Asymptote : un survol

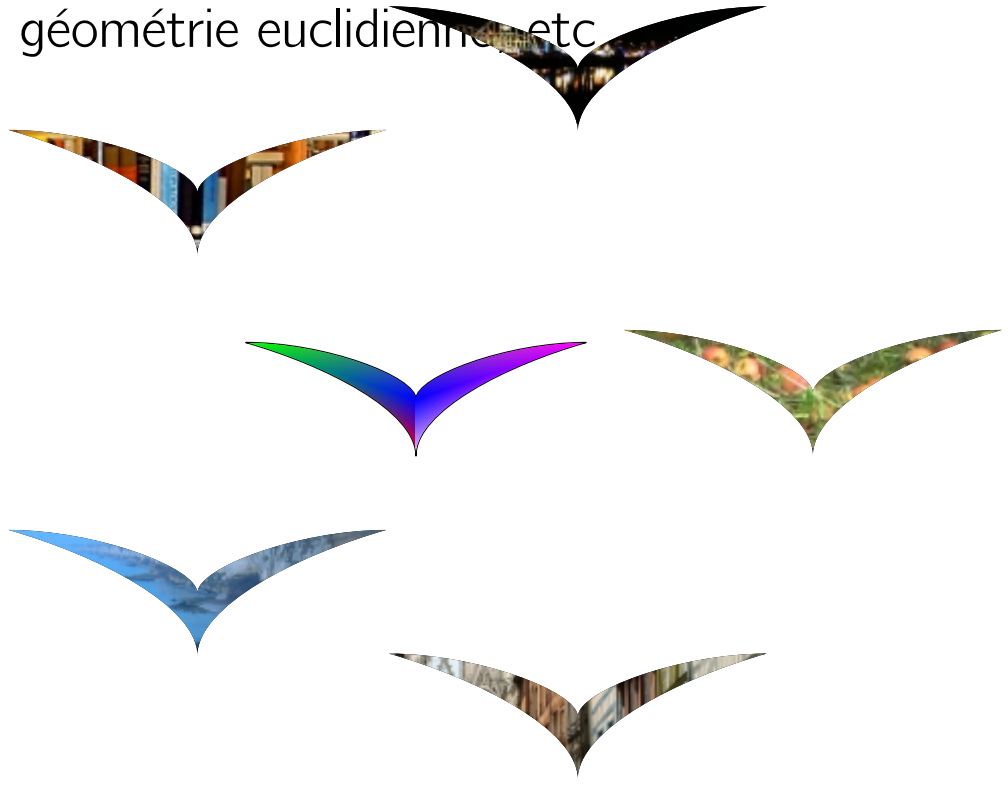
Olivier Guibé

Laboratoire de Mathématiques Raphaël Salem
CNRS - Université de Rouen

APMEP – 25 octobre 2009

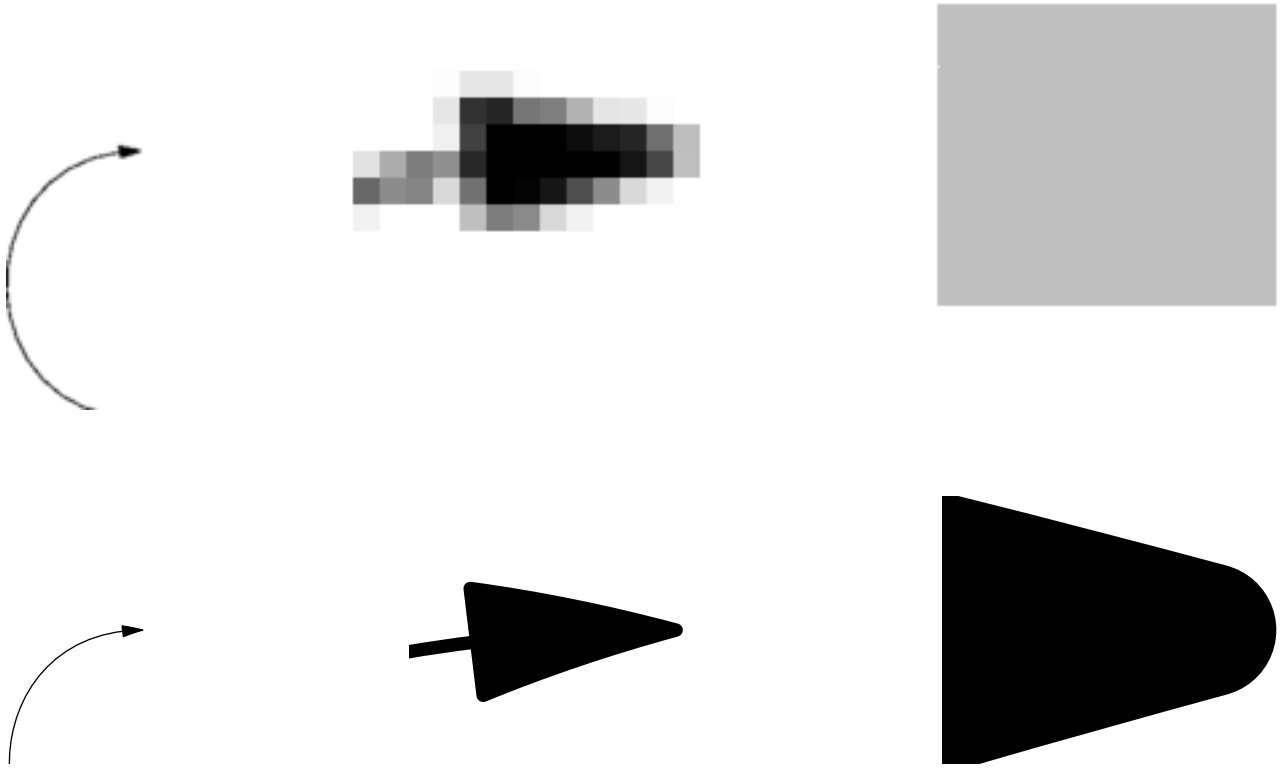
But

- graphiques de {très} (haute) qualité
- diversité des formats : png, pdf, eps, etc
- cohérence typographique
- créer/programmer des extensions pour des besoins précis : équations différentielles, graphes, géométrie euclidienne, etc
- créer des animations
- Vive la 3D !



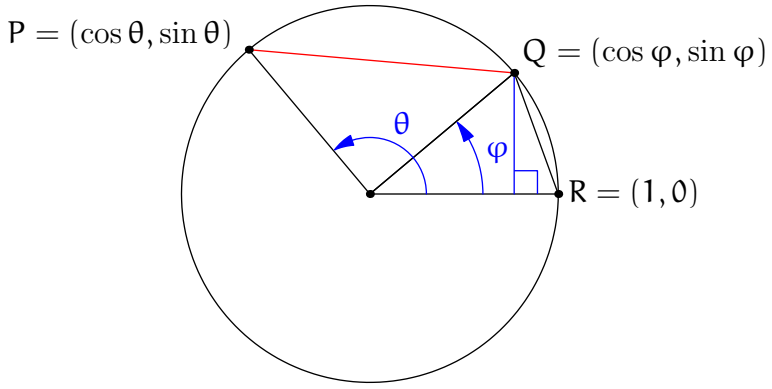
Haute qualité

Plutôt qu'un format bitmap (image composée de points) qui supporte peu ou pas l'agrandissement selon la résolution, on préfère un format vectoriel

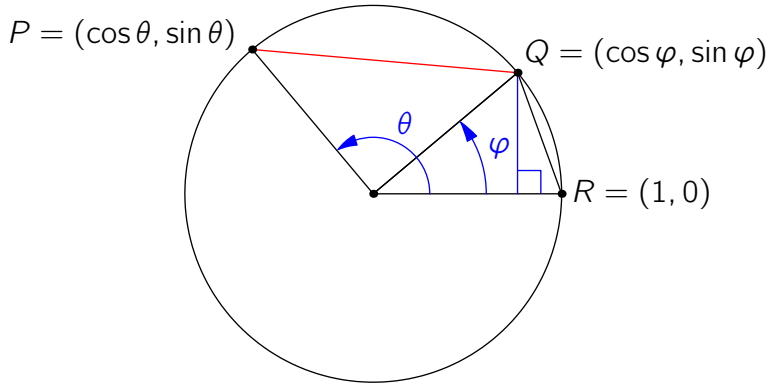


Cohérence typographique

- Formule de trigonométrie : soient P, Q et R tels que . . ., $\cos(\theta+\varphi) = \cdots$



- Formule de trigonométrie : soient P, Q et R tels que . . ., $\cos(\theta+\varphi) = \cdots$



Au crayon, à la souris ou à la ligne de code ?

- Il paraît plus naturel de dessiner via la main
- « programmer » un dessin est long, nécessite l'apprentissage d'un langage

  Cependant c'est « idéal »
- les graphes de fonctions
- les hauteurs se coupent vraiment en un point
- répéter des éléments (quadrillage, fractal, etc)
- modifier une figure de géométrie sans recommencer depuis le début
- la 3D
- symétrie, placement des objets, etc (l'oeil est très sensible à ce genre de détails)
- pour le plaisir des neurones
- passer le temps

Offre actuelle

- PsTricks : code Postscript intégré au document LaTeX via des macros, excellente intégration au document LaTeX
- Ti*k*Z : commandes LaTeX, compilation LaTeX et pdfLaTeX, excellente intégration au document LaTeX
- METAPOST : langage de programmation créé par J. Hobby inspiré de METAFONT créé par D. Knuth, semi-intégré (quoique ce point évolue avec LuaTeX)
- TeXGraph : langage de programmation crée par P. Fradin, dessin « à la souris » aussi, semi-intégré
- Xfig : dessin vectoriel à la souris, insertion possible de code LaTeX
- Inkscape : dessin vectoriel à la souris
- autres libres/gratuits/payants

Asymptote en 1 page. . .

