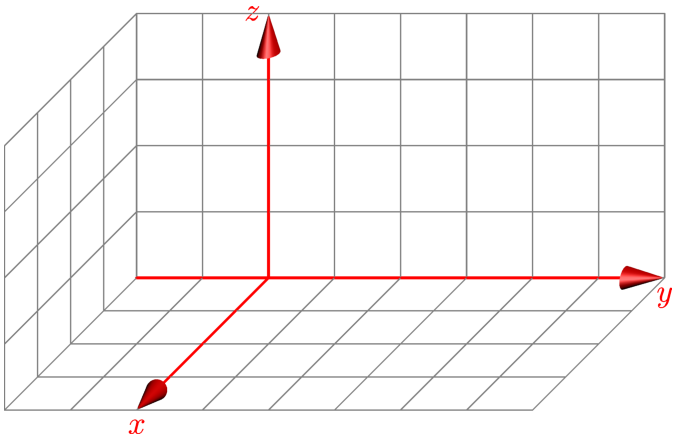
```
import grid3;
size(6cm);
limits((0,0,0),(5,5,3));
currentprojection=perspective(camera=(20,16,7));

xaxis3(Label("$x$",MidPoint,align=Y-Z),
       Bounds(Both,Min),
       OutTicks(endlabel=false),p=blue);
yaxis3(Label("$y$",MidPoint,align=X-Z),
       Bounds(Both,Min),
       OutTicks(),p=red);
zaxis3(Label("$z$",MidPoint,align=X-Y),
       Bounds(Both,Min),
       InTicks(),p=1bp+.5green);

grid3(XYXgrid);
```




CODE 70

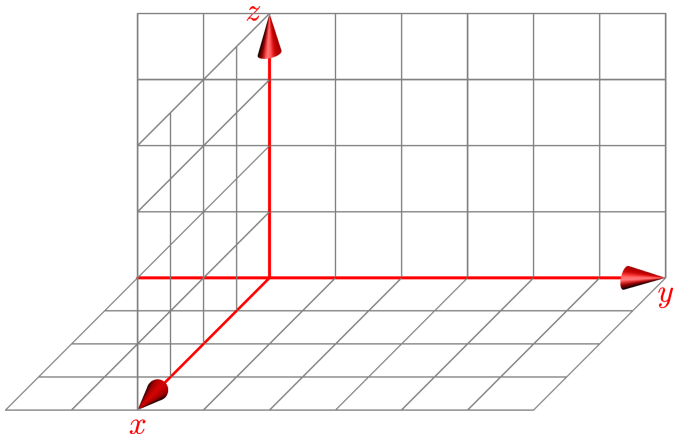
```
import grid3;
size(6cm);
limits((0,0,0),(5,5,3));
currentprojection=perspective(camera=(20,16,7));

xaxis3(Label("$x$",MidPoint,align=Y-Z),
       Bounds(Both,Min),
       OutTicks(endlabel=false),p=blue);
yaxis3(Label("$y$",MidPoint,align=X-Z),
       Bounds(Both,Min),
       OutTicks(),p=red);
zaxis3(Label("$z$",MidPoint,align=X-Y),
       Bounds(Both,Min),
       InTicks(),p=1bp+.5green);

grid3(XY_XZgrid);
//grid3(YX_YZgrid);
```

Le paramètre **Relative**(**real x**=0) définit la position de la grille par rapport à l'axe qui lui est perpendiculaire. pos=**top**, pos=**middle** et pos=**bottom** peuvent être utilisés respectivement pour **Relative**(1), **Relative**(0.5) et **Relative**(0).




CODE 71

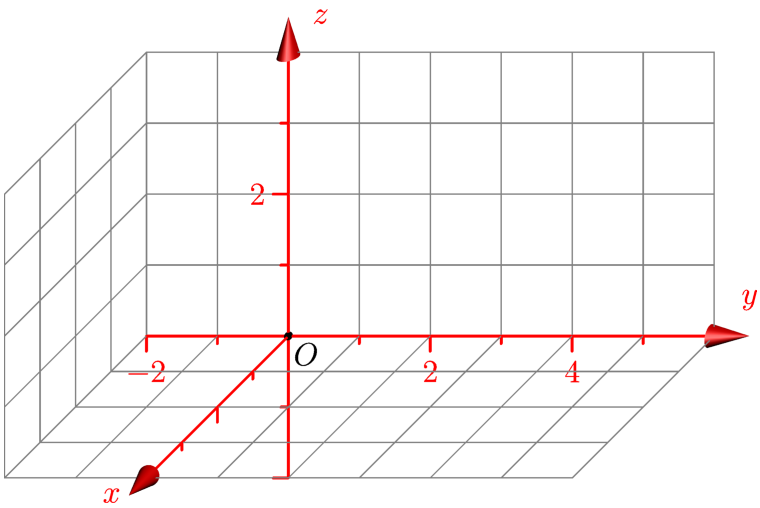
```
import graph3;
import grid3;

size3(7cm);
currentprojection=obliqueX;

limits((0,-2,0),(4,6,4));

grid3(XYXgrid,Step=1,step=0);
grid3(ZYZgrid,Step=1,step=0);
grid3(XZXgrid,Step=1,step=0);

axes3("$x$", "$y$", "$z$",
      red+linewidth(1pt), Arrow3);
```




CODE 72

```
import graph3;
import grid3;

size3(7cm);
currentprojection=obliqueX;

limits((0,-2,0),(4,6,4));

grid3(XYXgrid,Step=1,step=0);
grid3(ZYZgrid,Step=1,step=0);
grid3(XZXgrid(Relative(0.25)),Step=1,step=0);

axes3("$x$", "$y$", "$z$",
      red+linewidth(1pt), Arrow3);
```

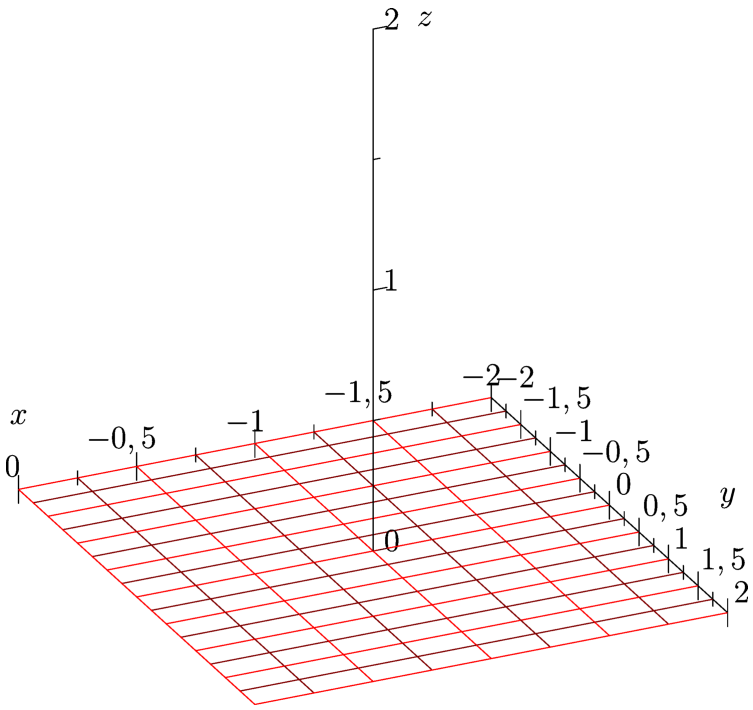
On peut également « costumiser » la grille avec les derniers paramètres qui sont similaires à ceux utilisés dans **Ticks**.




CODE 73

```
import graph3;
import grid3;

size3(8cm);
currentprojection=obliqueX;

limits((0,-2,0),(4,6,4));

grid3(XYXgrid,Step=1,step=0);
grid3(ZYZgrid,Step=1,step=0);
grid3(XZXgrid,Step=1,step=0);

xaxis3(Label("$x$",position=EndPoint,align=W),
             0,4.5,
             red+linewidth(1pt),
             OutTicks(NoZero,
                      beginlabel=false,
                      endlabel=false,
                      Step=2,step=1,
                      end=false),
              Arrow3);
yaxis3(Label("$y$",position=EndPoint,align=N),
             -2,6.5,
             red+linewidth(1pt),
             OutTicks(NoZero,
                      beginlabel=true,
                      endlabel=false,
                      Step=2,step=1,
                      end=false),
              Arrow3);
zaxis3(Label("$z$",position=EndPoint,align=E),
             -2,4.5,
             red+linewidth(1pt),
             OutTicks(NoZero,
                      beginlabel=false,
                      endlabel=false,
                      Step=2,step=1,
                      end=false),
             Arrow3);
dot("$O$",O,SE);
```

Dans le dernier exemple de Philippe Ivaldi [4], on appelle **XYgrid** et **YXgrid** directement dans les routines **xaxis3** et **yaxis3**.




CODE 74

```
import grid3;

size(10cm,0,IgnoreAspect);
currentprojection=orthographic(0.25,1,0.25);
limits((-2,-2,0),(0,2,2));

real Step=0.5, step=0.25;
xaxis3(Label("$x$",position=EndPoint,align=Z), YZEquals(-2,0),
       InOutTicks(Label(align=0.5*(Z-Y)),
                  Step=Step, step=step,
                  gridroutine=XYgrid,
                  pGrid=red, pgrid=0.5red));

yaxis3(Label("$y$",position=EndPoint,align=Z), XZEquals(-2,0),
       InOutTicks(Label(align=-0.5*(X-Z)), Step=Step, step=step,
                  gridroutine=YXgrid,
                  pGrid=red, pgrid=0.5red));

zaxis3("$z$", XYEquals(-1,0), OutTicks(Label(align=-0.5*(X+Y))));
```

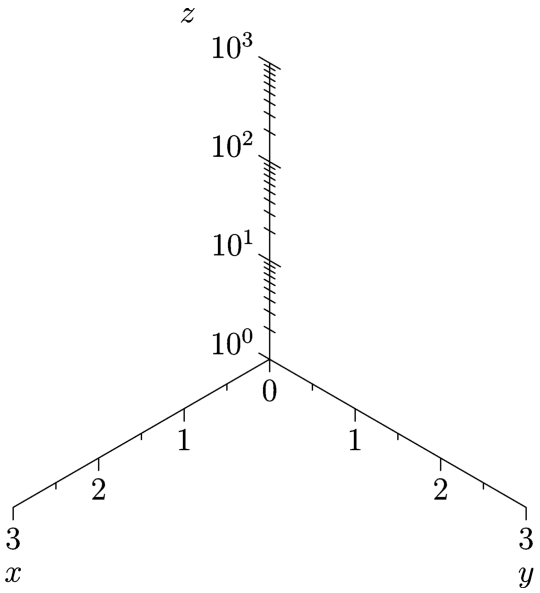
### 3 . Repère semi-logarithmique

Pour dessiner des repères semi-logarithmiques, on peut utiliser une routine définie dans le module **graph.asy** (en deux dimensions donc) :

```
void scale(picture pic=currentpicture, scaleT x, scaleT y=x, scaleT z=y)
```

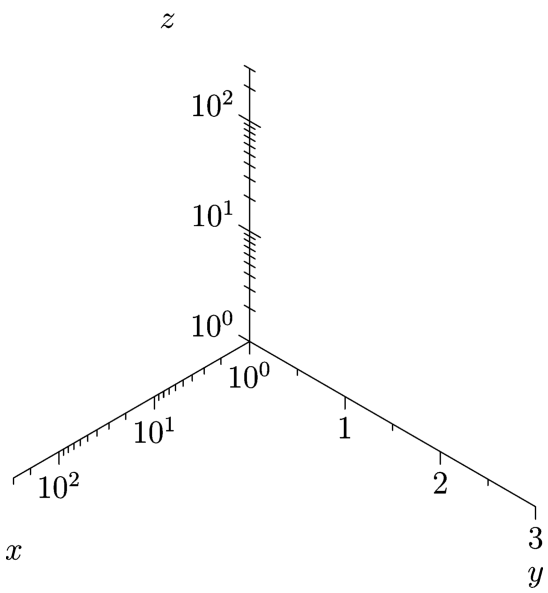
où **scaleT** peut prendre (entre autres) les valeurs **Linear** ou **Log** :

```
scaleT Linear;

scaleT Log=scaleT(log10,pow10,logarithmic=true);
scaleT Logarithmic=Log;
```




CODE 75

```
import graph3;
size(7.5cm);
currentprojection=orthographic(1,1,1);

limits((0,0,0),(3,3,3));
scale(Linear,Linear,Log);

xaxis3("$x$", OutTicks());
yaxis3("$y$", OutTicks());
zaxis3("$z$", InOutTicks());
```

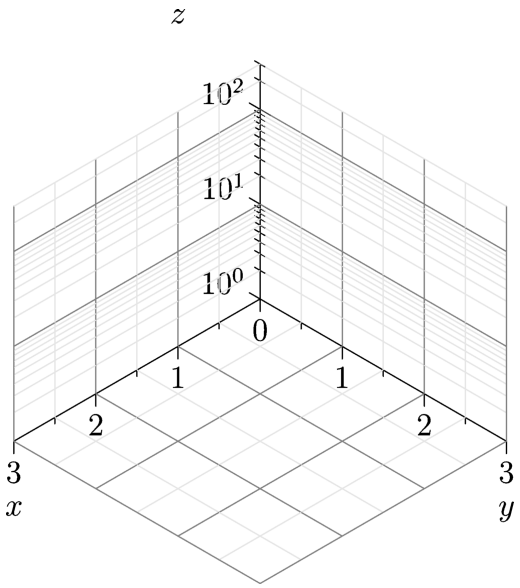



CODE 76

```
import graph3;
size(7.5cm);
currentprojection=orthographic(1,1,1);

scale(Log,Linear,Log);
limits((1,0,1),(300,3,300));

xaxis3("$x$", OutTicks());
yaxis3("$y$", OutTicks(NoZero));
zaxis3("$z$", InOutTicks());
```

On remarquera l'importance de l'ordre donné aux commandes **scale** et **limits**.

Les grilles suivent bien évidemment :




CODE 77

```
import grid3;
size(7.5cm);
currentprojection=orthographic(1,1,1);

scale(Log,Linear,Log);
limits((1,0,1),(300,3,300));

grid3(XYZgrid);

xaxis3("$x$", OutTicks());
yaxis3("$y$", OutTicks(NoZero));
zaxis3("$z$", InOutTicks());
```

# VI – Courbes et surfaces

On sait déjà tracer des surfaces définies par des chemins fermés. Nous verrons dans ce paragraphe comment représenter les surfaces définies à l'aide de fonctions à plusieurs variables.