C'est un langage de programmation graphique (vectoriel)

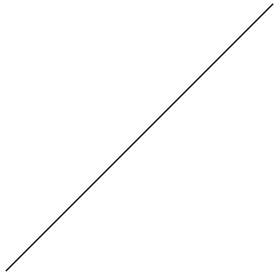
- syntaxe à la C++, précision numérique (idem qu'en C++)
- orienté mathématiquement : rotation de vecteurs, multiplication de complexes, transformation de base en 2D et 3D
- étiquettes gérées avec LaTeX, semi-intégré
- méthode du simplexe pour calculer la taille des objets, mise à l'échelle
- machine virtuelle, rapidité
- généralisation de l'algorithme de J. Hobby pour les chemins à la 3D
- commande graphique de haut-niveau, création de structure, opérations sur ces structures (comme en C++)
- en 3D : moteur de rendu OpenGL (gestion des faces cachées) et sortie PRC (Adobe)

. . .et demi

- Auteurs : John Bowman, Andy Hammerlindl, Tome Prince
- asymptote.sourceforge.net
- Le site asymptote.sourceforge.net et celui de Philippe Ivaldi www.piprime.fr ont été pillés pour l'occasion

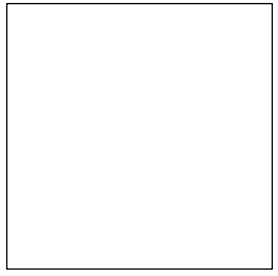
Coordonnées cartésiennes

```
draw((0,0)--(100,100)) ;
```



- l'unité est le point (big) `PostScript` (1 `bp` $= 2.54/72$ `cm`)
- `--` signifie que les points sont reliés par un segment pour créer un *chemin (path)*
- chemin cyclique :

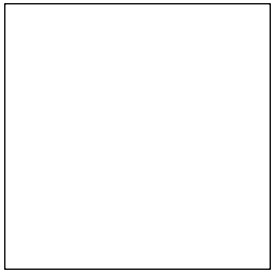
```
size(100,100) ;
draw((0,0)--(1,0)--(1,1)--(0,1)--cycle) ;
```



Spécifier une taille

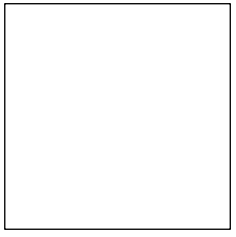
- L'unité `PostScript` n'est toujours pratique.
- L'utilisateur peut choisir une unité : `101 bp` $\approx$ `3.56cm`)

```
size(101,101) ;
draw((0,0)--(1,0)--(1,1)--(0,1)--cycle) ;
```



- Si on préfère les centimètres :

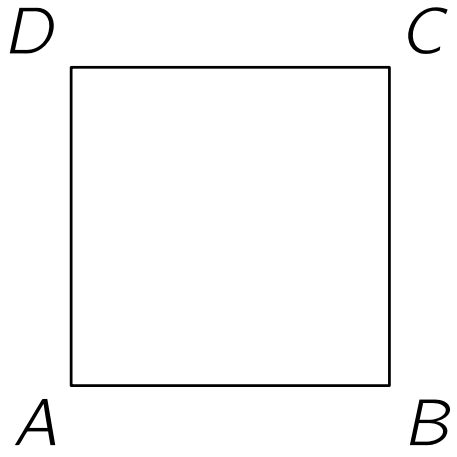
```
size(3cm,3cm) ;
draw(unitsquare) ;
```



Les étiquettes (labels)

- Ajouter et aligner des étiquettes LaTeX est facile :

```
size(3cm) ;
draw(unitsquare) ;
label("$A$",(0,0),SW) ;
label("$B$",(1,0),SE) ;
label("$C$",(1,1),NE) ;
label("$D$",(0,1),NW) ;
```



Positions N,S,W,E

- On peut jouer avec `N{ord}, S{ud}, W{Ouest}, E{st}`

```
unitsize(5cm);import fontsize;
defaultpen(fontsize(32pt));
dot("$A$",(0,0),SE);
dot("$A$",(.5,0),sqrt(.5)*S+sqrt(.5)*E);
dot("$A$",(1,0),S+E);
dot("$A$",(1.5,0),4S+E);
```

Positions N,S,W,E

- On peut jouer avec `N{ord}, S{ud}, W{Ouest}, E{st}`

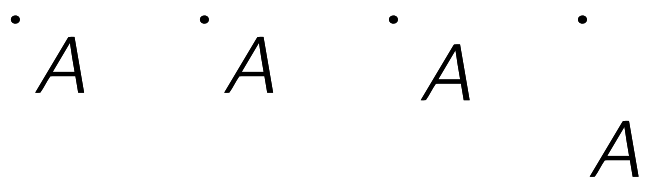
```
unitsize(5cm);import fontsize;
defaultpen(fontsize(32pt));
dot("$A$",(0,0),SE);
dot("$A$",(.5,0),sqrt(.5)*S+sqrt(.5)*E);
dot("$A$",(1,0),S+E);
dot("$A$",(1.5,0),4S+E);
```

- `unitsize(10cm)`

Positions N,S,W,E

- On peut jouer avec `N{ord}, S{ud}, W{Ouest}, E{st}`

```
unitsize(5cm);import fontsize;
defaultpen(fontsize(32pt));
dot("$A$",(0,0),SE);
dot("$A$",(.5,0),sqrt(.5)*S+sqrt(.5)*E);
dot("$A$",(1,0),S+E);
dot("$A$",(1.5,0),4S+E);
```



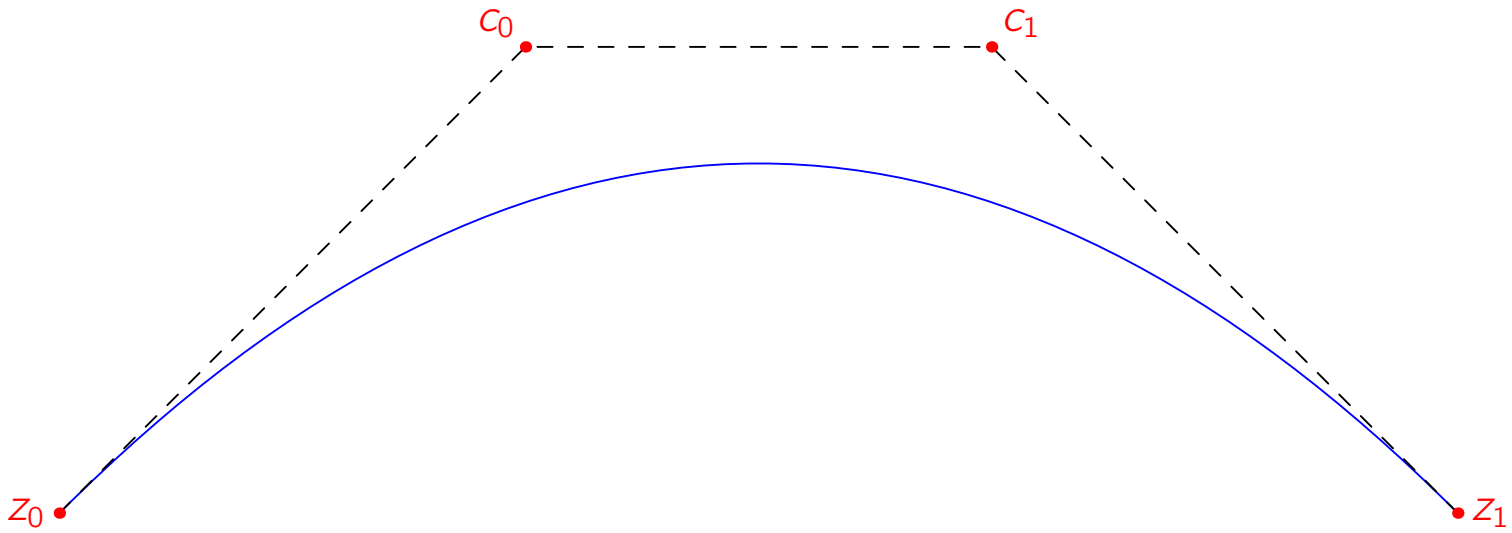
- `unitsize(10cm)`

- Les lettres, points et positions `S, E` n'ont pas été mis à l'échelle...

Asymptote sait tracer autre chose que des segments : courbes de Bézier (cubique, 2D)

- Utiliser `..` à la place de `--` construit une *courbe spline de Bézier cubique* :

```
draw(z0 .. controls c0 and c1 .. z1,blue);
```

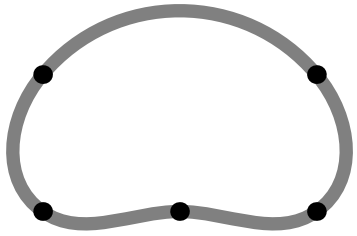


$$(1-t)^3 z_0 + 3t(1-t)^2 c_0 + 3t^2(1-t)c_1 + t^3 z_1, \qquad t \in [0,1].$$

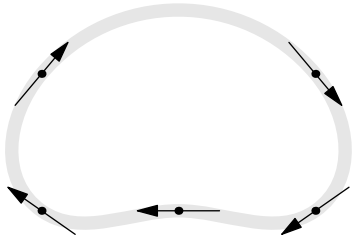
Chemin (courbe) régulier

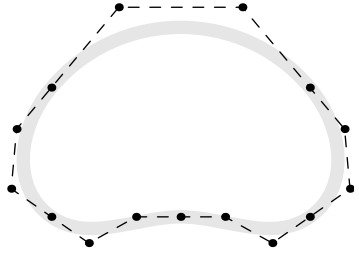
- Asymptote utilise l'algorithme de J. Hobby et D. Knuth pour le choix des points de contrôle. (algorithme éprouvé, pas d'effet surprise)

```
pair[] z={(0,0), (0,1), (2,1), (2,0), (1,0)};
draw(z[0]..z[1]..z[2]..z[3]..z[4]..cycle,
     grey+linewidth(5));
dot(z,linewidth(7));
```



- Un système linéaire permet d'abord de trouver les tangentes en chaque noeud. Ensuite les points de contrôle sont déterminés : courbe $\mathcal{C}^2$.