## 1 . Surfaces d'équation $z = f(x; y)$

Pour définir une surface d'équation $z = f(x; y)$, on peut utiliser une des routines :

```
surface surface(real f(pair z), pair a, pair b, int nx=nmesh, int ny=nx,
                bool cond(pair z)=null);
surface surface(real f(pair z), pair a, pair b, int nx=nmesh, int ny=nx,
                splinetype xsplinetype, splinetype ysplinetype=xsplinetype,
                bool cond(pair z)=null);
```

Dans les deux cas,

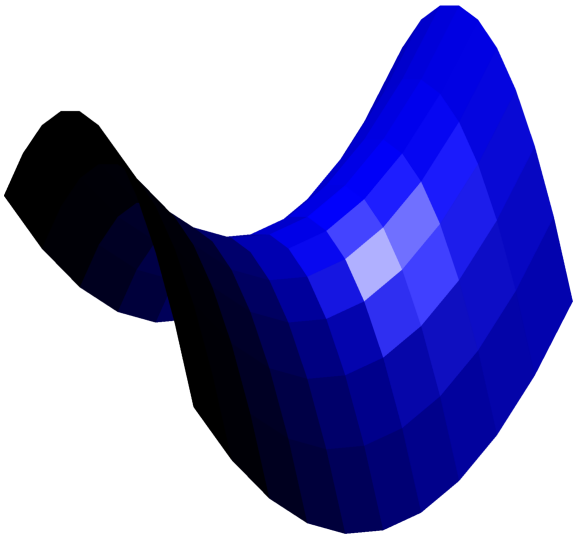
**real** f(**pair z**) désigne la fonction de deux variables qu'il faudra définir préalablement ;

**pair** a et **pair** b détermine la boîte dans laquelle sera tracée la surface, c'est-à-dire, a=(**xmin**,**ymin**) et b=(**xmax**,**ymax**).

Les paramètres **nx** et **ny** détermine le nombres de « carreaux de Bézier ». Par défaut, **nmesh**=10.

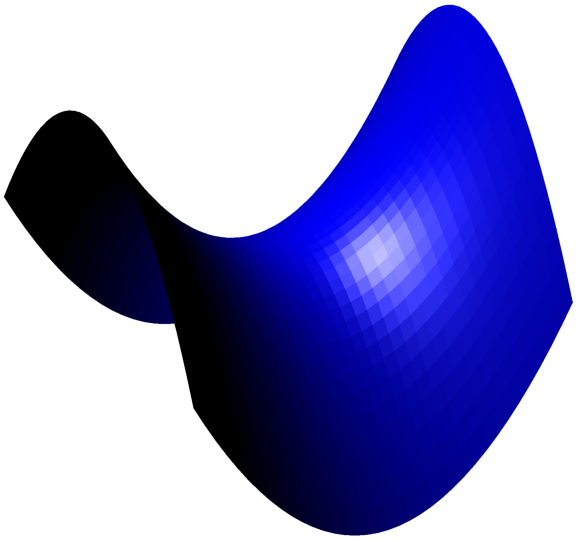
les derniers paramètres de la seconde routine permettent de déterminer quel type d'interpolation utiliser pour construire ces carreaux de Bézier.

L'influence du paramètre **nx** apparait simplement sur les deux exemples suivants :



CODE 78

```
import graph3;
currentprojection=orthographic(2,4,3);
currentlight=(2,0,3);
size(7.5cm,0);

real f(pair z) {return z.x^2-z.y^2;}
surface s=surface(f, (-1,-1), (1,1), nx=10);
draw(s,blue);
```



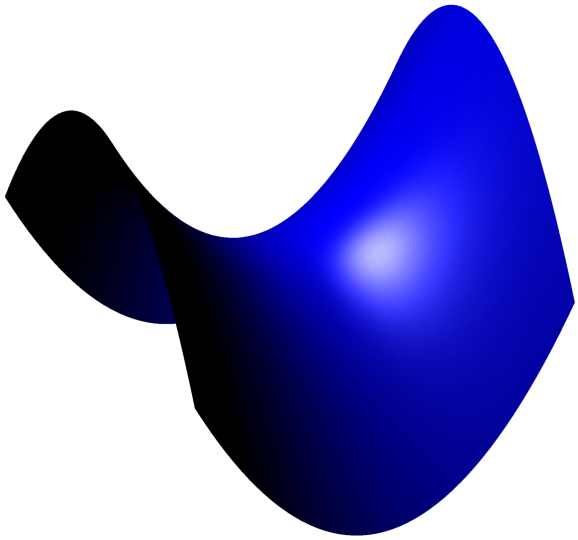
CODE 79

```
import graph3;
currentprojection=orthographic(2,4,3);
currentlight=(2,0,3);
size(7.5cm,0);

real f(pair z) {return z.x^2-z.y^2;}
surface s=surface(f, (-1,-1), (1,1), nx=35);
draw(s,blue);
```

L'influence du type d'interpolation est encore plus flagrante. On peut utiliser :

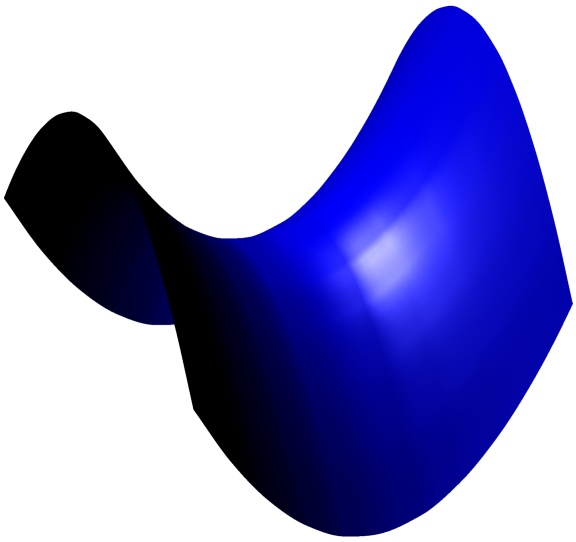
- **Spline** qui correspondrait à des interpolations cubiques par des courbes de Béziers.
- **notaknot**, **natural**, **periodic**, **clamped**(**real** slopea, **real** slopeb) ou **monotonic** qui correspondraient à des options d'interpolation (cubique) de l'Hermite.

**Remarque :**
Dans le pratique, j'utilise le plus souvent les paramètres **nx** et **ny** ou le type d'interpolation **Spline**. Étant loin d'être un spécialiste d'analyse numérique, je ne peux pas vous en dire plus.



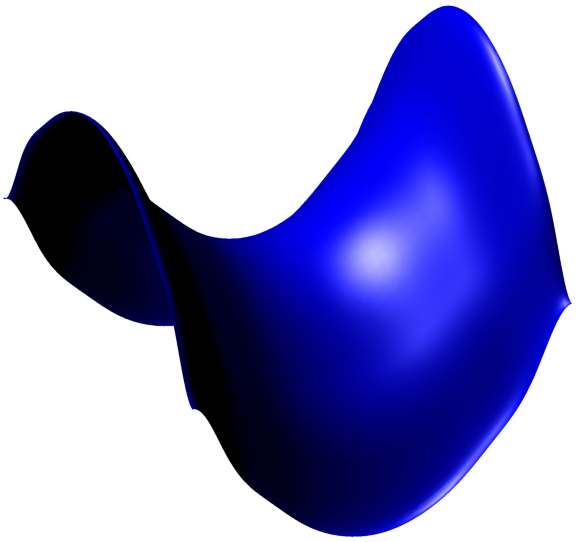
CODE 80

```
import graph3;
currentprojection=orthographic(2,4,3);
currentlight=(2,0,3);
size(7.5cm,0);

real f(pair z) {return z.x^2-z.y^2;}
surface s=surface(f, (-1,-1), (1,1), Spline);
draw(s,blue);
```



CODE 81

```
import graph3;
currentprojection=orthographic(2,4,3);
currentlight=(2,0,3);
size(7.5cm,0);

real f(pair z) {return z.x^2-z.y^2;}
surface s=surface(f, (-1,-1), (1,1), monotonic);
draw(s,blue);
```



CODE 82

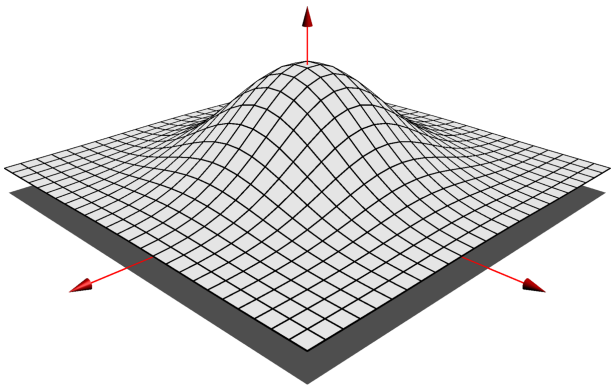
```
import graph3;
currentprojection=orthographic(2,4,3);
currentlight=(2,0,3);
size(7.5cm,0);

real f(pair z) {return z.x^2-z.y^2;}
surface s=surface(f, (-1,-1), (1,1), periodic);
draw(s,blue);
```

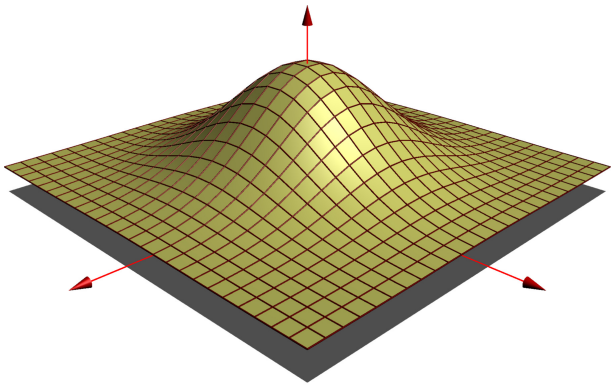
Il ne faut pas non plus oublier les paramètres optionnels de la routine **draw** :

```
void draw(picture pic=currentpicture, surface s, int nu=1, int nv=1,
          material surfacepen=currentpen, pen meshpen=nullpen,
          light light=currentlight, light meshlight=light, string name="",
          render render=defaultrender);
void draw(picture pic=currentpicture, surface s, int nu=1, int nv=1,
```

```
          material[] surfacepen, pen meshpen,
          light light=currentlight, light meshlight=light, string name="",
          render render=defaultrender);
void draw(picture pic=currentpicture, surface s, int nu=1, int nv=1,
          material[] surfacepen, pen[] meshpen=nullpens,
          light light=currentlight, light meshlight=light, string name="",
          render render=defaultrender);
```

En particulier, les paramètres **meshpen** et **light** retiendront notre attention :





CODE 83

```
import graph3;
currentprojection=perspective(2,2,2);
size(8cm);

real a=2;
real f(pair z) {return exp(-abs(z)^2)+0.25;}
surface s=surface(f, (-a,-a), (a,a), nx=25);
path3 pl=plane(2*(a,0,0), 2*(0,a,0), (-a,-a,0));

draw(surface(pl), gray);
draw(s,lightgray, meshpen=black+thick(), nolight);
xaxis3("",0,a+0.5,red,Arrow3);
yaxis3("",0,a+0.5,red,Arrow3);
zaxis3("",0,1.75,red,Arrow3);
```

CODE 84

```
import graph3;
currentprojection=perspective(2,2,2);
size(8cm);

real a=2;
real f(pair z) {return exp(-abs(z)^2)+0.25;}
surface s=surface(f, (-a,-a), (a,a), nx=25);
path3 pl=plane(2*(a,0,0), 2*(0,a,0), (-a,-a,0));

draw(surface(pl), gray);
draw(s,lightyellow, meshpen=brown+thick());
xaxis3("",0,a+0.5,red,Arrow3);
yaxis3("",0,a+0.5,red,Arrow3);
zaxis3("",0,1.75,red,Arrow3);
```

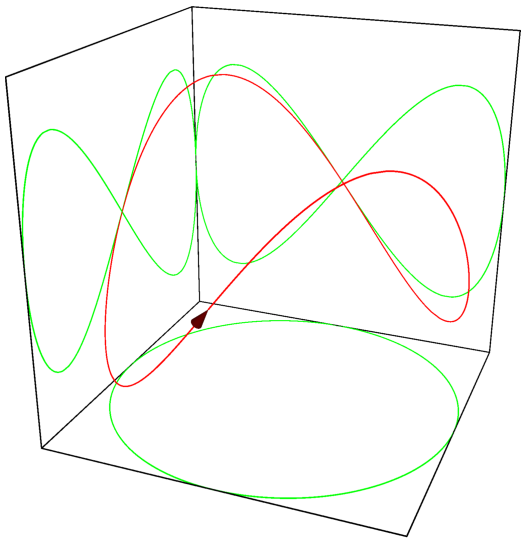
Quant à **material** :

```
struct material {
                pen[] p; // diffusepen,ambientpen,emissivepen,specularpen
                real opacity;
                real shininess;
                ...
}
```

celui qui voudra approfondir devra fouiller le fichier three_light.asy.

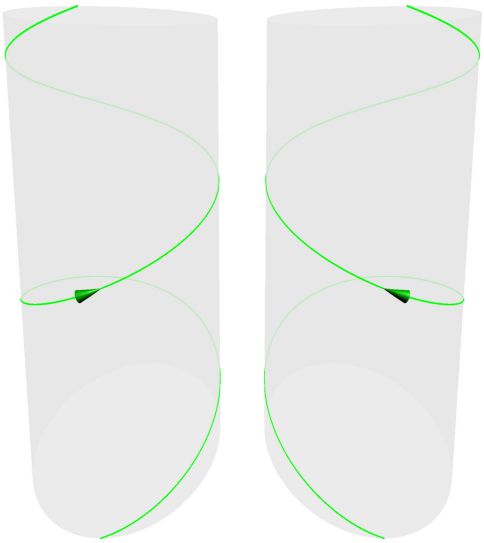
## 2. Courbes gauches et surfaces paramétrées

Les courbes paramétriques se définissent comme dans le plan. Regardons les exemples d'une courbe de crêpe et des hélices circulaires.