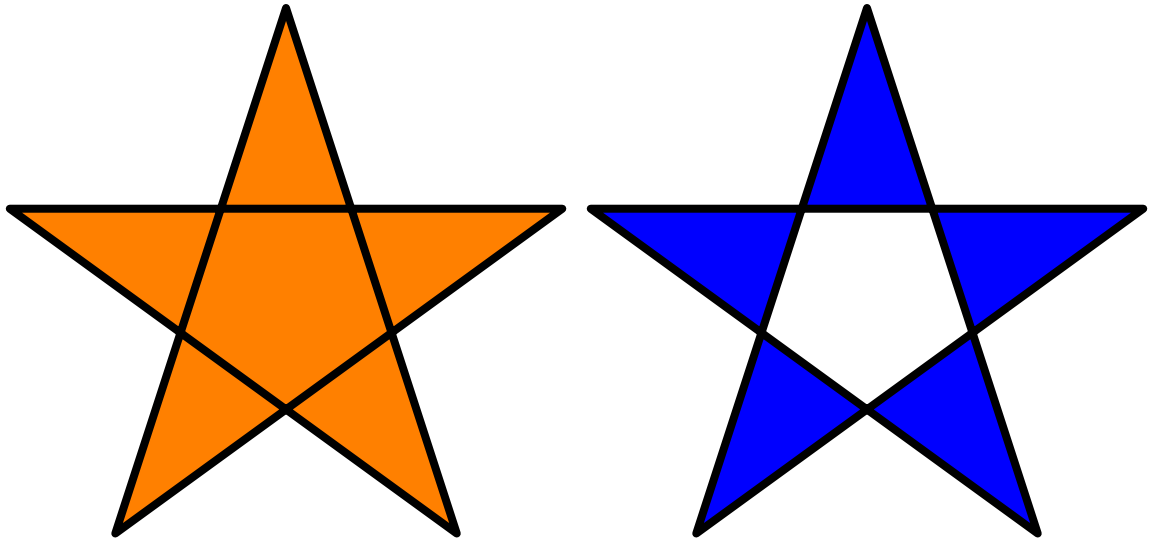




Colorier, remplir

- Utiliser `fill` pour colorier la région intérieure délimitée un chemin :

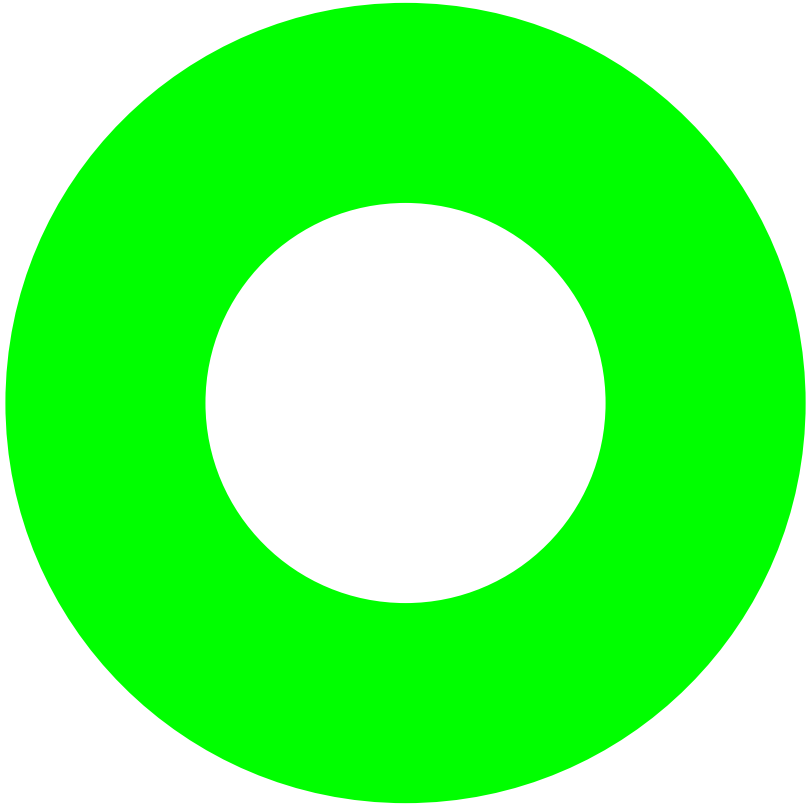
```
path star;
for (int i=0; i<5; ++i)
  star=star--dir(90+144i);
star=star--cycle;
fill(shift(-1,0)*star,orange+zerowinding);
draw(shift(-1,0)*star,linewidth(3));
fill(shift(1,0)*star,blue+evenodd);
draw(shift(1,0)*star,linewidth(3));
```



Colorier/remplir

- On peut utiliser une liste de chemin pour remplir une région à trou :

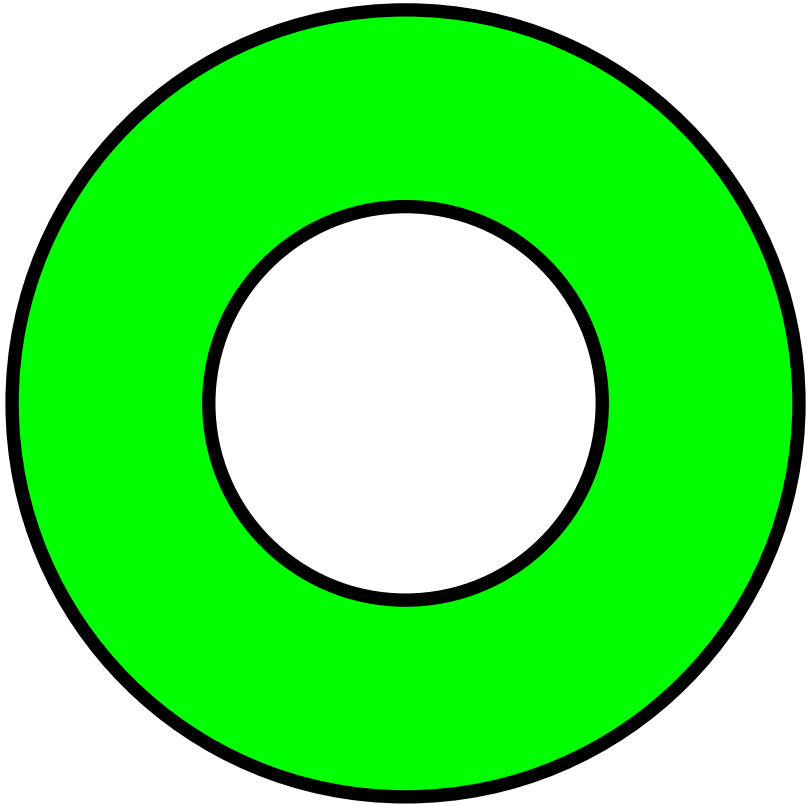
```
path[] p={scale(2)*unitcircle, reverse(unitcircle)};
fill(p,green+zerowinding);
```



Colorier/remplir

- Autre méthode pour concaténer les chemins et avec une autre règle de remplissage

```
path [] p=scale(2)*unitcircle^^unitcircle;
fill(p,evenodd+green);
draw(p,linewidth(5));
```



Découpage (clipping)

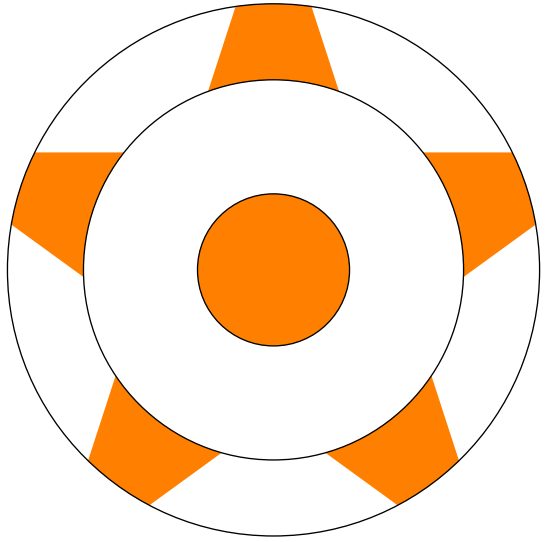
- Les figures peuvent être découpés selon un chemin :

```
fill(star,orange+zerowinding) ;
clip(scale(0.7)*unitcircle) ;
draw(scale(0.7)*unitcircle) ;
```

- Toutes les possibilités graphiques d'Asymptote sont basées sur ces quatre commandes primitives : `draw`, `fill`, `clip`, et `label`.

evenodd bis

```
fill(star,orange+zerowinding) ;
path [] p=scale(0.5)*unitcircle^^scale(0.2)*unitcircle
          ^^scale(0.7)*unitcircle ;
clip(p,evenodd) ;
draw(p) ;
```



- Exercice : à quoi correspond ce `evenodd` ?

Transformations affines

- Transformations affines : translation, rotation, réflexion et homothétie .

```
transform t=rotate(90) ;
write(t*(1,0)) ;  // Writes (0,1).
```

- Ces transformations agissent sur les coordonnées de points, chemins, pinceaux, chaînes de caractère et toute figure.

```
fill(P,blue) ;
fill(shift(2,0)*reflect((0,0),(0,1))*P, red) ;
fill(shift(4,0)*rotate(30)*P, yellow) ;
fill(shift(6,0)*yscale(0.7)*xscale(2)*P, green) ;
```

Syntaxe à la C++/Java

```
// Déclaration : x déclaré comme réel (flottant) :
real x ;

// Affectation : x affecté de la valeur 1.
x=1.0 ;

// Test  : x égal à 1 ou non.
if(x == 1.0) {
  write("x vaut 1.0") ;
} else {
  write("x n'est pas égal à 1.0") ;
}

// Boucle : itération de 10.
for(int i=0 ; i < 10 ; ++i) {
  write(i) ;
}
```

Notation mathématique

- La division de deux entiers retourne le type `real`. Utiliser `quotient` pour la division entière :

```
3/4==0.75          quotient(3,4)==0
```

- « ^ » pour élever à la puissance :

```
2^3    2.7^3    2.7^3.2
```

- La multiplication par un scalaire est implicite pour beaucoup d'expressions :

```
2pi    10cm    2x^2    3sin(x)    2(a+b)
```

- Le type `pair` équivaut au nombre complexe :

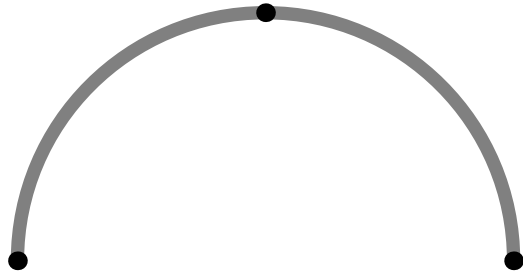
```
(0,1)*(0,1)==(-1,0)
```

Type

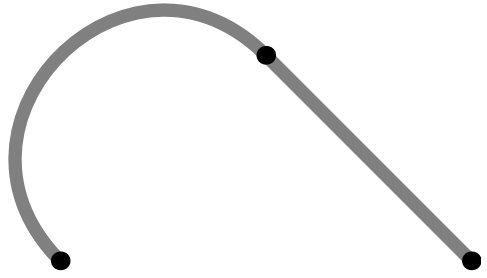
Type	Description
`bool`	le classique booléen `true` ou `false`
`int`	le type entier
`real`	le type réel (flottant)
`pair`	le type paire équivaut à `(real, real)`, nombre complexe
`triple`	le type 3 coordonnées réelles pour la 3D
`path`	courbe de Bézier spline cubique résolue (points de contrôle déterminés par Asymptote ou vos soins)
`guide`	courbe de Bézier spline cubique non résolue (points de contrôle non tous déterminés) jusqu'au tracé
`picture`	un canevas pour dessiner avec unité de longueur spécifiée par l'utilisateur
`frame`	un canevas pour dessiner avec unité `Postscript`
`pen`	pinceau utilisé pour le tracé. Couleur, forme, épaisseur, motif sont définissables
`transform`	transformation affine. S'applique à tout ce qui se dessine

Guide/Path what is the difference ?

```
guide p=(1,0)..(0,1) ;
guide pp=p..(-1,0) ;
draw(pp,grey+linewidth(5)) ;
dot(pp,linewidth(7)) ;
```



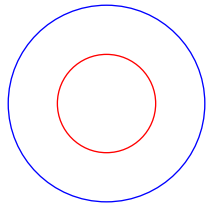
```
path p=(1,0)..(0,1) ;
path pp=p..(-1,0) ;
draw(pp,grey+linewidth(5)) ;
dot(pp,linewidth(7)) ;
```



Appel de fonction

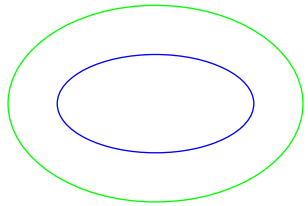
- Une fonction peut prendre des arguments par défaut et qu'importe la position. La règle, c'est la 1ère possibilité qui prime :

```
void drawEllipse(real xsize=1, real ysize=xsize, pen p=blue) {
  draw(xscale(xsize)*yscale(ysize)*unitcircle, p) ;}

drawEllipse(2) ;
drawEllipse(red) ;
```



- Les arguments peuvent être spécifiés par leur nom :

```
drawEllipse(xsize=2, ysize=1) ;
drawEllipse(ysize=2, xsize=3, green) ;
```



Fonction avec différents arguments possible

L'exemple frappant est la commande `graph` d'Asymptote qui accepte comme arguments, une fonction et un intervalle, un tableau de `pair`, etc. Ce qui correspond à plusieurs situations pour un même but : créer un graphe.

- Exemple stupide

```
int stupide (int a, int b){
return a+b ;}
int stupide (real x){
return ceil(x*x) ;}
write(stupide(2,2)) ;
write(stupide(2.4)) ;
```

Affiche 4 et 6

. . .comme argument

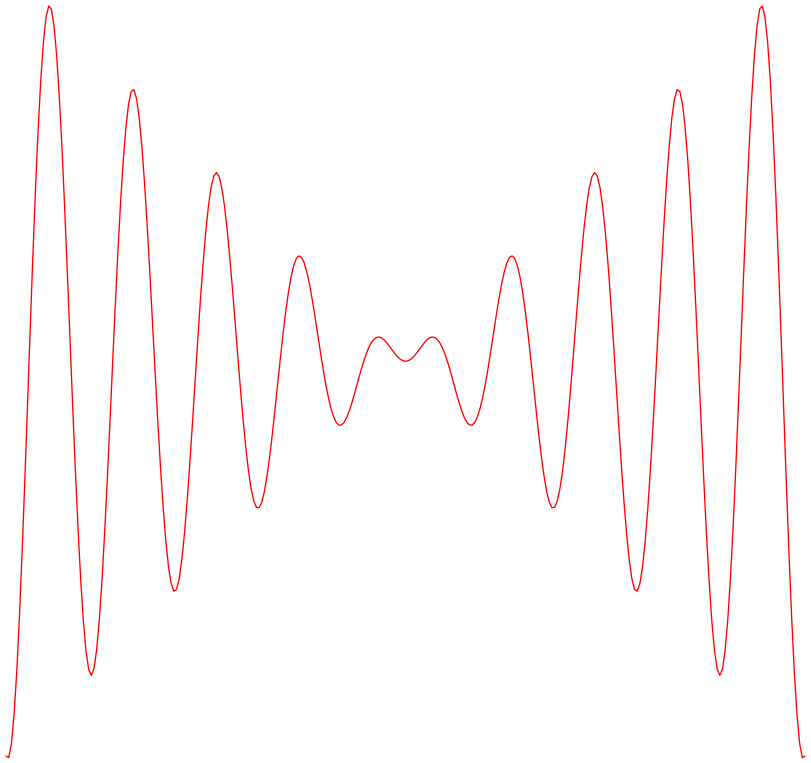
- Permet d'écrire une fonction qui prendra un nombre arbitraire d'arguments :

```
int som(... int[] nums) {
  int total=0;
  for (int i=0; i < nums.length; ++i)
    total += nums[i];
  return total;
}

som(1,2,3,4);                              // retourne 10
som();                                     // retourne 0
som(1,2,3 ... new int[] {4,5,6});          // retourne 21

int soustrait(int start ... int[] subs) {
  return start - sum(... subs);
}
```

Fonction de fonction

- Si tout est bien écrit, une fonction peut être donnée en argument à une autre fonction :

```
real f(real x) {
    return x*sin(10x) ;
}
draw(graph(f,-3,3,300),red) ;
```

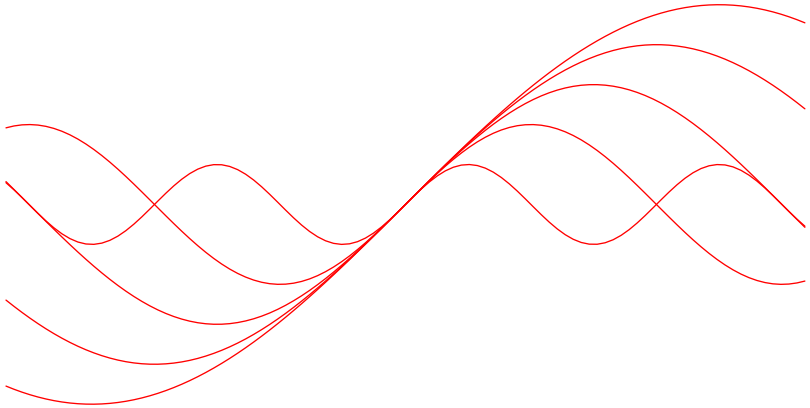


Fonctions de fonctions

- Une fonction peut retourner une fonction :

$$f_n(x) = n \sin\left(\frac{x}{n}\right).$$

```
typedef real func(real);
func f(int n) {
  real fn(real x) {
    return n*sin(x/n);
  }
  return fn;
}

func f1=f(1);
real y=f1(pi);

for (int i=1; i<=5; ++i)
  draw(graph(f(i),-10,10),red);
```



Fonction anonyme

- Création d'une nouvelle fonction avec `new` :

```
path p=graph(new real (real x) { return x*sin(10x) ; },-3,3,red)
func f(int n) {
  return new real (real x) { return n*sin(x/n) ; } ;
}
```

équivalent à l'exemple précédent

- La syntaxe de définition d'une fonction revient en fait à assigner une fonction objet à une variable.

```
real square(real x) {
  return x^2 ;
}
```

est équivalent à

```
real square(real x) ;
square=new real (real x) {
  return x^2 ;
} ;
```

Structures

- Comme d'autres langages, les structures aident à regrouper/structurer des données ;

```
struct Personne {
  string prenom, nom ;
  int age ;
}
Personne bob=new Personne ;
bob.firstname="Roger" ;
bob.lastname="Dupond" ;
bob.age=24 ;
```

- Tout code dans le corps de la structure est exécuté à chaque fois qu'une nouvelle structure est allouée

```
struct Personne {
  write("fabrication d'identité.") ;
  string prenom, nom ;
  int age=18 ;
}
Personne eve=new Personne ;   // écrit  "fabrication d'identité.
write(eve.age) ;              // écrit 18.
```

Programmation orientée objet

- Des fonctions peuvent être définies pour chaque instance d'une structure.

```
struct Quadratique {
  real a,b,c ;
  real discriminant() {
    return b^2-4*a*c ;
  }
  real eval(real x) {
    return a*x^2 + b*x + c ;
  }
}
```

- Ainsi on peut déclarer dans la structure de l'objet des fonctions permettant d'exploiter une facette de l'objet : ici la fonction polynôme

```
Quadratique poly=new Quadratique ;
poly.a=-1 ; poly.b=1 ; poly.c=2 ;

real f(real x)=poly.eval ;
real y=f(2) ;
draw(graph(poly.eval, -5, 5)) ;
```

En vrac

- Création d'objet spécialisé en redéfinissant la méthode/structure
- Gestion des tableaux classique et aussi à-la-Python (slice)
- Asymptote sait résoudre un système linéaire. Si si, c'est utile pour faire l'interpolation spline cubique naturel, encastré, not-a-knot ou monotone par exemple
- Résolution numérique de polynôme de degré 3, 4
- ...