CODE 85

```
import grid3;
size(0,200);
size3(200,IgnoreAspect);
currentprojection=perspective(5,2,5);

real x(real t) {return cos(t);}
real y(real t) {return sin(t);}
real z(real t) {return sin(2*t)+1.5;}

path3 p=graph(x,y,z,0,2*pi,operator ..);
draw(p,red,Arrow3);

draw((-1,-1,3)--(-X-Y)--(-X+Y)--(X+Y)
                        --(X-Y)--(-X-Y));
draw((-1,-1,3)--(1,-1,3)--(1,-1,0));
draw((-1,-1,3)--(-1,1,3)--(-1,1,0));

transform3 pr1=planeproject(X,(-1,0,0));
path3 p1=pr1*p;
draw(p1,green);
transform3 pr2=planeproject(Y,(0,-1,0));
path3 p2=pr2*p;
draw(p2,green);
transform3 pr3=planeproject(Z,(0,0,0));
path3 p3=pr3*p;
draw(p3,green);
```



CODE 86

```
import graph3;
currentprojection=perspective(4,0,2);
size(7.5cm,200,IgnoreAspect);

real x(real t) {return cos(2pi*t);}
real y(real t) {return sin(2pi*t);}
real y2(real t) {return -sin(2pi*t);}
real z(real t) {return t;}

path3 p=graph(x,y,z,0,2,operator ..);
draw(shift(0,-1.25,0)*zscale3(2)*unitcylinder,
                lightgray+opacity(0.8),
                nolight);
draw(shift(0,-1.25,0)*p, green, MidArrow3);

path3 p2=graph(x,y2,z,0,2,operator ..);
draw(shift(0,1.25,0)*zscale3(2)*unitcylinder,
                lightgray+opacity(0.8),
                nolight);
draw(shift(0,1.25,0)*p2, green, MidArrow3);
```

Et la balle de tennis :



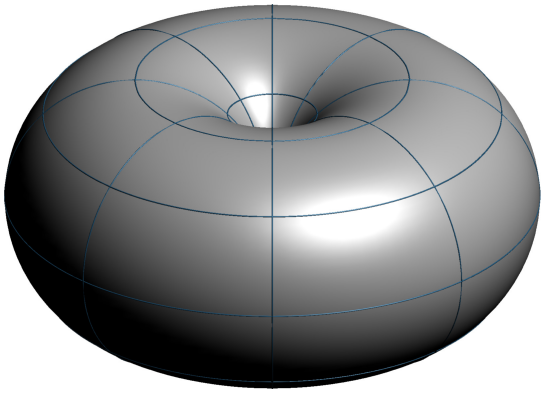
CODE 87

```
import graph3;
currentprojection=orthographic(
        camera=(2.15730812876489,-2.844599737062,-3.88952071904703),
        up=(0.00705050928241745,-0.00766843678995264,0.00951885485929442),
        target=(-4.44089209850063e-16,4.44089209850063e-16,-4.44089209850063e-16),
        zoom=1);
size(6cm,0);

real a=1;
real b=0.5;
real c=2*sqrt(a*b);

real x(real t) {return a*cos(t) + b*cos(3*t);}
real y(real t) {return a*sin(t) - b*sin(3*t);}
real z(real t) {return c*sin(2*t);}

draw(scale3(a+b)*unitsphere, lightyellow+opacity(0.5), nolight);
path3 s=graph(x,y,z,0,2*pi);
draw(s,white+linewidth(10pt));
```

Pour les surfaces paramétriques on utilisera les routines :

```
surface surface(triple f(pair z), pair a, pair b, int nu=nmesh, int nv=nu,
        bool cond(pair z)=null);
surface surface(triple f(pair z), pair a, pair b, int nu=nmesh, int nv=nu,
        splinetype[] usplinetype, splinetype[] vsplinetype=Spline,
        bool cond(pair z)=null);
```

qui permettent de dessiner une surface définie par f(**u**,**v**) où (**u**,**v**) décrit la boite **box**(a,b).
Comme illustration, un exemple de la documentation officielle et le ruban de Moëbius :



CODE 88

```
import graph3;

size(200,0);
currentprojection=orthographic(4,0,2);

real R=2;
real a=1.9;

triple f(pair t) {
  return ((R+a*cos(t.y))*cos(t.x),
          (R+a*cos(t.y))*sin(t.x),
          a*sin(t.y));
}

pen p=rgb(0.2,0.5,0.7);
surface s=surface(f,(0,0),(2pi,2pi),8,8,Spline);
draw(s,lightgray,meshpen=p+thick());
```

CODE 89

```
import graph3;
currentprojection=orthographic(5,18,10);
currentlight=light((15,-25,-5),(-15,15,5));
size(200,0);
int k=1;

triple f(pair z) {
       return ((2+z.x*cos(k*z.y))*cos(2*z.y),
               (2+z.x*cos(k*z.y))*sin(2*z.y),
               z.x*sin(k*z.y));}
surface s=surface(f,(-1,0),(1,pi),Spline);
draw(s,brown);
```

## 3. Surface définie par une matrice

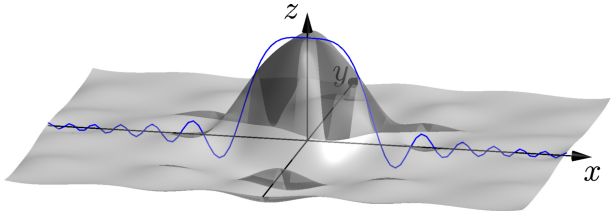
Lorsque que l'on possède les coordonnées des points de la surfaces on peut utiliser les routines :

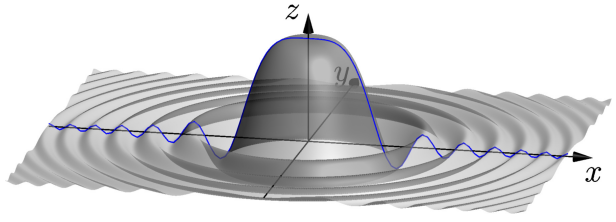
```
surface surface(real[][] f, pair a, pair b, bool[][] cond={});
surface surface(real[][] f, pair a, pair b, splinetype xsplinetype,
                splinetype ysplinetype=xsplinetype, bool[][] cond={});
surface surface(real[][] f, real[] x, real[] y,
                splinetype xsplinetype=null, splinetype ysplinetype=xsplinetype,
                bool[][] cond={})
surface surface(triple[][] f, bool[][] cond={});
```

Je vous renvoie pour leur utilisation aux exemples teapot.asy et BezierSurface.asy de la gallerie d'exemples du site officiel.

## 4. À propos de la compilation

Il ne faut pas hésiter à augmenter le paramètre **nmesh**. La compilation supporte très bien tout des valeurs autour de 300. Par exemple, on voit bien ici qu'avec la valeur de **nmesh** par défaut, (**nmesh**=10), la surface générée n'est pas très belle et la courbe ne se « collent » pas à la surface. Avec **nmesh**=300, la surface a bien meilleure allure et le chemin bleu « colle » bien à la surface.

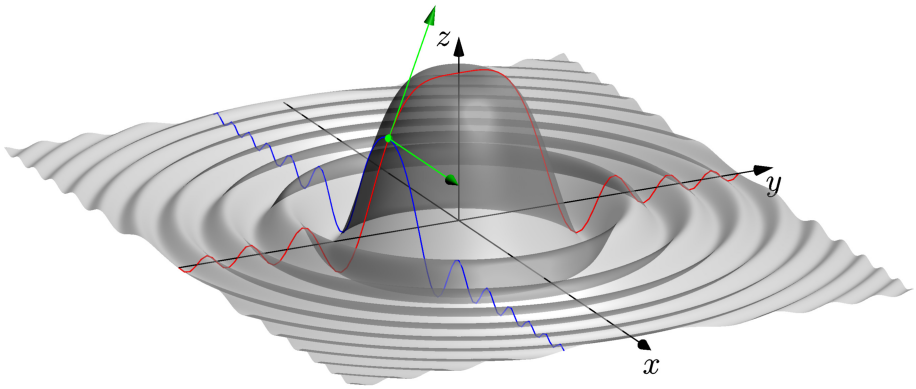



Par contre, l'utilisation de **meshpen** et en particulier de **thick**() est très coûteuse en ressource.
La figure précédente compile sur mon ordinateur avec **nmesh**=100 et l'utilisation de **meshpen=black** sans trop de problèmes (environ 15 secondes) mais avec **meshpen=black+thick**() cela dure plus d'une minute.

Avec **nmesh**=200 et **meshpen=black+thick**(), j'ai du redémarrer mon ordinateur sauvagement, plus rien ne réagissait après plusieurs minutes de compilation ...

Avec **nmesh**=350 et sans **meshpen**, on obtient la figure suivante, avec en prime les tangentes :




CODE 90

```
import graph3;
currentprojection=orthographic(10,-5,4);
size(12cm,0);

//boîte
pair a=(-2.5,-2);
pair b=(2.5,2);

//Definition de la surface

real f(pair z) {
real r=2pi*(z.x^2+z.y^2);
if (r!=0) return sin(r)/r;
else return 1;
}
surface s=surface(f,a,b,350,350,Spline);

//Intersection avec y=cy
real cy=-0.5;
real x1(real t) {return t;}
real y1(real t) {return cy;}
real z1(real t) {pair z=(t,cy); return f(z);}

path3 inter1=graph(x1,y1,z1,a.x,b.x);

//Intersection avec x=cx
real cx=-0;
real x2(real t) {return cx;}
real y2(real t) {return t;}
real z2(real t) {pair z=(cx,t); return f(z);}

path3 inter2=graph(x2,y2,z2,a.y,b.y);

//intersection des deux chemins
triple inter=(cx,cy,f((cx,cy)));

//Derivées partielles
real dx=0.1^15, dy=dx;

//Tangente dans la direction (Ox)
pair interplusdx=(inter.x+dx,inter.y);
pair interx=(inter.x,inter.y);
real partialx=(f(interplusdx)-f(interx))/dx;
path3 tgx=inter--(inter.x+1,inter.y,inter.z+partialx);
```

```
//Tangente dans la direction (Oy)
pair interplusdy=(inter.x,inter.y+dy);
pair intery=(inter.x,inter.y);
real partialy=(f(interplusdy)-f(intery))/dy;
path3 tgy=inter--(inter.x,inter.y+1/3,inter.z+partialy/3);


// Eléments graphiques
draw(s,lightgray+opacity(0.5));
draw(inter1,blue);
draw(inter2,red);
dot(inter, 3bp+green);
axes3("$x$", "$y$", "$z$",
        min=(a.x,a.y,0),
        max=(b.x+0.25,b.y+0.25,1.25),
        Arrow3);
draw(tgx,green,Arrow3);
draw(tgy,green,Arrow3);
```

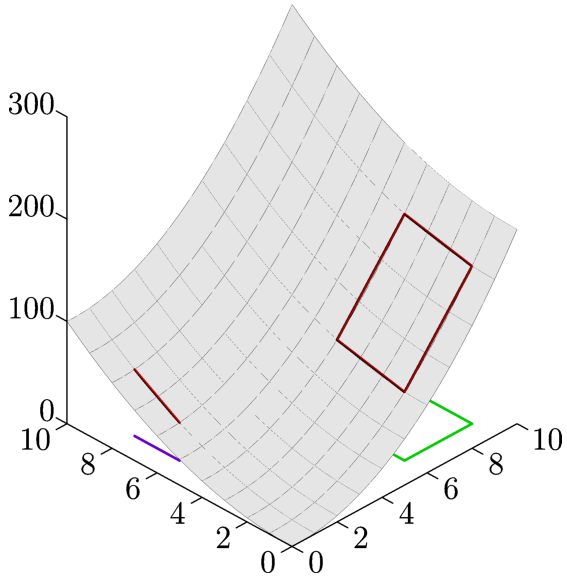
## 5. La routine **lift** et les lignes de niveau

La routine :

```
guide3[][] lift(real f(real x, real y), guide[][] g,
                interpolate3 join=operator --)
```

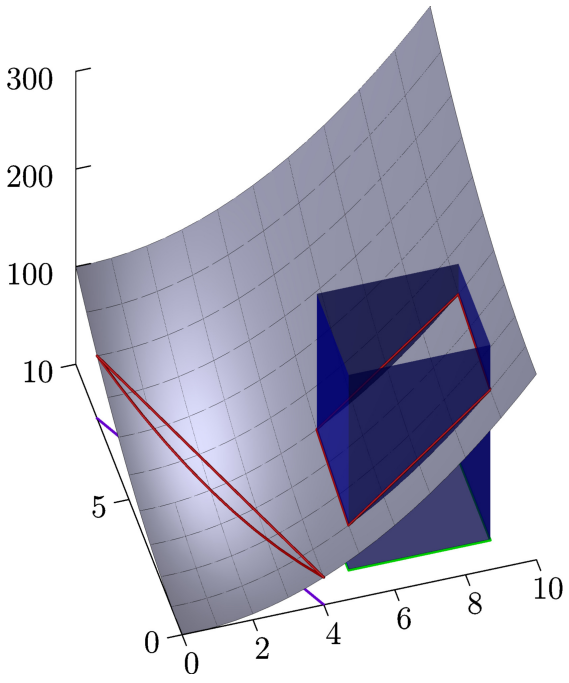
permet de passer d'un tableau bi-dimensionnel de guides 2D à un tableau bi-dimensionnel de guides 3D où la hauteur **z** au point de chaque élément du guide 2D est la valeur de f en cet élément.

Voici deux exemples signés Gaétan Marris [5] pour comprendre cette routine :




CODE 91

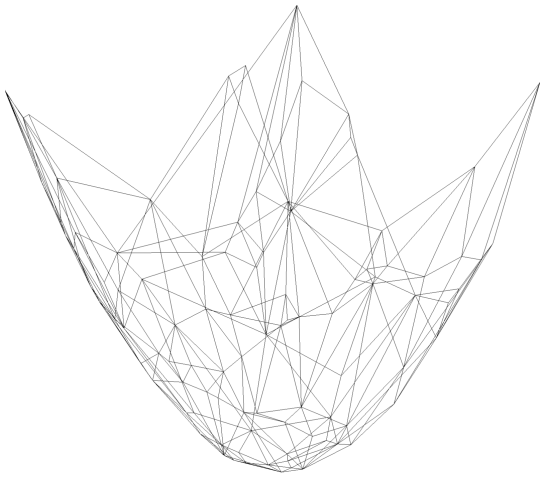
```
import graph3;
import contour;
size(7.5cm,0);
size3(7.5cm,IgnoreAspect);

currentprojection=orthographic(-25,-25,600);
limits((0,0,0),(10,10,300));

xaxis3(OutTicks(Step=2));
yaxis3(OutTicks(Step=2));
zaxis3(Bounds(Min,Max),
   InTicks(Step=100,Label(align=Y)));

real f(pair z) {
return 2z.x^2-z.x+z.y^2;
}
draw(surface(f,(0,0),(10,10),nx=10,Spline),
             lightgray,meshpen=black,nolight);

guide[][] tabgui={{(6,1)--(9,1)--(9,4)--(6,4)--cycle},
                  {(1,6)--(1,8)}};

draw(lift(f,tabgui),1bp+red);

path[] p1=tabgui[0], p2=tabgui[1];
draw(path3(p1),1bp+.8green);
draw(path3(p2),1bp+.8purple);
```




CODE 92

```
import graph3;
import contour;
size(7.5cm,0);
size3(7.5cm,IgnoreAspect);

currentprojection=orthographic(-6,-20,600);
currentlight=White;
limits((0,0,0),(10,10,300));
xaxis3(OutTicks(Step=2));
yaxis3(OutTicks(Step=5));
zaxis3(Bounds(Min,Max),
   InTicks(Step=100,Label(align=-X),NoZero));

real f(pair z) {
return 2z.x^2-z.x+z.y^2;
}
draw(surface(f,(0,0),(10,10),nx=10,Spline),
             lightgray,meshpen=black);

path p1=(5,1)--(9,1)--(9,4)--(5,4)--cycle,
 p2=(4,0)--(0,8),
p3;
for(int i=0; i<=10;++i) p3=p3..point(p2,i/10);

guide[][] tabgui={{p1,p2,p3}};

draw(lift(f,tabgui),1bp+red);

draw(path3(p1),1bp+.8green);
draw(path3(p2),1bp+.8purple);

draw(extrude(p1,axis=200Z),blue+opacity(.75));
```

L'exemple est un exemple d'Olivier Guibé [2] et illustre la triangulation d'une surface avec la routine **lift**.



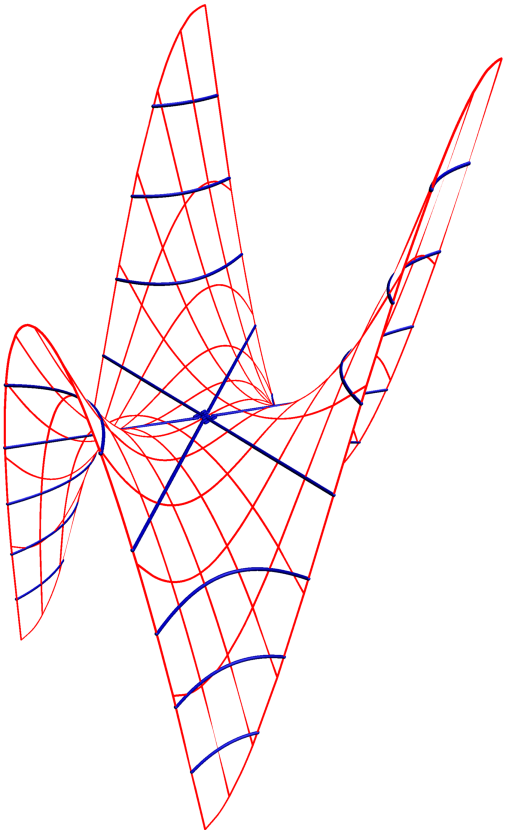
CODE 93

```
import graph3;
size(200);

int np=100;
pair[] points;
real r() {return 1.2*(rand()/randMax*2-1);}
for(int i=0; i < np; ++i)
points.push((r(),r()));

int[][] trn=triangulate(points);
guide[][] mong;
for(int i=0; i < trn.length; ++i) {
guide[] gg={points[trn[i][0]]--points[trn[i][1]]
           --points[trn[i][2]]--cycle};
mong.push(gg);
}

real f(pair z)
{
return z.x^2+z.y^2;
}

draw( lift(f,mong));
```

L'utilisation de **lift** combinée avec la routine **contour** permet dessiner des lignes de niveau sur une surface d'équation $z = f(x; y)$ :

```
guide[][] contour(real f(real, real), pair a, pair b,
                  real[] c, int nx=ngraph, int ny=nx,
                  interpolate join=operator --, int subsample=1);
```

Les entiers, **nx** et **ny** définissent la résolution. La résolution par défaut **ngraph**× **ngraph** (avec **ngraph** = 100 par défaut) peut être augmentée pour plus de précision.
L'opérateur par défaut d'interpolation est **Linear**. Le paramètre **Spline** peut rendre les contours plus lisses mais peut amener à un dépassement des capacités (overshooting).
Le paramètre **subsample** indique le nombre de points qui doit se trouver à l'intérieur d'un carré de côté 1. La valeur par défaut (**subsample** = 1) est satisfaisante dans la plupart des cas.



CODE 94

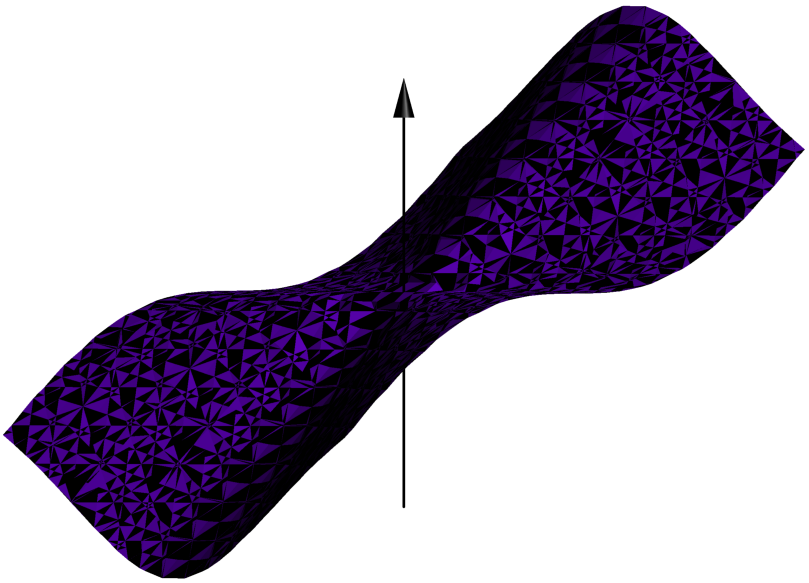
```
import graph3;
import contour;
size(7.5cm,0);
currentprojection=perspective(2,2,2);
limits((0,0,0),(1.5,1.5,2));
//Selle de singe

// Définition de la fonction
int n=3;
real a=1;
real f(pair z) {return ((z^n).x)/a^(n-1);}

// Lignes de niveau à tracer
real[] lignesniveaux={-1.5,-1,-0.5,0,0.5,1,1.5};

// On trace la surface
draw(surface(f,(-a,-a),(a,a),nx=10,Spline),
             white,meshpen=.8bp+red,
             nolight);

// On trace les lignes de niveau
draw(lift(f,contour(f,(-a,-a),(a,a),lignesniveaux)),
     1.5bp+blue);
```

## 6 . Surface définie implicitement

On peut également dessiner une surface définie par une équation implicite : $f(x;y;z)=0$ à l'aide du module **contour3**.

```
vertex[][] contour3(real f(real, real, real), triple a, triple b,
                    int nx=nmesh, int ny=nx, int nz=nx,
                    projection P=currentprojection)
```

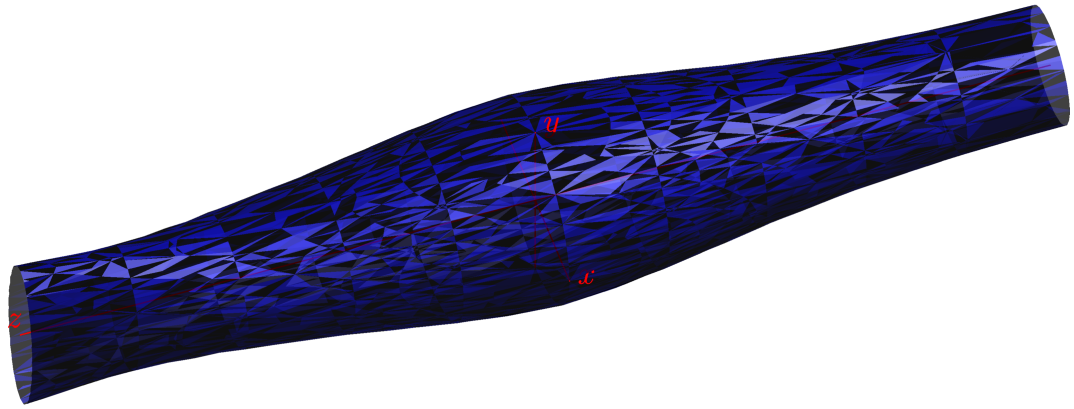
Premier exemple avec le parapluie de Cartan :



CODE 95

```
import graph3;
import contour3;
size(300,0);
currentprojection=orthographic((-75,-101,0.8),up=Z);

real f(real x, real y, real z) {return z*(x^2+y^2)-x^3;}

draw(surface(contour3(f,(-1,-1,-1),(1,1,1),15)),purple);
zaxis3(-0.75,0.75,1bp+black,Arrow3);
```

Deuxième exemple provenant de la documentation officielle [3] :




CODE 96

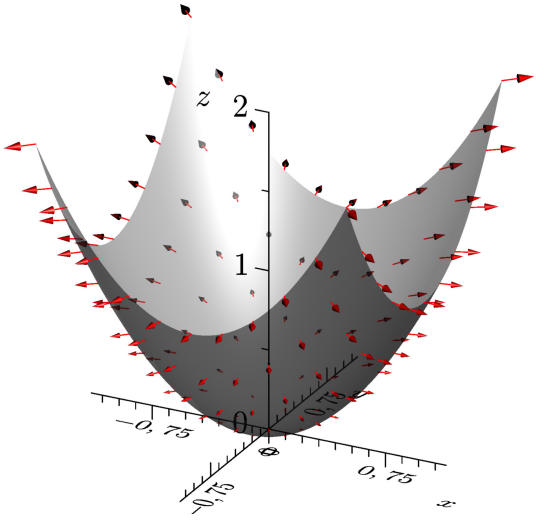
```
import graph3;
import contour3;

size(400,0);
currentprojection=orthographic((6,8,2),up=Y);

real a(real z) {return (z < 6) ? 1 : exp((abs(z)-6)/4);}
real b(real z) {return 1/a(z);}
real B(real z) {return 1-0.5cos(pi*z/10);}

real f(real x, real y, real z) {return 0.5B(z)*(a(z)*x^2+b(z)*y^2)-1;}

draw(surface(contour3(f,(-2,-2,-10),(2,2,10),10)), blue+opacity(0.75), render(merge=true));

xaxis3(Label("$x$",1),red);
yaxis3(Label("$y$",1),red);
zaxis3(Label("$z$",1),red);
```

## 7 . Champ de vecteurs

Un champ de vecteurs de $nu \times nv$ flèches peut être dessiner sur une surface paramétrique définie sur une boîte **box**(a,b) avec la routine :

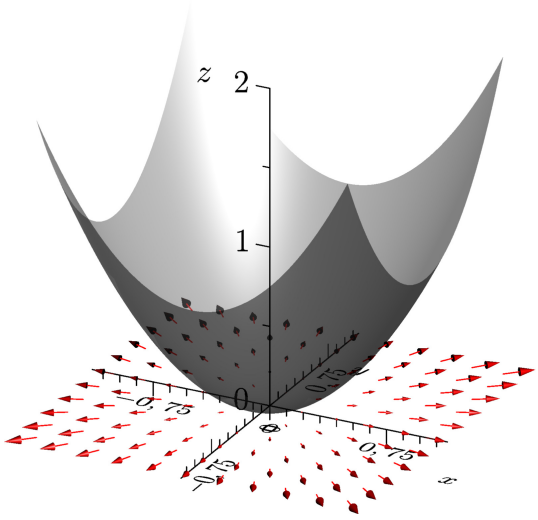
```
picture vectorfield(path3 vector(pair v), triple f(pair z), pair a, pair b,
                    int nu=nmesh, int nv=nu, bool truesize=false,
                    real maxlength=truesize ? 0 : maxlength(f,a,b,nu,nv),
                    bool cond(pair z)=null, pen p=currentpen,
                    arrowbar3 arrow=Arrow3, margin3 margin=PenMargin3)
```

Rappelons que **nmesh**=10 par défaut.
**vectorfield** est de type **picture**, on doit donc utiliser la syntaxe **add**(**vectorfield**()) pour dessiner le champ de vecteurs.
Voici un exemple de la documentation officielle [3] :




CODE 97

```
import graph3;
currentprojection=orthographic(1,-2,1);
currentlight=(1,-1,0.5);
size(7cm,0);

real f(pair z) {return abs(z)^2;}

path3 gradient(pair z) {
  static real dx=sqrtEpsilon, dy=dx;
  return O--((f(z+dx)-f(z-dx))/2dx,
             (f(z+I*dy)-f(z-I*dy))/2dy,
             0);
}

pair a=(-1,-1);
pair b=(1,1);

triple F(pair z) {return (z.x,z.y,f(z));}

add(vectorfield(gradient,F,a,b,red));

draw(surface(f,a,b,Spline),gray+opacity(0.5));

xaxis3(XY()*"$x$",OutTicks(XY()*Label));
yaxis3(XY()*"$y$",InTicks(YX()*Label));
zaxis3("$z$",OutTicks);
```




CODE 98

```
import graph3;
currentprojection=orthographic(1,-2,1);
currentlight=(1,-1,0.5);
size(7cm,0);

real f(pair z) {return abs(z)^2;}

path3 gradient(pair z) {
  static real dx=sqrtEpsilon, dy=dx;
  return O--((f(z+dx)-f(z-dx))/2dx,
             (f(z+I*dy)-f(z-I*dy))/2dy,
             0);
}

pair a=(-1,-1);
pair b=(1,1);

triple F(pair z) {return (z.x,z.y,0);}

add(vectorfield(gradient,F,a,b,red));

draw(surface(f,a,b,Spline),gray+opacity(0.5));

xaxis3(XY()*"$x$",OutTicks(XY()*Label));
yaxis3(XY()*"$y$",InTicks(YX()*Label));
zaxis3("$z$",OutTicks);
```

# VII – Module solids

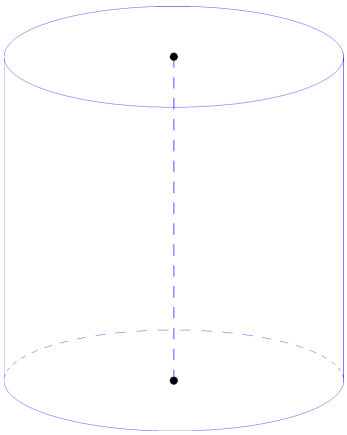
Cette extension définit une structure **revolution** que l'on peut utiliser pour dessiner des surfaces de révolution. Cette structure permet à son tour de définir les solides usuels que sont la sphère , le cône et le cylindre .