Operateurs

- Les opérateurs +,-, etc sont en réalité des appels de fonctions et peuvent être (re)définis via la commande `operator`.

```
int add(int x, int y)=operator +;
write(add(2,3));  // Writes 5.

// Mieux vaut éviter cette loi de composition
// interne non commutative !
int operator +(int x, int y) {
  return add(2x,y);
}
write(2+3);  // Writes 7.
```

- Cela permet de définir des opérateurs pour les nouveaux types/structures.

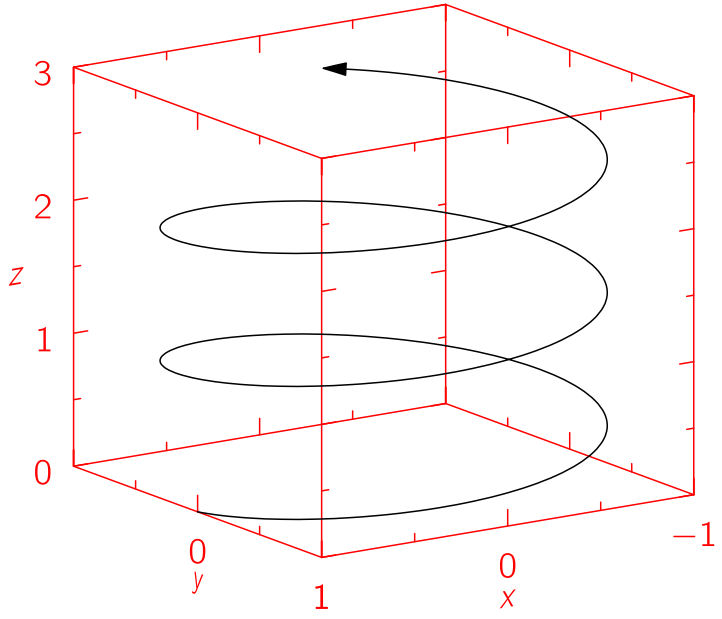
Operateurs

- Les opérateurs pour construire les chemins sont aussi des fonctions

  `a.. controls b and c .. d--e`
  est équivalent à

  `operator --(operator ..(a, operator controls(b,c), d), e)`

- Ceci permet de redéfinir tous les opérateurs concernant les chemins/guide pour leur équivalent en 3D



Paquet ou extension

- Les définitions de fonctions et de structures peuvent être regroupées dans un paquet :

```
// powers.asy
real square(real x) { return x^2; }
real cube(real x) { return x^3; }
```

et importé via la commande

```
import powers;
real eight=cube(2.0);
draw(graph(powers.square, -1, 1));
```

- En vrac : interpolation, géométrie, dessin « main levée », diagramme de Feynman, arbre, motif de remplissage, marqueur (angle, segment), spline, animation, texte sur un chemin, slide, graphe de fonctions (2D, 3D), tube, solides, edo, ligne de niveau (2D et 3D), les vôtres dès demain matin

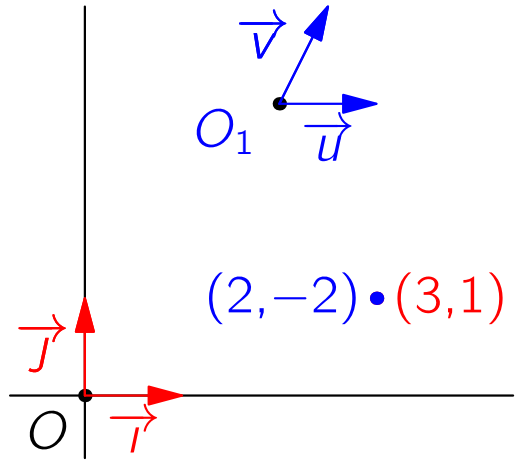
Paquet de géométrie euclidienne

`geometry.asy` est certainement la meilleure extension pour faire de la géométrie dans le plan. L'auteur est Philippe Ivaldi : www.piprime.fr/asymptote

- Structure pour les point, vecteur, point massique, triangle, droite (segment, demi-droite), conique (cercle, ellipse, parabole, hyperbole), arc
- changement de repère (affine), ajout de transformations affines
- plusieurs types d'abcisse, dont l'abcisse curviligne
- triangle : orthocentre, cercle conscrit, point de Gergonne, triangle de Cevian, etc
- inversion
- La Rolex de la géométrie euclidienne

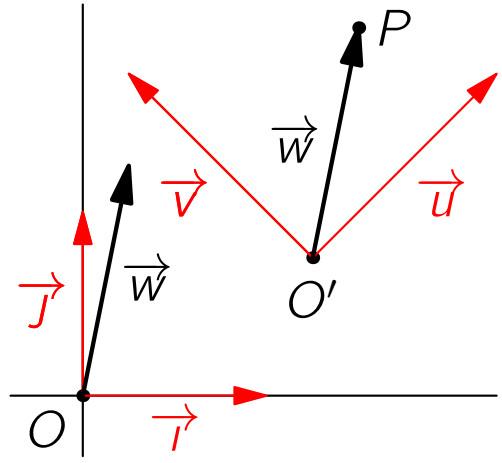
Géométrie euclidienne : des répères

```
import geometry ;
  size(4cm,0) ;
  coordsys R=cartesiansystem((2,3), i=(1,0), j=(0.5,1)) ;
  show(currentcoordsys) ;
  show(Label("$O_1$",blue), Label("$\vec{u}$",blue),
  Label("$\vec{v}$",blue), R, xpen=invisible, ipen=blue) ;
  pair A=(3,1) ;
  dot("", A, red) ;
  point B=point(R, A/R) ;
  dot("", B, W, blue) ;
```



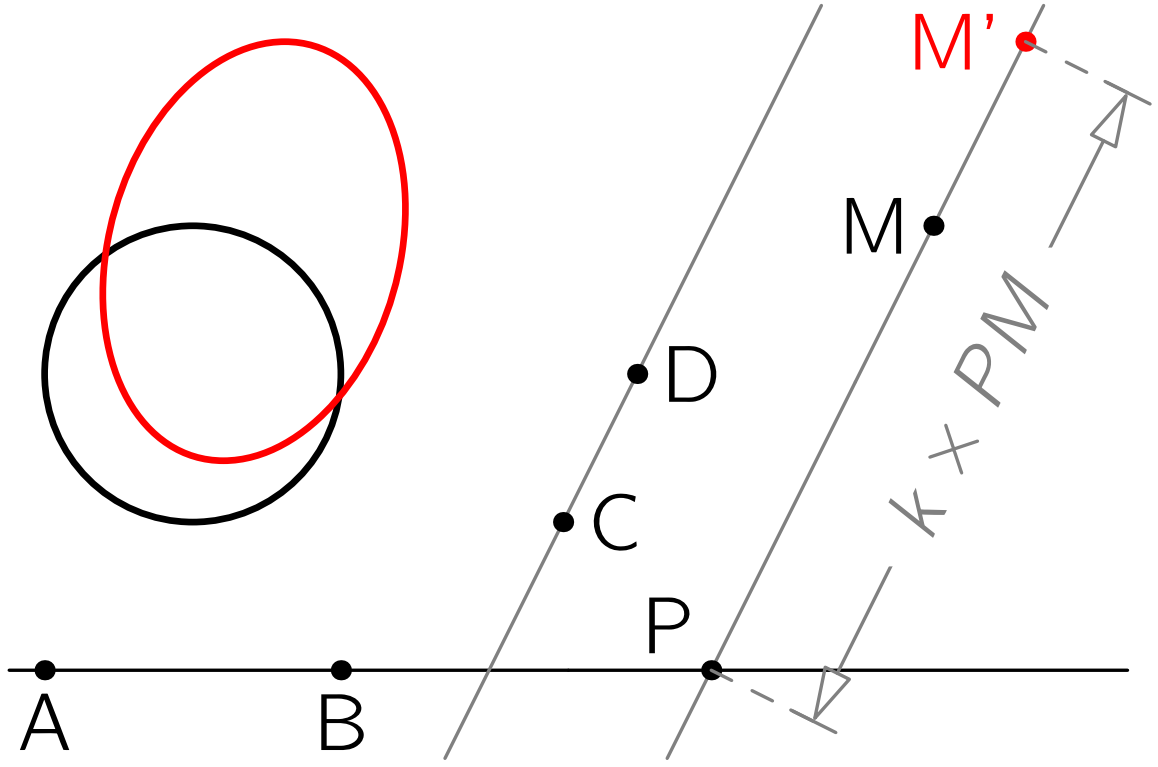
Géométrie euclidienne : vecteurs

```
import geometry ; size(4cm,0) ;
currentcoordsys=cartesiansystem((1.25,0.75), i=(1,1), j=(-1,1)) ;
coordsys Rp=currentcoordsys ; coordsys R=defaultcoordsys ;
show(R) ;
show("$O'$","$\vec{u}$","$\vec{v}$", Rp, xpen=invisible) ;
point P=(0.75,0.5) ; dot("$P$",P) ; vector w=P ;
pen bpp=linewidth(bp) ;
draw("$\vec{w}$", origin()--origin()+w, W, bpp, Arrow(3mm)) ;
draw("$\vec{w}$", origin--locate(w), E, bpp, Arrow(3mm)) ;
```



Géométrie euclidienne : transformations affines

- `scale(k,A,B,C,D)` ou affinité de rapport k d'axe (AB) de direction (CD). `scale` déjà défini est complété dans geometry.asy