## 1 . Solides usuels

### 1. 1 . Cylindre de révolution

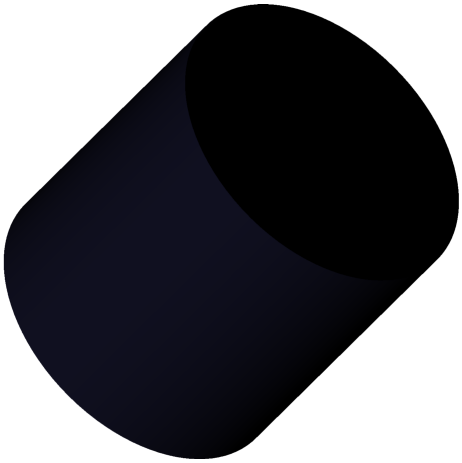
```
revolution cylinder(triple c=0, real r, real h, triple axis=Z)
{
  triple C=c+r*perp(axis);
  axis=h*unit(axis);
  return revolution(c,C--C+axis,axis);
}
```

dessine le cylindre de centre **c**, de rayon **r**, de hauteur **h** et d'axe $(Oz)$.



CODE 99

```
import solids;

currentprojection=orthographic(5,4,2);

size(6cm);

triple p0=(0,0,0);
draw(cylinder(p0,1,2),blue);
dot(p0);
dot((0,0,2));
draw(p0--(0,0,2),lightblue+dashed);

shipout(bbox(2mm,invisible));
```



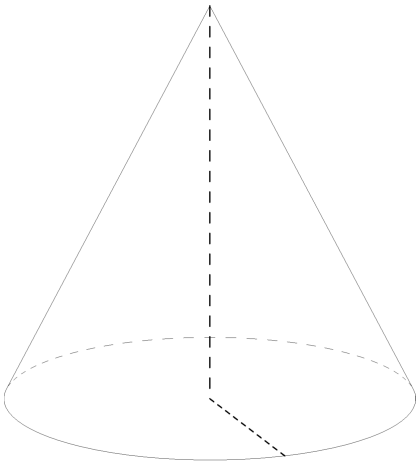
CODE 100

```
import solids;

currentprojection=orthographic(5,4,2);

size(6cm);

triple p0=(0,0,0);

revolution cyl=cylinder(p0,1,2,Y+Z);
draw(surface(cyl),orange);
```

### 1. 2 . Cône de révolution

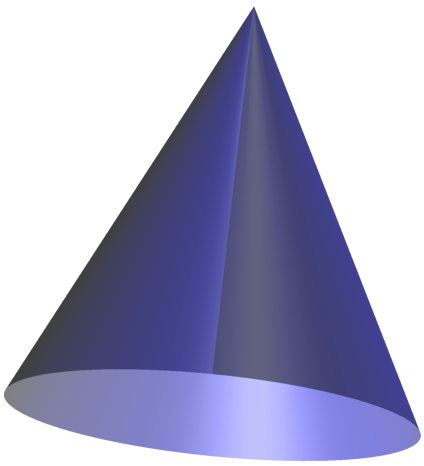
```
revolution cone(triple c=0, real r, real h, triple axis=Z, int n=nslice)
{
  axis=unit(axis);
  return revolution(c,approach(c+r*perp(axis)--c+h*axis,n),axis);
}
```

dessine le cône de centre **c**, de rayon **r**, de hauteur **h** et d'axe $(Oz)$.



CODE 101

```
import solids;

currentprojection=orthographic(5,4,2);

size(6cm);

triple pO=(0,0,0);
triple pA=(0,0,2);
triple pB=(cos(pi/3),sin(pi/3),0);

draw(cone(pO,r=1,h=2,n=1));
draw(pA--pO--pB,dashed);
```



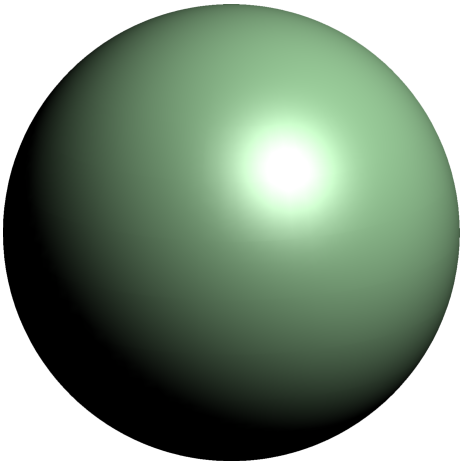
CODE 102

```
import solids;

size(6cm);

currentprojection=orthographic(10,0,2);

revolution CoRev=cone(O,1,2,
                      axis=0.2*Y+2*Z,
                      n=1);
draw(surface(CoRev),blue+opacity(0.5));
```

### 1. 3 . Sphère de révolution

```
revolution sphere(triple c=O, real r, int n=nslice)
{
  return revolution(c,Arc(c,r,180,0,0,0,Y,n),Z);
}
```

dessine la sphère de centre **c** et de rayon **r**.



CODE 103

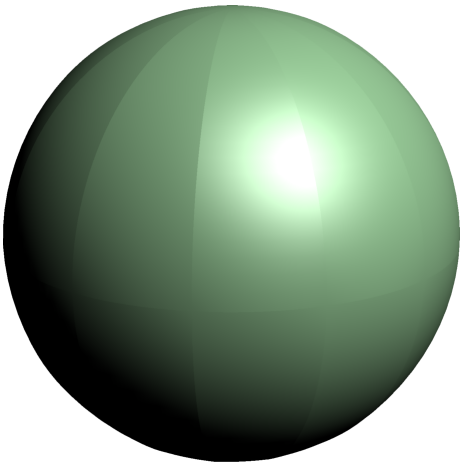
```
import solids;

size(6cm);
currentprojection=orthographic(5,4,2);

triple p0=(0,0,0);
real a=2;

revolution s=sphere(p0,a);

draw(surface(s),palegreen);
```

CODE 104

```
import solids;
size(6cm);
currentprojection=orthographic(2,2,1);

triple p0=(0,0,0);
triple pA=(0,0,2);

revolution c=cylinder(p0,1,2);
revolution s=sphere(pA,1);

draw(surface(s),orange);
draw(surface(c),yellow+opacity(0.75));
```

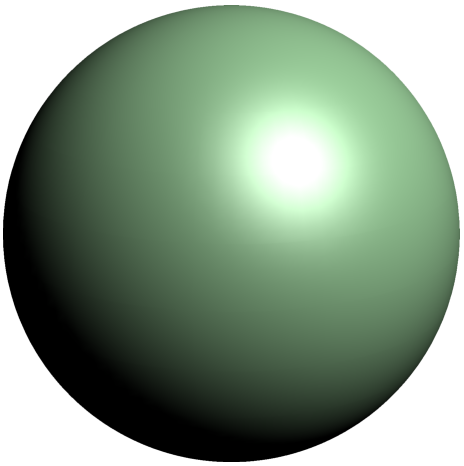
### 1. 4. Influence du paramètre **nslice**

La valeur du paramètre **nslice** peut être augmentée pour une meilleure résolution.
Dans le premier exemple où **n**=2, on voit bien que la sphère est réunion de deux demi-sphères, elles même découpées en plusieurs parties. On remarque aussi les approximations au niveau du « recollement ».
Dans le deuxième exemple, la valeur par défaut est utilisée. La surface apparait plus « lisse ».
Par défaut, **nslice**=12.



CODE 105

```
import solids;
size(6cm);

revolution s=sphere(0,1,n=2);
draw(surface(s),palegreen);
```



CODE 106

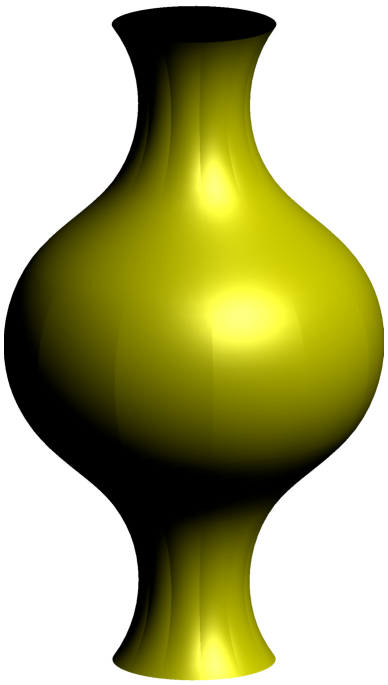
```
import solids;
size(6cm);

revolution s=sphere(0,1);
draw(surface(s),palegreen);
```

## 2. Créer son propre solide

D'une manière plus générale, la structure **revolution** peut s'employer ainsi :

```
revolution(triple c, path3 g,triple axis,real angle1,real angle2);
```

où le **path3 g** est un chemin plan qui subit une rotation autour du point **triple c** perpendiculairement à l'axe **triple** axe entre les angles **angle1** et **angle2**.

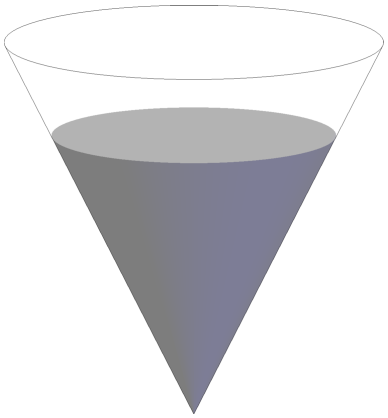


**CODE 107**

```
import solids;

currentprojection = perspective(10,100,25);
unitsize(2.5cm);

triple p0=(0,0,0);

path3 gene=(0,0.5,-2)..(0,0.5,-1)..
                (0,1,-0.5)..(0,1,0.5)
                ..(0,0.5,1)..(0,0.5,2);
revolution sur=revolution(p0,gene,Z,0,360);
draw(surface(sur),yellow);
```

**CODE 108**

```
import solids;

currentprojection = perspective(10,100,25);
unitsize(2.5cm);

triple p0=(0,0,0);

path3 gene=(0,0.5,-2)..(0,0.5,-1)..
                (0,1,-0.5)..(0,1,0.5)
                ..(0,0.5,1)..(0,0.5,2);
revolution sur=revolution(p0,gene,Z,120,370);
draw(surface(sur),yellow);
```



**CODE 109**

```
import solids;
currentprojection=orthographic(10,0,2);

size(5cm,0);

path3 gene = (0,0,0)--(0,1,2);
revolution CoRev=revolution((0,0,0),
                gene,axis=Z,0,360);
draw(CoRev);

path3 gene2=(0,0,0)--(0,0.75,1.5);
revolution eau=revolution((0,0,0),
                gene2,axis=Z,0,360);
draw(surface(eau),blue+opacity(0.3));
```

### 3 . la routine **draw** de **solids.asy** : squelette d'une surface de révolution

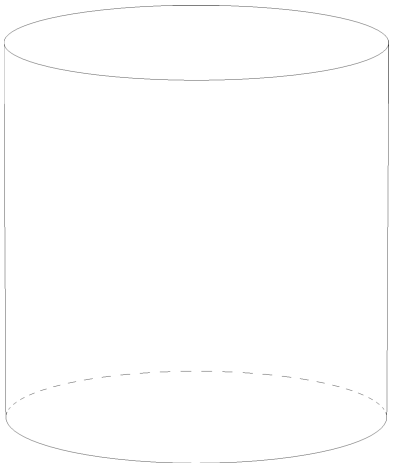
La routine **draw** de **solids.asy** permet de tracer le squelette d'une surface de révolution ainsi que des coupes transversales (perpendiculaires à l'axe de révolution) et **la** section longitudinale (parallèle à l'axe de révolution) :

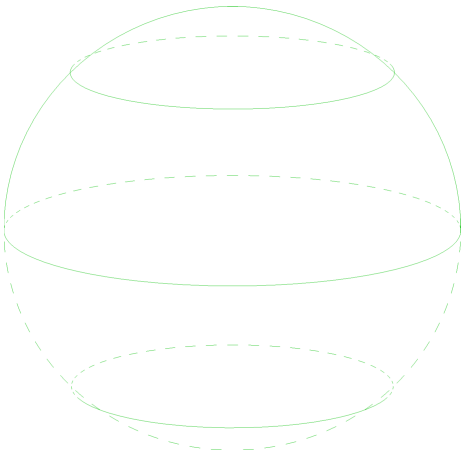
```
void draw(picture pic=currentpicture, revolution r, int m=0, int n=nslice,
          pen frontpen=currentpen, pen backpen=frontpen,
          pen longitudinalpen=frontpen, pen longitudinalbackpen=backpen,
          light light=currentlight, string name="",
          render render=defaultrender, projection P=currentprojection)
```

où $m$ donnera le nombre de section de type **transverse** où les arc de cercle sont approximés avec $n$ points.
On peut définir les différents pinceaux **frontpen**, **backpen** pour les section de type **transverse** et **longitudinalpen** et **longitudinalbackpen** pour **la** section de type **longitudinal**.

CODE 110

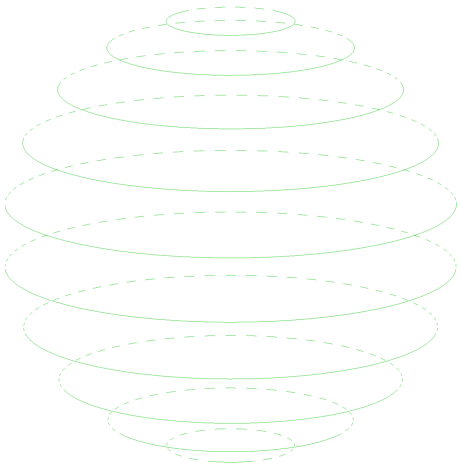
```
import solids;
currentprojection = perspective(10,100,25);
size(6cm);

real a=2.5;
revolution r = cylinder(0, a, 2*a);
draw(r);
```





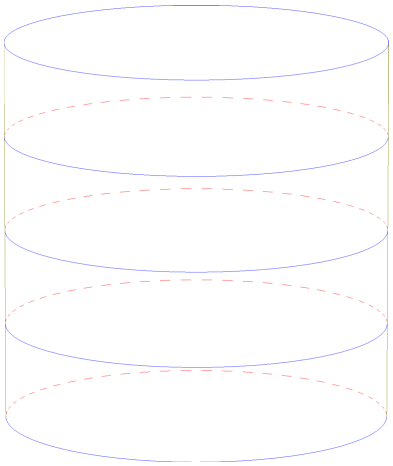
CODE 111

```
import solids;
currentprojection = perspective(10,100,25);
size(6cm);

real a = 2.5;
revolution r = sphere(0, a);
draw(r,5,heavygreen);
```



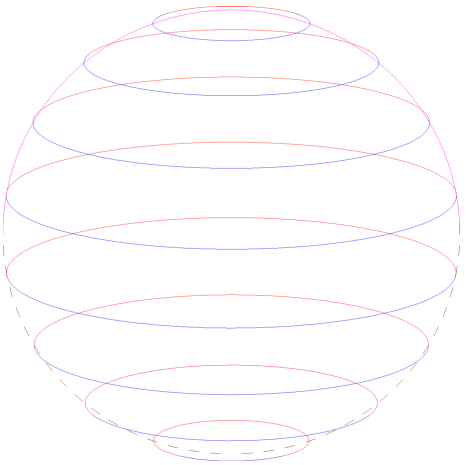
CODE 112

```
import solids;
currentprojection = perspective(10,100,25);
size(6cm);

real a = 2.5;
revolution r = sphere(0, a);
draw(r,12,
     frontpen=heavygreen,
     longitudinalpen=nullpen);
```



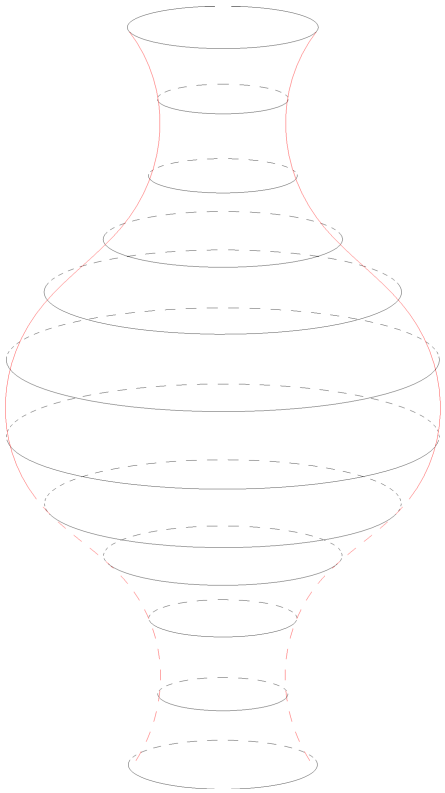
CODE 113

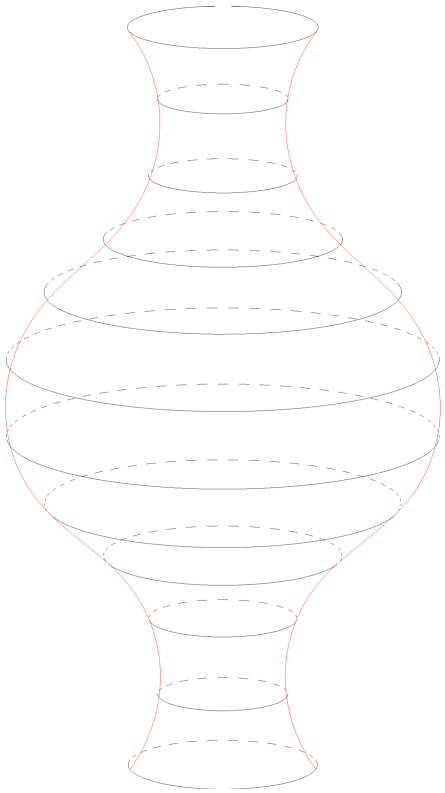
```
import solids;
currentprojection = perspective(10,100,25);
size(6cm);

real a = 2.5;
revolution r = cylinder(0, a, 2*a);
draw(r,5,
     frontpen=blue,
     backpen=red,
     longitudinalpen=olive);
```



CODE 114

```
import solids;
currentprojection = perspective(10,100,25);
size(6cm);

real a = 2.5;
revolution r = sphere(0, a);
draw(r,10,
     frontpen=blue,
     backpen=solid+red,
     longitudinalpen=magenta,
     longitudinalbackpen=brown);
```





CODE 115

```
import solids; import solids;
currentprojection = perspective(10,100,25);
unitsize(2.5cm);

path3 gene=(0,0.5,-2)..(0,0.5,-1)..
                (0,1,-0.5)..(0,1,0.5)
                ..(0,0.5,1)..(0,0.5,2);
revolution sur=revolution(0,gene,Z,0,360);
//draw(surface(sur),yellow);
draw(sur,12,
     longitudinalpen=red,
     longitudinalbackpen=red);
```

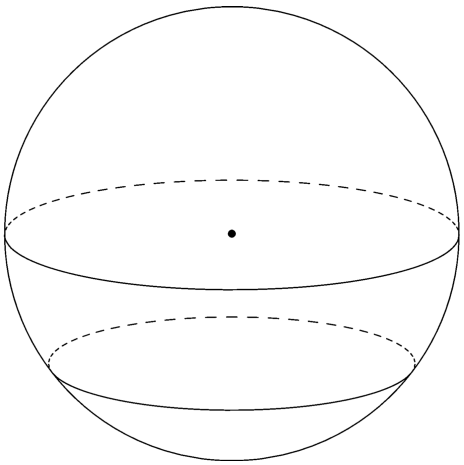
CODE 116

```
import solids; import solids;
currentprojection = perspective(10,100,25);
unitsize(2.5cm);

path3 gene=(0,0.5,-2)..(0,0.5,-1)..
                (0,1,-0.5)..(0,1,0.5)
                ..(0,0.5,1)..(0,0.5,2);
revolution sur=revolution(0,gene,Z,0,360);
//draw(surface(sur),yellow);
draw(sur,12,
     longitudinalpen=red,
     longitudinalbackpen=red+solid);
```

Une dernière routine définie par **solids.asy** est la structure **skeleton** qui permet de dessiner des sections des ces solides par des plans :

```
struct skeleton {
  struct curve {
    path3[] front;
    path3[] back;
  }
  // transverse skeleton (perpendicular to axis of revolution)
  curve transverse;
  // longitudinal skeleton (parallel to axis of revolution)
  curve longitudinal;
}
```

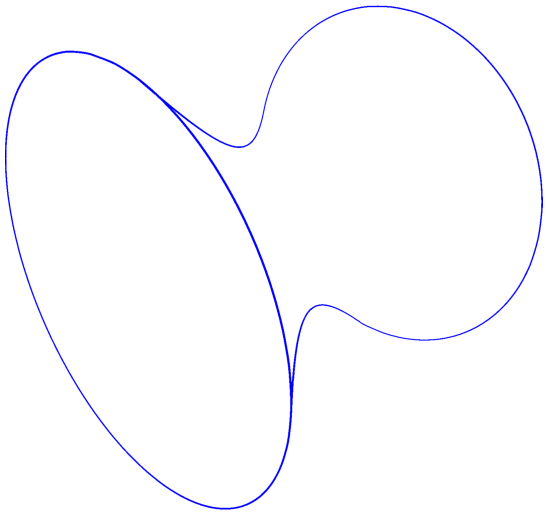
Elle permettra de tracer le squelette d'une surface de révolution et des sections de type **transverse** c'est-à-dire perpendiculaire à l'axe du solide de révolution.
**longitudinal** représente une section parallèle à l'axe du solide de révolution.



CODE 117

```
import solids;
currentprojection = perspective(10,100,25);

size(6cm);

real a = 2.5;

triple p0 = (0,0,0);
dot(p0);

revolution r = sphere(p0, a);
//On utilise la commande silhouette pour obtenir le contour de la sphère
draw(r.silhouette());
skeleton s;

//g est une génératrice ; On définit les sections
r.transverse(s,reltime(r.g,0.5));
r.transverse(s,reltime(r.g,0.3));

//On trace ce qui est visible
draw(s.transverse.front, solid+black);
//On trace ce qui est caché
draw(s.transverse.back, dashed+black);
```

**Remarque :**
Dans cette exemple, on utilise la commande **silhouette** qui projette en deux dimensions une surface. Si vous compiler le code précédent, vous obtiendrait donc un message d'avertissement : « silhouette routine is intended only for 2d projections ».
Voici un exemple de la documentation officielle [3] d'utilisation de **silhouette**.



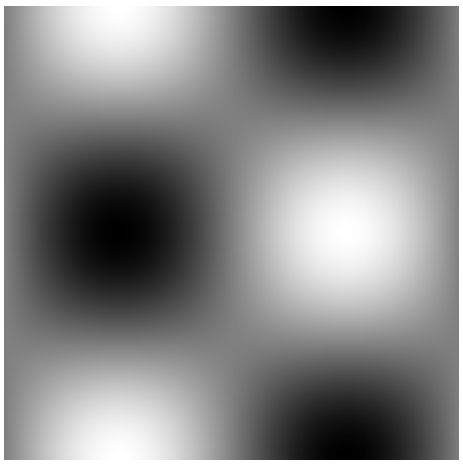
CODE 118

```
import solids;
size(200);

currentprojection=perspective(4,4,3);
revolution hyperboloid=revolution(new real(real x) {return sqrt(1+x*x);},
                                  -2,2,20,operator..,X);
draw(hyperboloid.silhouette(64),blue);
```

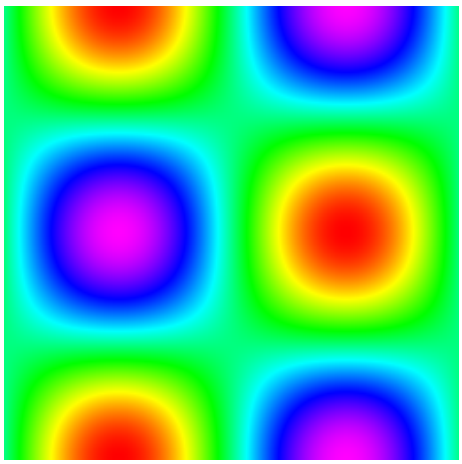
# VIII – What else ?

## 1 . La couleur avec le module palette.asy

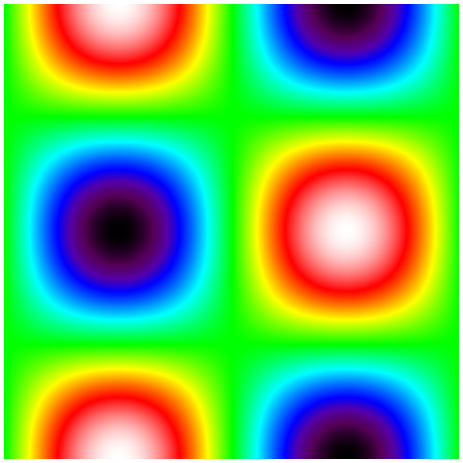
Dans le module **palette.asy** sont définies :



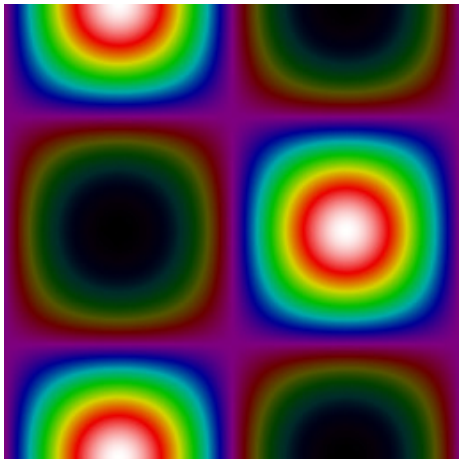
```
pen[] Grayscale(int NColors=256)
```



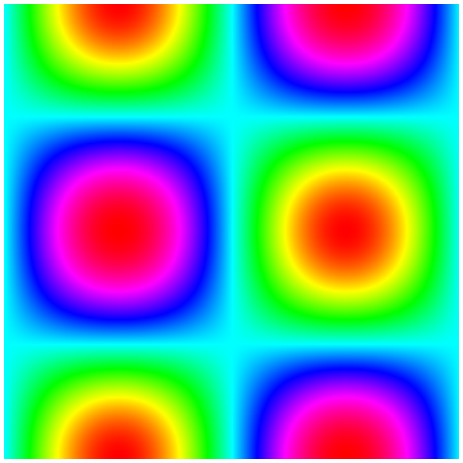
```
pen[] Rainbow(int NColors=32766)
```



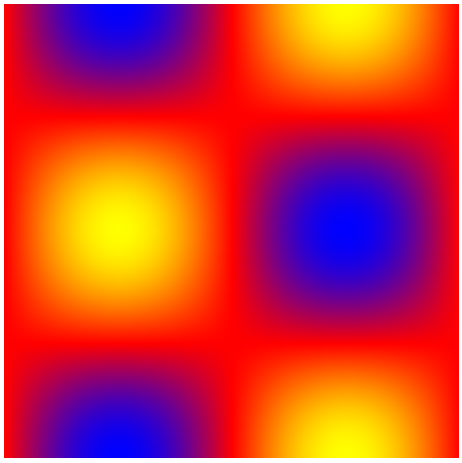
```
pen[] BWRainbow(int NColors=32761)
```



```
pen[] BWRainbow2(int NColors=32761)
```



```
pen[] Wheel(int NColors=32766)
```



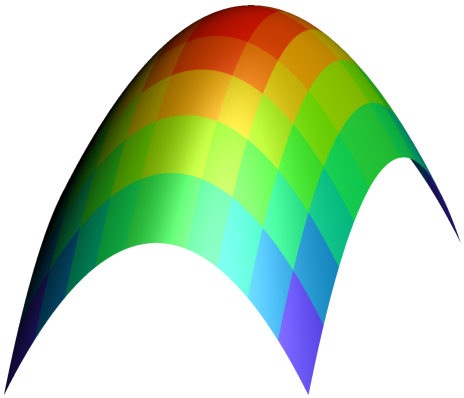
```
pen[] Gradient(int NColors=256 ... pen[] p)
```

utilisée ici avec **red**, **yellow** et **blue**.