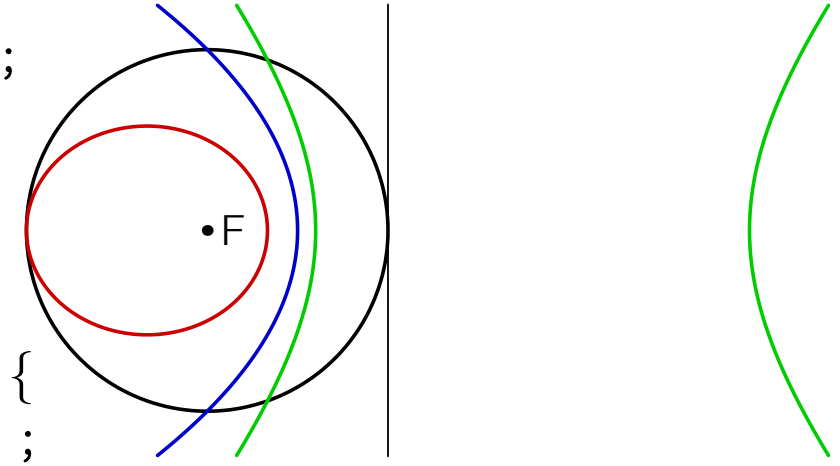
Géométrie euclidienne : transformations affines

```
import geometry ;
size(6cm,0) ;
pen bpp=linewidth(bp) ; real k=sqrt(2) ;
point A=(0,0), B=(2,0), C=(3.5,1) ;
point D=(4,2), M=(6,3) ;
path cle=shift(1,2)*unitcircle ;
draw(cle, bpp) ;
draw(line(A,B)) ;
draw(line(C,D), grey) ;
transform dilate=scale(k,A,B,C,D) ;
draw(dilate*cle, bpp+red) ;
point Mp=dilate*M ;
point P=intersectionpoint(line(A,B), line(M,Mp)) ;
draw(line(P,M), grey) ;
dot("A", A, S) ; dot("B", B, S) ; dot("C", C) ;
dot("D", D) ; dot("M", M, W) ; dot("P", P, NW) ;
dot("M'", Mp, W, red) ;
distance("$k\times PM$", P, Mp, 6mm, grey,
         joinpen=grey+dashed) ;
```

Géométrie euclidienne : les coniques

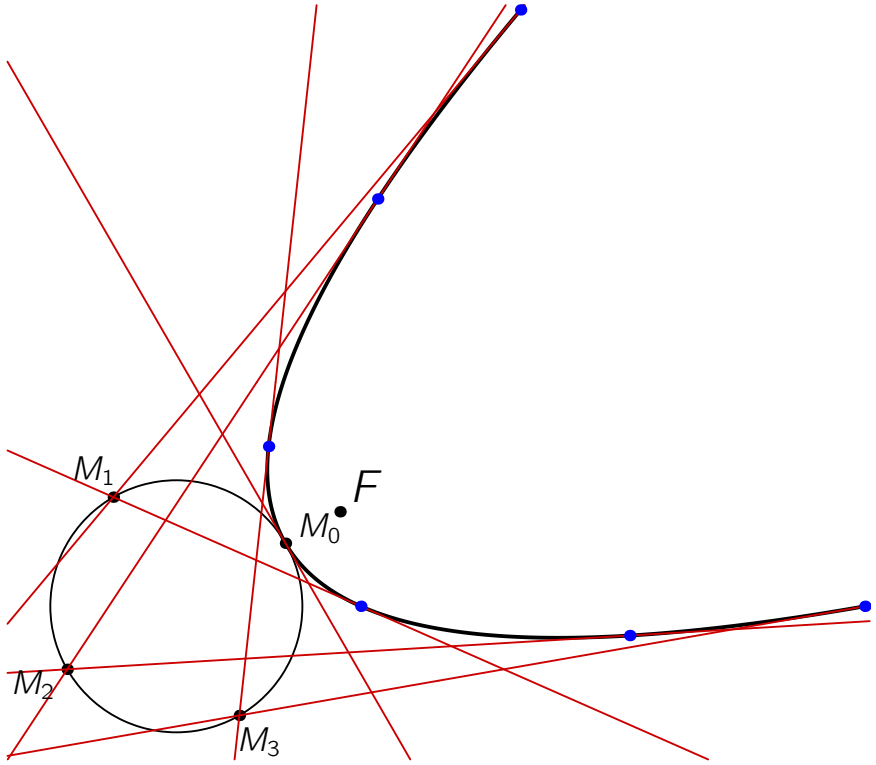
- définition via les classiques foyer, excentricité, directrice ou passant par quatre points donnés (sous condition). Merci de ne poser aucune question sur les coniques.

```
import geometry;
size(8cm,0);
point F=(0,0); dot("F", F);
line l=line((1,0),(1,1));
draw(l);
pen[] p=new pen[] {black,
        red,blue,green};
for (int i=0; i < 4; ++i) {
  conic co=conic(F,l,0.5*i);
  draw(co, bp+0.8*p[i]);
}
draw(box((-1,-1.25), (3.5,1.25)),
     invisible);
```



Géométrie euclidienne : les coniques

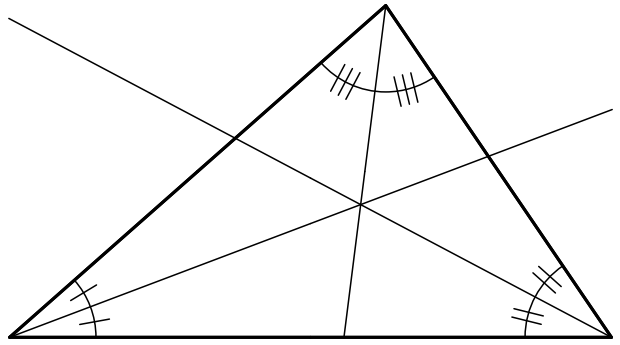
- équations de coniques, intersection de coniques via la résultante et approximation numérique de la solution
- cercle : cercle de diamètre fixé, inscrit à un triangle, puissance d'un point, tangente, etc
- arc (orienté) d'un cercle, d'une ellipse



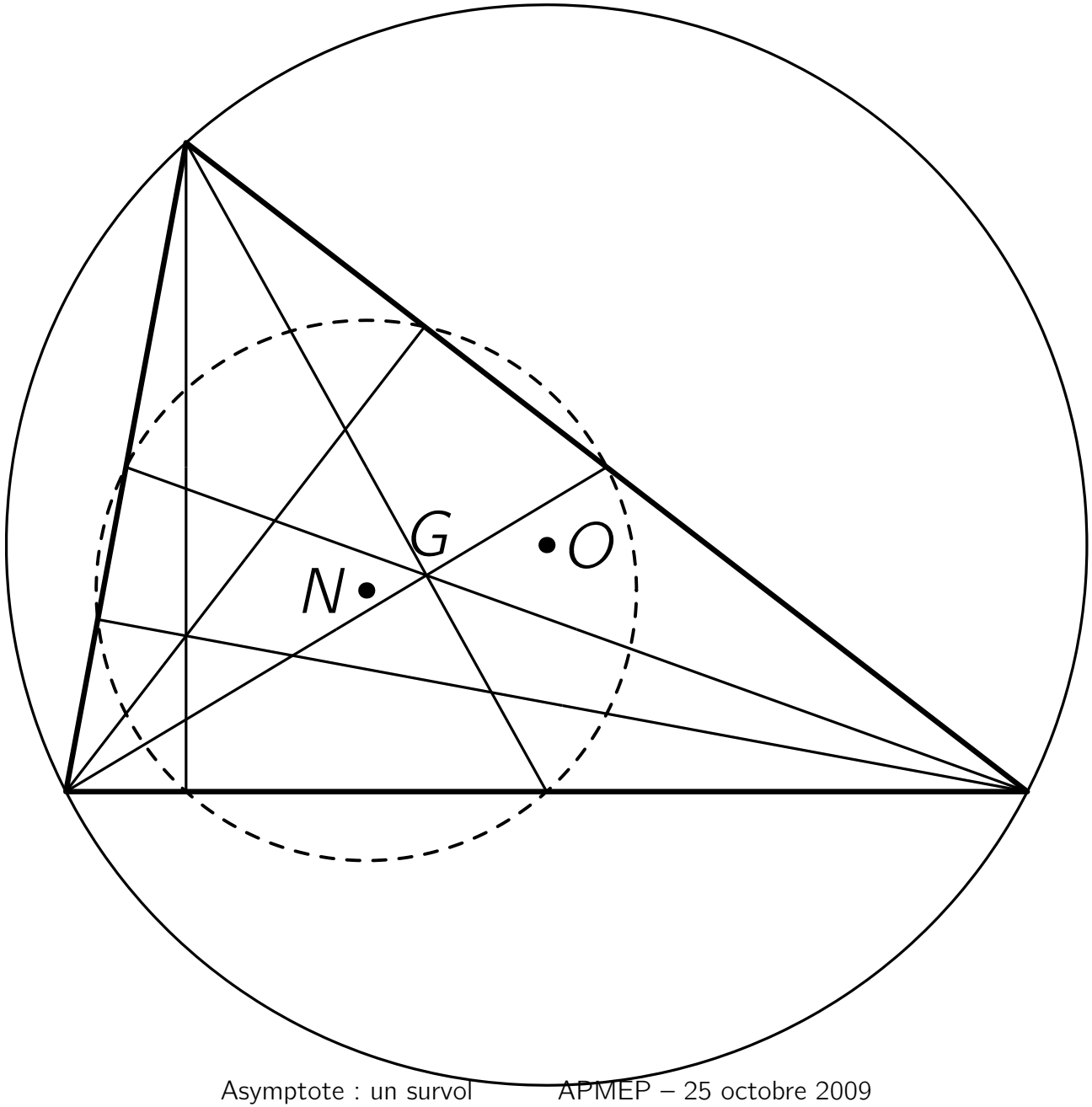
Géométrie euclidienne : les triangles

Le côté orienté objet permet une structure complexe du triangle, incluant les sommets, les côtes, la numérotation des sommets suivant l'orientation. Ainsi à tout moment on peut « récupérer » un des éléments définissant un triangle et créer des fonctions dont l'argument est le triangle et non tel ou tel côté, tel ou tel point.

- triangle défini selon 3 longueurs données (+angle), 3 points donnés, 1 angle et 2 longueurs (+angle), 3 droites.
- tous les éléments particuliers sont définis (à un ensemble de mesure non nulle près) : droite de Simson, droite isotomique, etc



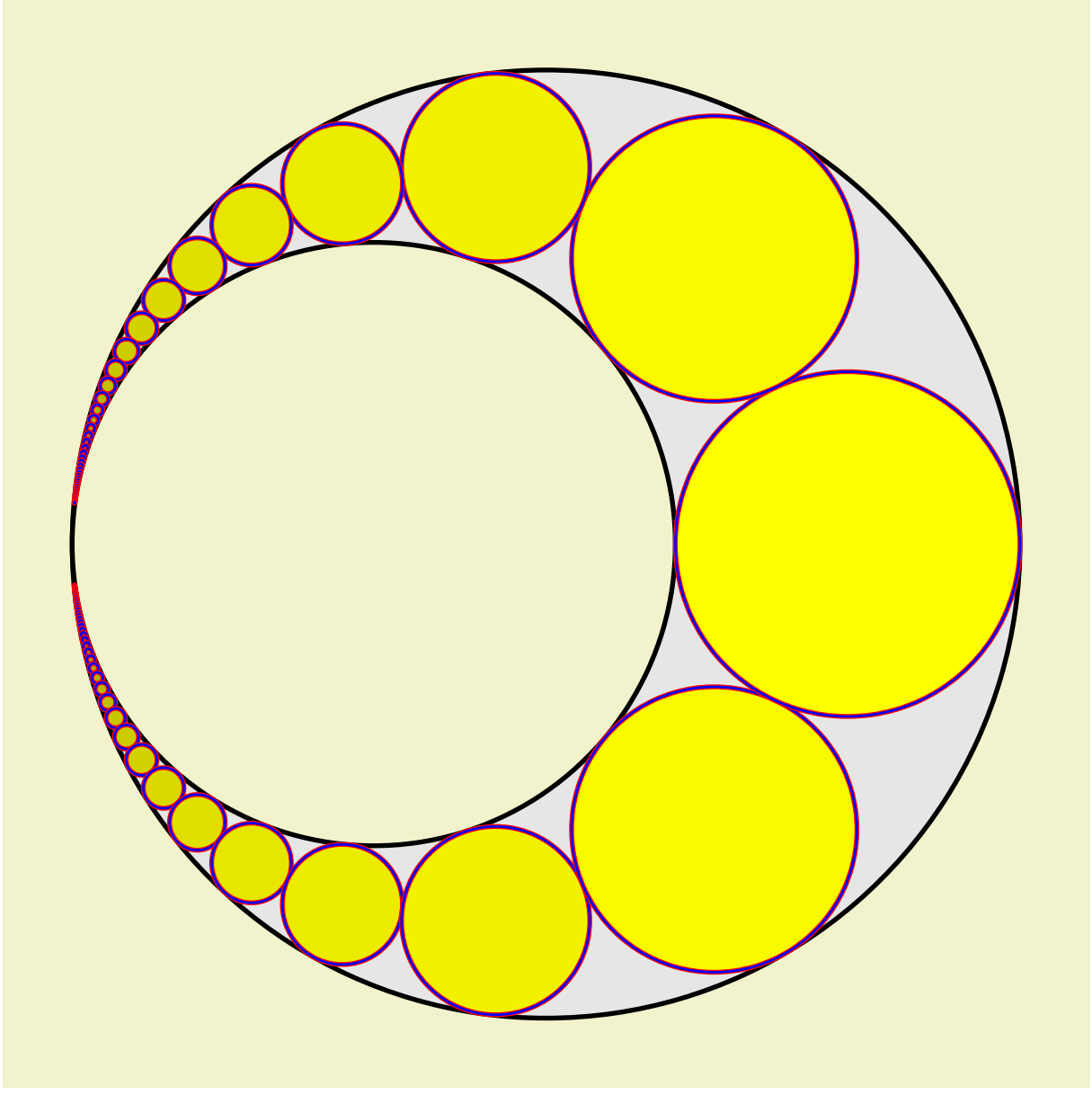
Le cercle d'Euler



Le cercle d'Euler : le code

```
import geometry ; size(200,0) ;
point A=(.5,3.5), B=(0,0.8), C=(4,.8) ;
triangle t=triangle(A,B,C) ;
draw(t,.7*linewidth(bp)) ;
line hc=altitude(t.AB) ; line ha=altitude(t.BC) ;
line hb=altitude(t.AC) ;
draw(segment(ha)^^segment(hb)^^segment(hc)) ;
point H=orthocentercenter(t) ;
draw(segment(median(t.AB))) ;
draw(segment(median(t.AC))) ;
draw(segment(median(t.BC))) ;
point G=centroid(t) ; label("$G$",G,N+.1E) ;
point pO=circumcenter(t) ; dot("$O$",pO) ;
circle c1=circle(t) ; draw(c1) ;
triangle tt=medial(t) ; circle c2=circle(tt) ;
dot("$N$",c2.C,W) ;
pen dashedd=linetype("4 4") ; draw(c2,dashedd+.6bp) ;
```

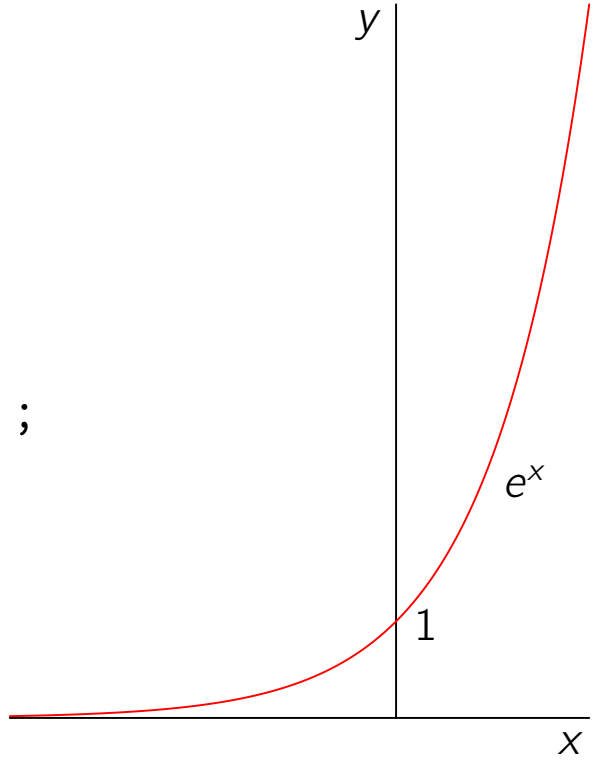
Et même les inversions



Animation

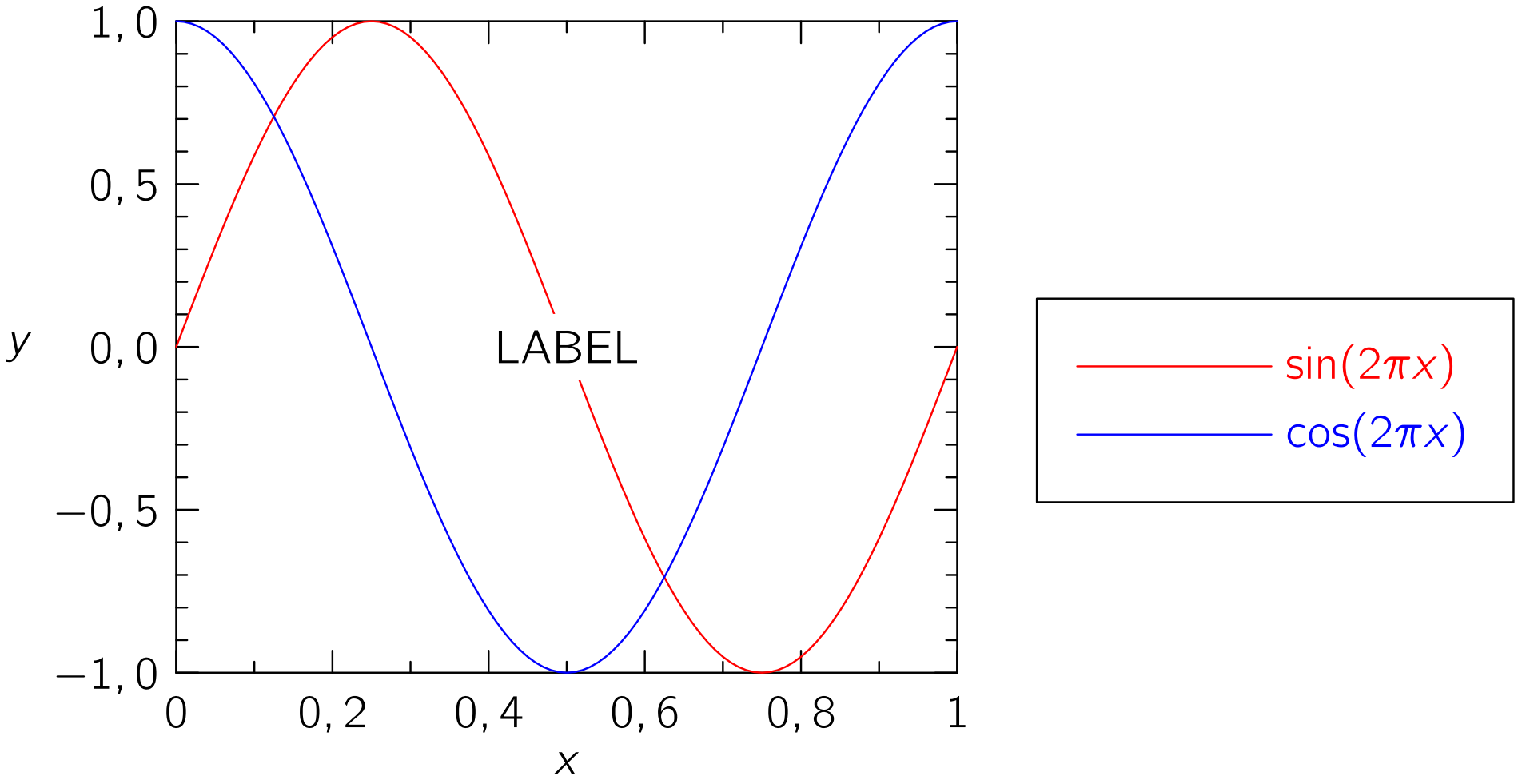
Graphes de fonctions

```
import graph ;
size(150,0) ;

real f(real x) {return exp(x) ;}
pair F(real x) {return (x,f(x)) ;}

xaxis("$x$") ;
yaxis("$y$",0) ;

draw(graph(f,-4,2,operator ..),red) ;

labely(1,E) ;
label("$e^x$",F(1),SE) ;
```