Le but de ce paragraphe n'étant pas de présenter de manière exhaustive le module **palette.asy**, on se contentera, sur quelques exemples, de voir comment colorer des surfaces définies par **z**=f(**x**;**y**).
Dans la pratique, le plus simple est peut-être de définir une liste de pinceaux **après** avoir défini notre surface **s** :

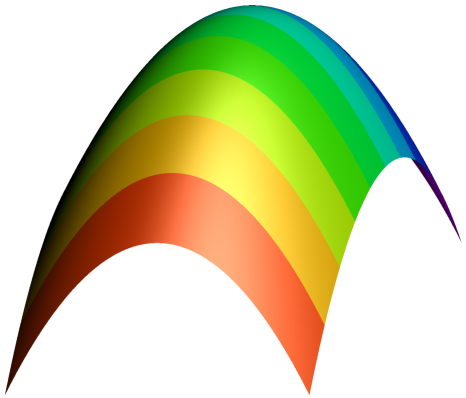
```
pen[] pens=mean(palette(s.map(triple direction),pen[] palette));
```

où **s.map(triple** direction) détermine la « direction » dans laquelle va être employée la liste des pinceaux comme le montrent les exemples :



CODE 119

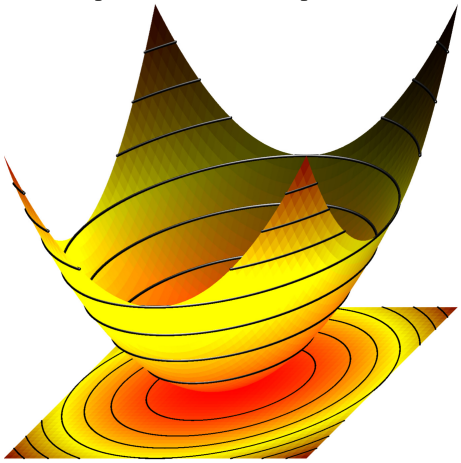
```
import graph3;
import palette;
currentprojection=obliqueX;
size(6cm);

real f(pair z) {return -z.x^2-z.y^2;}
surface s=surface(f,(-1,-1),(1,1), Spline);

pen[] pens=mean(palette(s.map(zpart), Rainbow()));

draw(s,pens);
```



CODE 120

```
import graph3;
import palette;
currentprojection=obliqueX;
size(6cm);

real f(pair z) {return -z.x^2-z.y^2;}
surface s=surface(f,(-1,-1),(1,1), Spline);

pen[] pens=mean(palette(s.map(xpart), Rainbow()));
draw(s,pens);
```

L'avantage de définir une liste de pinceaux est de pouvoir l'utiliser plusieurs fois comme pour ce paraboloïde :



CODE 121

```
import graph3;
import palette;
import contour;
currentprojection=obliqueX;
currentlight=light((1,1,3),(0,0,1));
size(6cm);

real f(pair z) {return z.x^2+z.y^2;}
surface s=surface(f,(-1,-1),(1,1), nx=50);

// On définit la palette
pen[] pens=mean(palette(s.map(zpart), Gradient(red,yellow,brown)));

// On trace la surface
draw(s,pens);

// On trace les lignes de niveau
real [] lignesdeniveau={0.25,0.5,0.75,1,1.25,1.5,1.75};
draw(lift(f,contour(f,(-1,-1),(1,1),lignesdeniveau)),1bp+black);

// On projette la surface
path3 pl=plane(X,Y,0);
draw(planeproject(pl)*s,pens , nolight);

// On projette les lignes de niveau
guide[][] pl=contour(f,(-1,-1),(1,1),lignesdeniveau);
for (int i=0; i < pl.length; ++i)
  for (int j=0; j < pl[i].length; ++j)
    draw(path3(pl[i][j]));
```

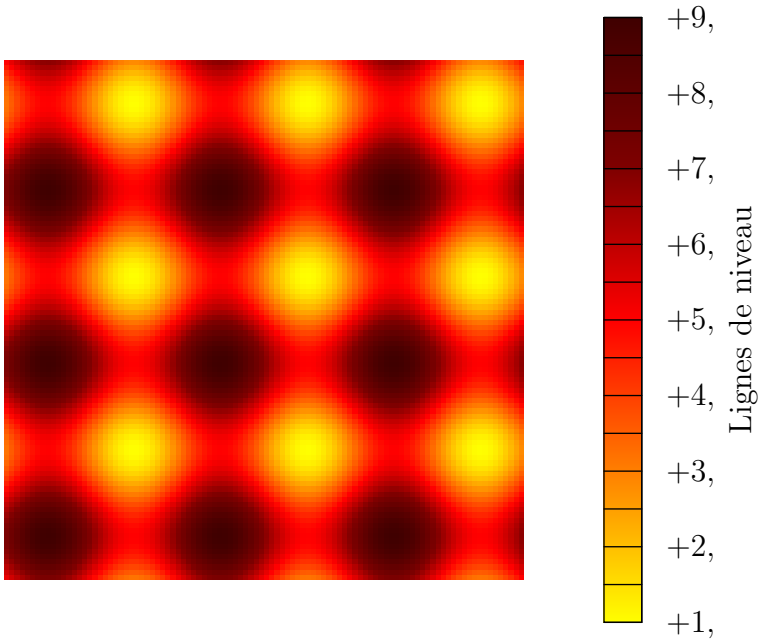
On peut aussi, grâce au module **palette.asy**, utiliser la routine **bounds** pour créer, à partir d'une fonction f(**x**;**y**) et d'une **palette**, une projection de la surface sur le plan. Les variations de couleurs dépendant de l'altitude.

```
bounds image(picture pic=currentpicture, real f(real,real),
             range range=Full, pair initial, pair final,
             int nx=ngraph, int ny=nx, pen[] palette, bool antialias=false)
```

Pour plus lisibilité, on peut aussi ajouter une barre de graduation grâce à :

```
void palette(picture pic=currentpicture, Label L="", bounds bounds,
             pair initial, pair final, axis axis=Right, pen[] palette,
             pen p=currentpen, paletteticks ticks=PaletteTicks,
             bool copy=true, bool antialias=false);
```

L'orientation de la barre de graduation se règle avec **axis** qui peut prendre les valeurs **Right**, **Left**, **Top** ou **Bottom**. La barre est dessinée dans le rectangle défini par initial et final.
Pour plus de précision, on se reportera à la documentation officielle [3].




CODE 122

```
import graph3;
import palette;
currentprojection=orthographic(65,-18,30);
currentlight=light(0,0,10);
size(10cm);
real a=2, b=1;

real g(real x, real y) {return a*(sin((x)/b)+sin((y/b)))+5;}
pen [] p=Gradient(yellow, orange, red, brown, darkbrown);

picture bar;
bounds range=image(g,Automatic,(0,0),(6pi,6pi),p);
palette(bar,"Lignes de niveau",range,(0,0),(0.5cm,8cm),Right,p,
PaletteTicks("$%+#.1g$"));
add(bar.fit(),point(E),30E);
```

## 2 . Complément sur les labels

### 2. 1 . Écrire sur une surface

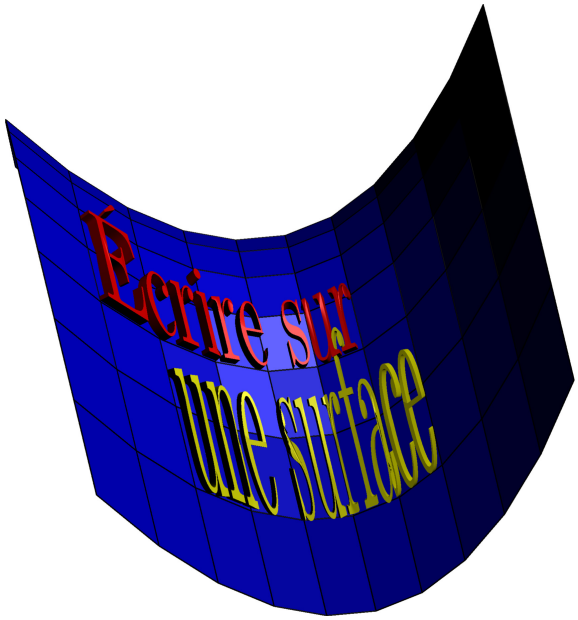
On utilise la routine :

```
surface surface(Label L,
                surface s,
                real uoffset,
                real voffset,
                real height=0,
                bool bottom=true,
                bool top=true)
```

Le positionnement du texte sur la surface est défini par les paramètres **uoffset** et **voffset** et par rapport à **ny** et **nx** (qui servent à définir la surface).

Le paramètre **height** définit la hauteur du texte qui est dessiné en relief sur la surface.

Regardez dans l'exemple suivant le rôle des paramètres **uoffset** et **voffset**. De plus, on se rend rapidement compte que la taille du texte doit être modifiée pour être lisible. Essayez sans le **xscale**(2)∗**scale**(0.25) ...




CODE 123

```
import graph3;
currentprojection=orthographic(
                        camera=(-0.283273333856929,-36.8201675308683,34.9026601261818),
                        up=(0.00248404205232355,0.00709396484845398,0.00750385074735482),
                        target=(0,0,-8.88178419700125e-16),
                        zoom=1);

size(7.5cm,0);
real f(pair z) {return z.x^2-z.y^2;}

surface s=surface(f, (-1,-1), (1,1), nx=10);
draw(s,blue, meshpen=black+thick());

draw(surface(xscale(2)*scale(0.25)*"\'Ecrire sur", s, uoffset=1, voffset=3, height=0.05), red);
draw(surface(xscale(2)*scale(0.25)*"une surface", s, uoffset=2, voffset=1, height=0.05), Yellow);
```

D'une manière générale, on peut tracer les **meshline** pour positionner le texte. Libre à vous de les faire apparaître ou non in fine.

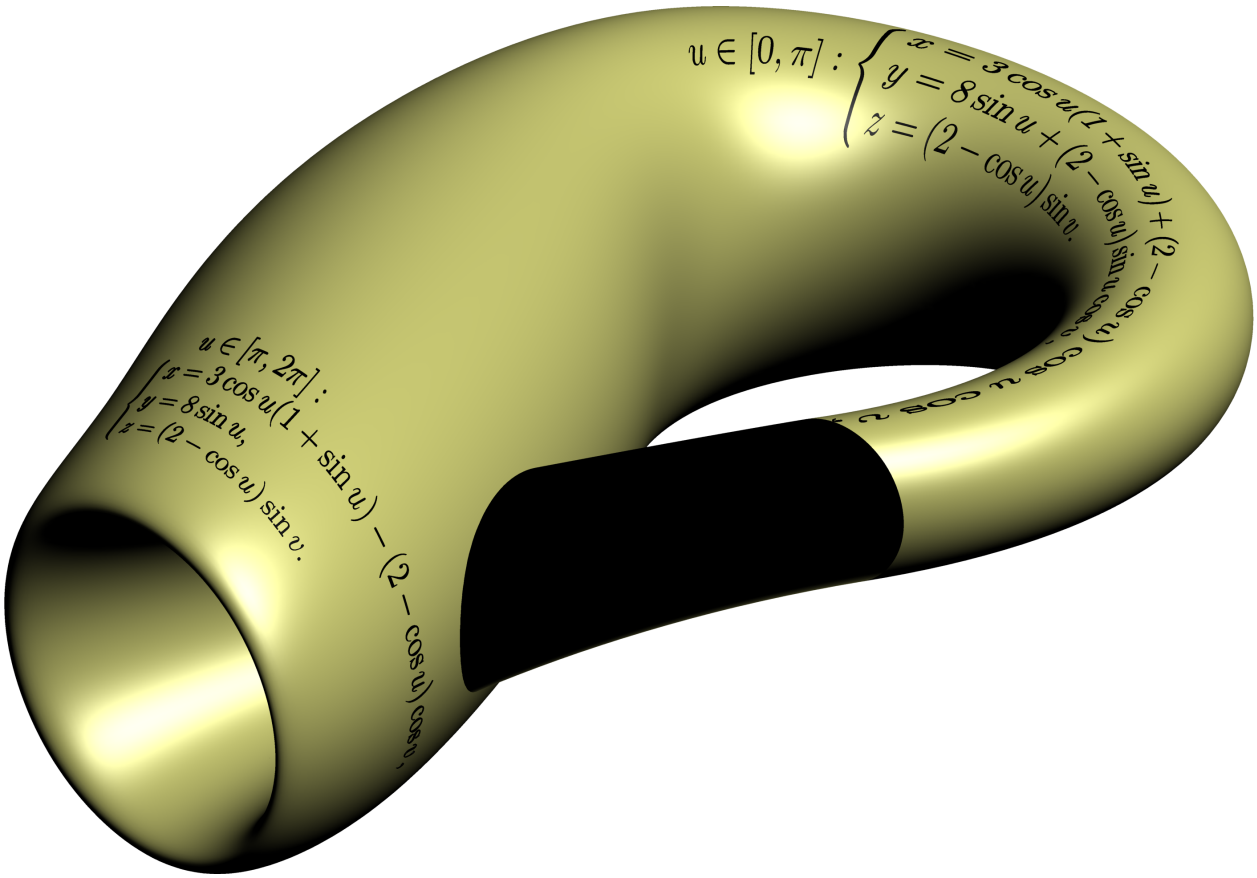

CODE 124

```
import solids;
currentprojection=orthographic(1,5,2);

size(7.5cm,0);
revolution sp=sphere(0,1);

surface s=surface(sp);
draw(s,blue, meshpen=black+thick());

draw(surface(xscale(1.5)*scale(0.15)
                        *"\'Ecrire sur",
             s,
             uoffset=1,
             voffset=8,
             height=0.05), red);
draw(surface(xscale(1.5)*scale(0.15)
                        *"une surface",
             s,
             uoffset=2,
             voffset=6,
             height=0.05), Yellow);
```


CODE 125

```
import solids;
currentprojection=orthographic(1,5,2);

size(7.5cm,0);
revolution sp=sphere(0,1);

surface s=surface(sp);
draw(s,blue);

draw(surface(xscale(1.5)*scale(0.15)
                        *"\'Ecrire sur",
             s,
             uoffset=1,
             voffset=8,
             height=0.05), red);
draw(surface(xscale(1.5)*scale(0.15)
                        *"une surface",
             s,
             uoffset=2,
             voffset=6,
             height=0.05), Yellow);
```

Toute la gamme des formules et des caractères LaTeX est bien sur utilisable comme le montre le (bel) exemple (simplifié) de la documentation officielle [3].




CODE 126

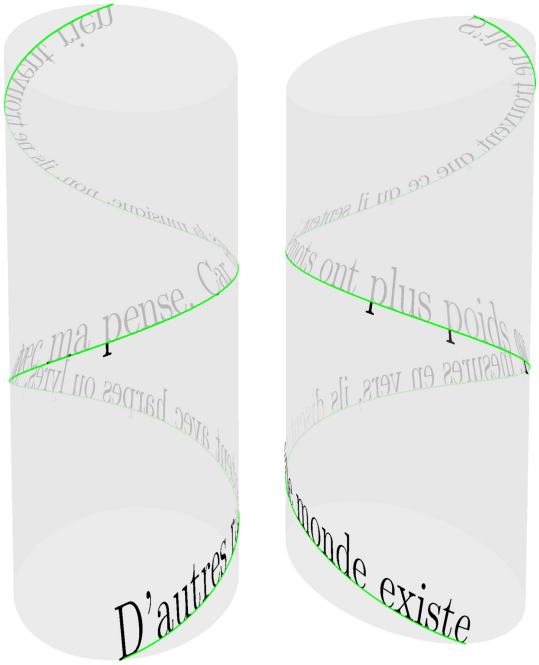
```
import graph3;
size(469pt);
currentprojection=perspective(
                        camera=(25.0851928432063,-30.3337528952473,19.3728775115443),
                        up=Z,
                        target=(-0.590622314050054,0.692357205025578,-0.627122488455679),
                        zoom=1,
                        autoadjust=false);

triple f(pair t) {
  real u=t.x;
  real v=t.y;
  real r=2-cos(u);
  real x=3*cos(u)*(1+sin(u))+r*cos(v)*(u < pi ? cos(u) : -1);
  real y=8*sin(u)+(u < pi ? r*sin(u)*cos(v) : 0);
  real z=r*sin(v);
  return (x,y,z);
}

surface s=surface(f,(0,0),(2pi,2pi),8,8,Spline);
draw(s,lightolive+white);

string lo="$\displaystyle u\in[0,\pi]: \cases{x=3\cos u(1+\sin u)+(2-\cos u)\cos u\cos v,\cr
                                              y=8\sin u+(2-\cos u)\sin u\cos v,\cr
                                              z=(2-\cos u)\sin v.\cr}$";

string hi="$\displaystyle u\in[\pi,2\pi]:\\\cases{x=3\cos u(1+\sin u)-(2-\cos u)\cos v,\cr
                                                  y=8\sin u,\cr
                                                  z=(2-\cos u)\sin v.\cr}$";

real h=0.0125;

draw(surface(xscale(-0.38)*yscale(-0.18)*lo,s,0,1.7,h));
draw(surface(xscale(0.26)*yscale(0.1)*rotate(90)*hi,s,4.9,1.4,h));
```

### 2. 2 . Écrire le long d'un chemin

le module **labelpath3.asy** est le pendant de **labelpath** en trois dimensions. Mais contrairement à son cousin en 2D, **labelpath3** ne nécessite pas l'utilisation de l'extension **PSTricks** et on peut donc compiler avec le moteur pdflatex.

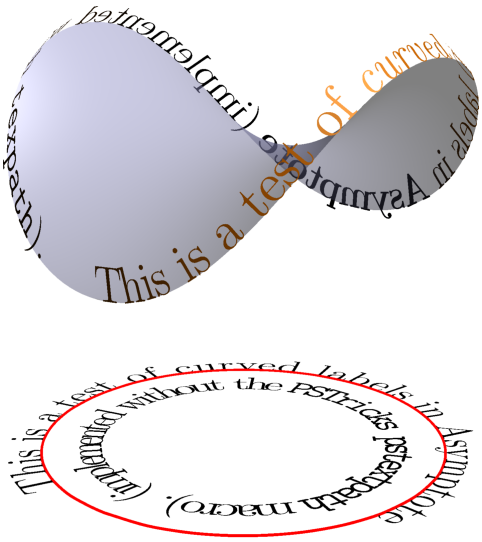
```
surface labelpath(string s, path3 p, real angle=90, triple optional=0)
```

On se contentera d'un poème et d'un exemple de la documentation officielle :




CODE 127

```
import graph3;
import labelpath3;
currentprojection=perspective(4,-1,1.25);
size(7cm,9cm,IgnoreAspect);

real x(real t) {return cos(2pi*t);}
real y(real t) {return sin(2pi*t);}
real y2(real t) {return -sin(2pi*t);}
real z(real t) {return t;}

path3 p=graph(x,y,z,0,2,operator ..);
draw(shift(0,-1.25,0)*zscale3(2)*unitcylinder,
                lightgray+opacity(0.8),
                nolight);
draw(shift(0,-1.25,0)*p, green);
string debut="D'autres racontent avec harpes ou lyres,
              Moi avec ma pensée.
              Car à travers la musique,
                              non, ils ne trouvent rien";
draw(labelpath(debut, shift(0,-1.25,0)*p, 100));

path3 p2=graph(x,y2,z,0,2,operator ..);
draw(shift(0,1.25,0)*zscale3(2)*unitcylinder,
                lightgray+opacity(0.8),
                nolight);
draw(shift(0,1.25,0)*p2, green);
string fin="S'ils ne trouvent que ce qu il sentent.
            Les mots ont plus poids quand,
            mesures en vers, ils disent que
                                  le monde existe";
draw(labelpath(fin, shift(0,1.25,0)*reverse(p2), 180));
```




CODE 128

```
size(200);
import labelpath3;

path3 g=(1,0,0)..(0,1,1)..(-1,0,0)..(0,-1,1)..cycle;
path3 g2=shift(-Z)*reverse(unitcircle3);

string txt1="\hbox{This is a test of \emph{curved} 3D labels in
\textbf{Asymptote} (implemented with {\tt texpath}).}";

string txt2="This is a test of curved labels in Asymptote\\(implemented
without the {\tt PSTricks pstextpath} macro).";

draw(surface(g),paleblue+opacity(0.5));
draw(labelpath(txt1,subpath(g,0,reltime(g,0.95)),angle=-90),orange);

draw(g2,1bp+red);
draw(labelpath(txt2,subpath(g2,0,3.9),angle=180,optional=rotate(-70,X)*Z));
```

## 3. Tube

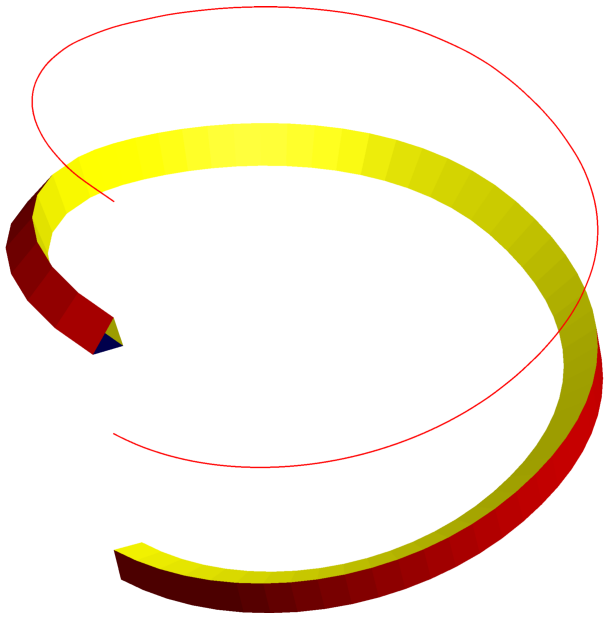
La surface **tube** définie dans **tube.asy** par :

```
surface tube(path3 g, coloredpath section,
             transform T(real)=new transform(real t) {return identity();},
             real corner=1, real relstep=0);
```

permet de dessiner une surface le long d'un chemin **g** dont la section est définie par coloredpath **section** :

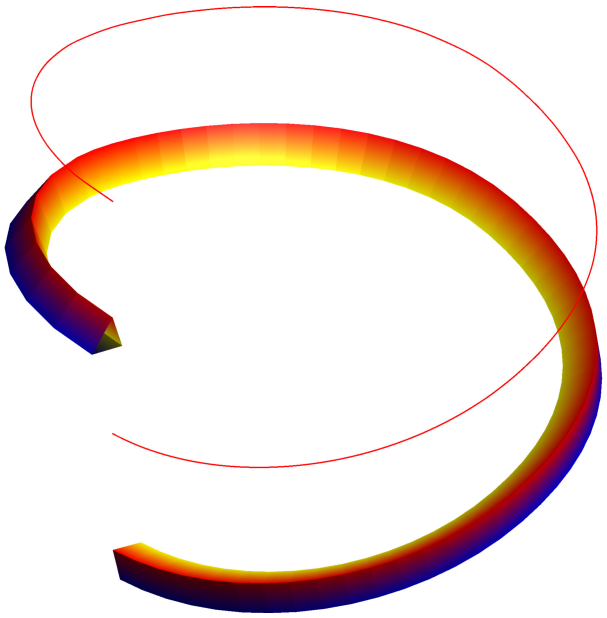
```
struct coloredpath
{
    path p;
    pen[] pens(real t);
    int colortype=coloredSegments;
}
```

où **p** est un chemin défini dans un plan et définira la section, **pen**[] pens(**real** t) retourne un tableau (cyclique) de couleur utilisées pour colorer le tube au point **relpoint**(**g**,t).
Si **colortype** =**coloredSegments** les différents segments de la section seront colorés avec le pinceau retourné par pens(t) alors que si **colortype** = **coloredNodes**, la coloration sera plus subtiles.



CODE 129

```
import graph3;
import tube;
import palette;
currentprojection=orthographic(2,2,2);
size(8cm);

path3 g=(1,0,0)..(0,1,0.25)..(-1,0,.5)..
        (0,-1,0.75)..(1,0,1);
draw(shift(0,0,0.5)*g,red);
pen[] pens;
      pens[0]=red;
      pens[1]=blue;
      pens[2]=yellow;

pair pA,pB,pC;
     pA=(0,0);
     pB=(1,0);
     pC=rotate(60,pA)*pB;
path sec=pA--pB--pC--cycle;
path section=scale(0.15)*sec;

coloredpath colorsec=coloredpath(section,
                      pens,
                      colortype=coloredSegments);
draw(tube(g,colorsec));
```



CODE 130

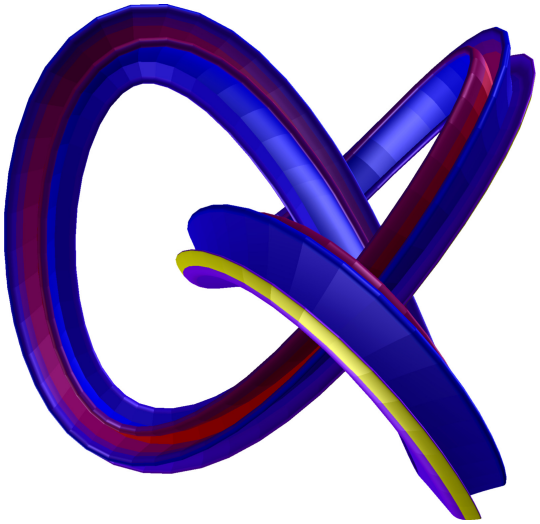
```
import graph3;
import tube;
import palette;
currentprojection=orthographic(2,2,2);
size(8cm);

path3 g=(1,0,0)..(0,1,0.25)..(-1,0,.5)..
        (0,-1,0.75)..(1,0,1);
draw(shift(0,0,0.5)*g,red);
pen[] pens;
      pens[0]=red;
      pens[1]=blue;
      pens[2]=yellow;

pair pA,pB,pC;
     pA=(0,0);
     pB=(1,0);
     pC=rotate(60,pA)*pB;
path sec=pA--pB--pC--cycle;
path section=scale(0.15)*sec;

coloredpath colorsec=coloredpath(section,
                      pens,
                      colortype=coloredNodes);
draw(tube(g,colorsec));
```

**Remarques :**

- On peut définir son propre coloredpath avec l'une des ces routines :

```
coloredpath coloredpath(path p, pen[] pens(real),
                        int colortype=coloredSegments);
coloredpath coloredpath(path p, pen[] pens=new pen[] {currentpen},
                        int colortype=coloredSegments);
coloredpath coloredpath(path p, pen pen(real));
```

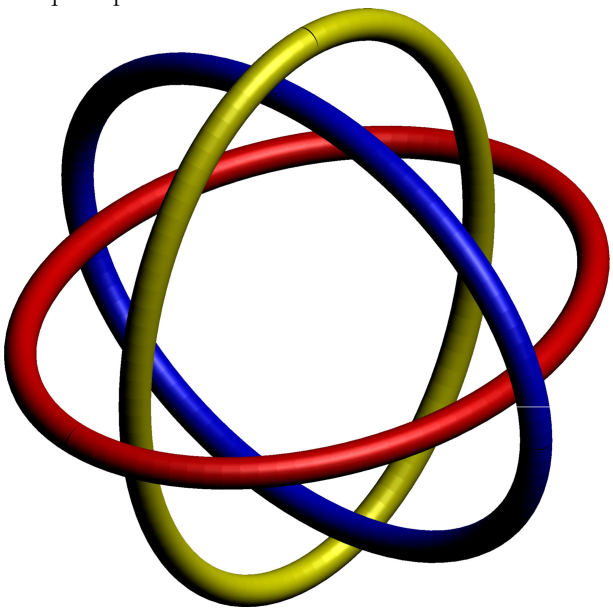
- La section peut être définie à partir de symboles comme le montre l'exemple (un peu plus complexe) de la documentation officielle [3] ci-dessous :



CODE 131

```
import tube;
import graph3;
import palette;
currentlight=White;

size(7cm);
currentprojection=perspective(1,1,1,up=-Y);

int e=1;
real x(real t) {return cos(t)+2*cos(2t);}
real y(real t) {return sin(t)-2*sin(2t);}
real z(real t) {return 2*e*sin(3t);}

path3 p=scale3(2)*graph(x,y,z,0,2pi,50,operator ..)&cycle;

pen[] pens=Gradient(6,red,blue,purple);
pens.push(yellow);
for (int i=pens.length-2; i >= 0 ; --i)
  pens.push(pens[i]);

path sec=scale(0.25)*texpath("$\pi$")[0];
coloredpath colorsec=coloredpath(sec, pens,colortype=coloredSegments);
draw(tube(p,colorsec),render(merge=true));
```

Un autre exemple, de mon cru celui là qui représente les « anneaux de Borromée » :



CODE 132

```
import graph3;
import tube;
currentprojection=orthographic(2,2,2);
size(8cm);

real a=2, b=1.5;
real x(real t) {return a*cos(t);}
real y(real t) {return b*sin(t);}
real z(real t) {return 0;}

path3 an1=graph(x,y,z,0,2*pi);
path3 an2=graph(z,x,y,0,2*pi);
path3 an3=graph(y,z,x,0,2*pi);

draw(tube(an1,scale(0.1)*unitcircle), red);
draw(tube(an2,scale(0.1)*unitcircle), blue);
draw(tube(an3,scale(0.1)*unitcircle), yellow);
```

# Index

## Z

# Références

[1] Christophe Grospellier. *Asymptote – Démarrage rapide*. 2010. url : http://cgmaths.fr/Atelier/Asymptote/Asymptote.html.

[2] Olivier Guibé. *Asymptote : un survol*. 2009. url : http://math.mad.free.fr/wordpress/?page_id=41.

[3] Andy Hammerlindl et al. *Asymptote : the Vector Graphics Language*. 2010. url : http://asymptote.sourceforge.net/asymptote.pdf.

[4] Philppe Ivaldi. *PIPRIME.fr*. url : http://www.piprime.fr/asymptote/.

[5] Gaétan Marris. *ASYMPTOTE, gallerie d'exemples. Galeries de 640 exemples, par thèmes, de figures réalisées avec Asymptote*. url : http://marris.org/asymptote/index.html.