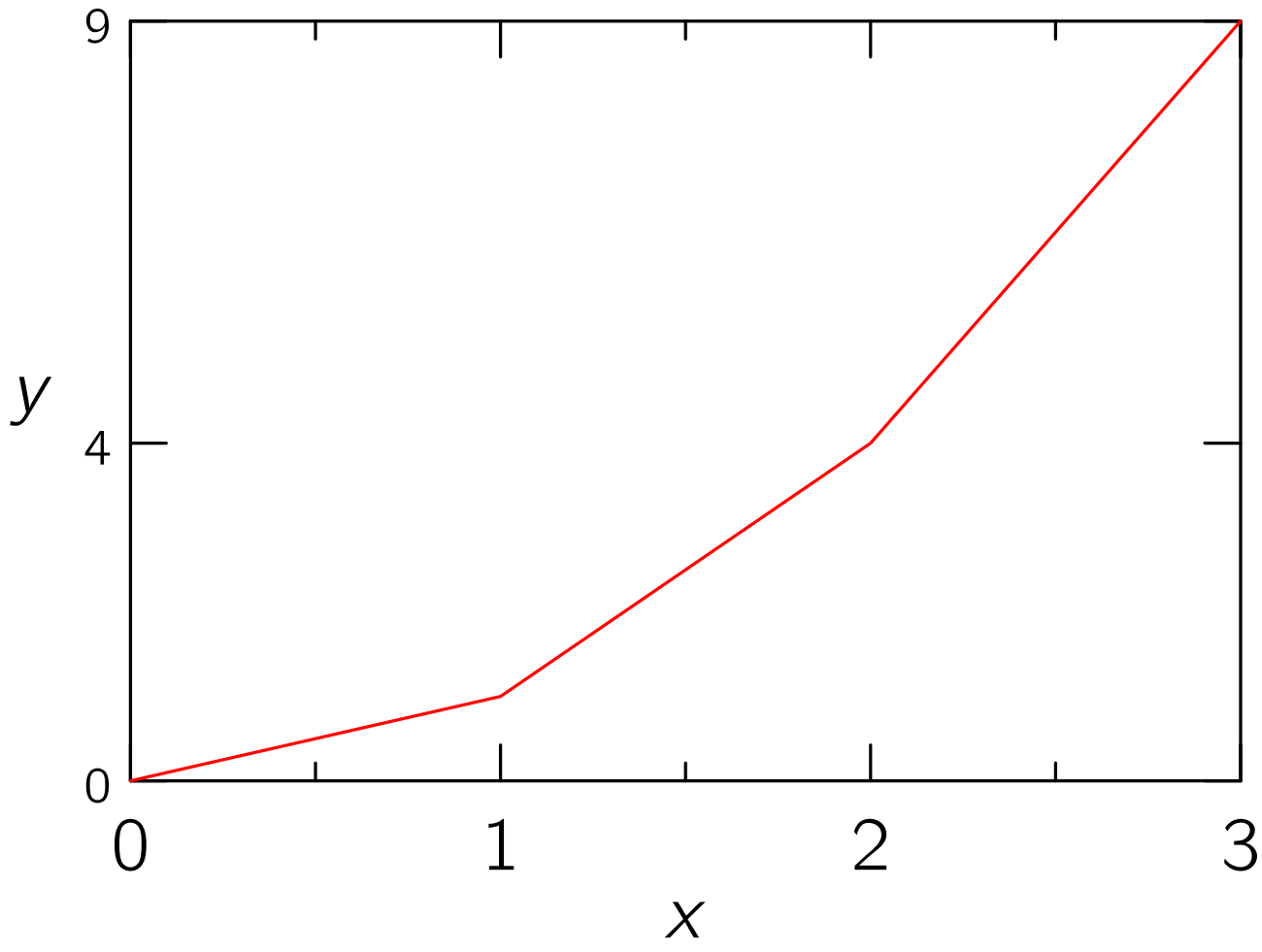
Graphes scientifiques

```
import graph ;

size(250,200,IgnoreAspect) ;

real Sin(real t) {return sin(2pi*t) ;}
real Cos(real t) {return cos(2pi*t) ;}

draw(graph(Sin,0,1),red,"$\sin(2\pi x)$") ;
draw(graph(Cos,0,1),blue,"$\cos(2\pi x)$") ;

xaxis("$x$",BottomTop,LeftTicks) ;
yaxis("$y$",LeftRight,RightTicks(trailingzero)) ;

label("LABEL",point(0),UnFill(1mm)) ;

attach(legend(),truepoint(E),20E,UnFill) ;
```


1,0
0,5
0,0
−0,5
−1,0
y
0
0,2
0,4
0,6
0,8
1
x
LABEL
sin(2πx)
cos(2πx)

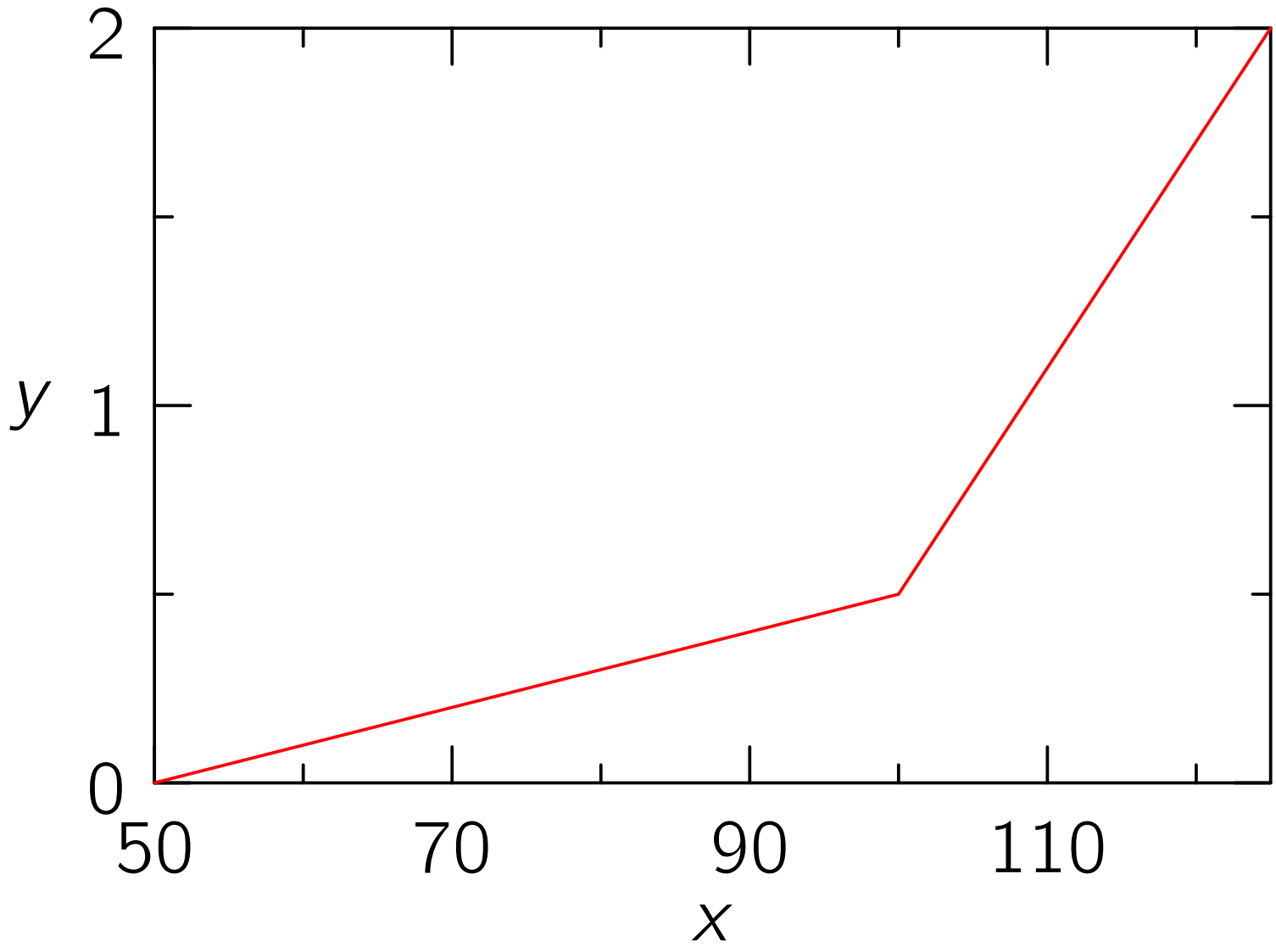
Données

```
import graph;

size(200,150,IgnoreAspect);

real[] x={0,1,2,3};
real[] y=x^2;

draw(graph(x,y),red);

xaxis("$x$",BottomTop,LeftTicks);
yaxis("$y$",LeftRight,
      RightTicks(Label(fontsize(8pt)),new real[]{0,4,9}));
```


9
y
4
0
0
1
2
3
x

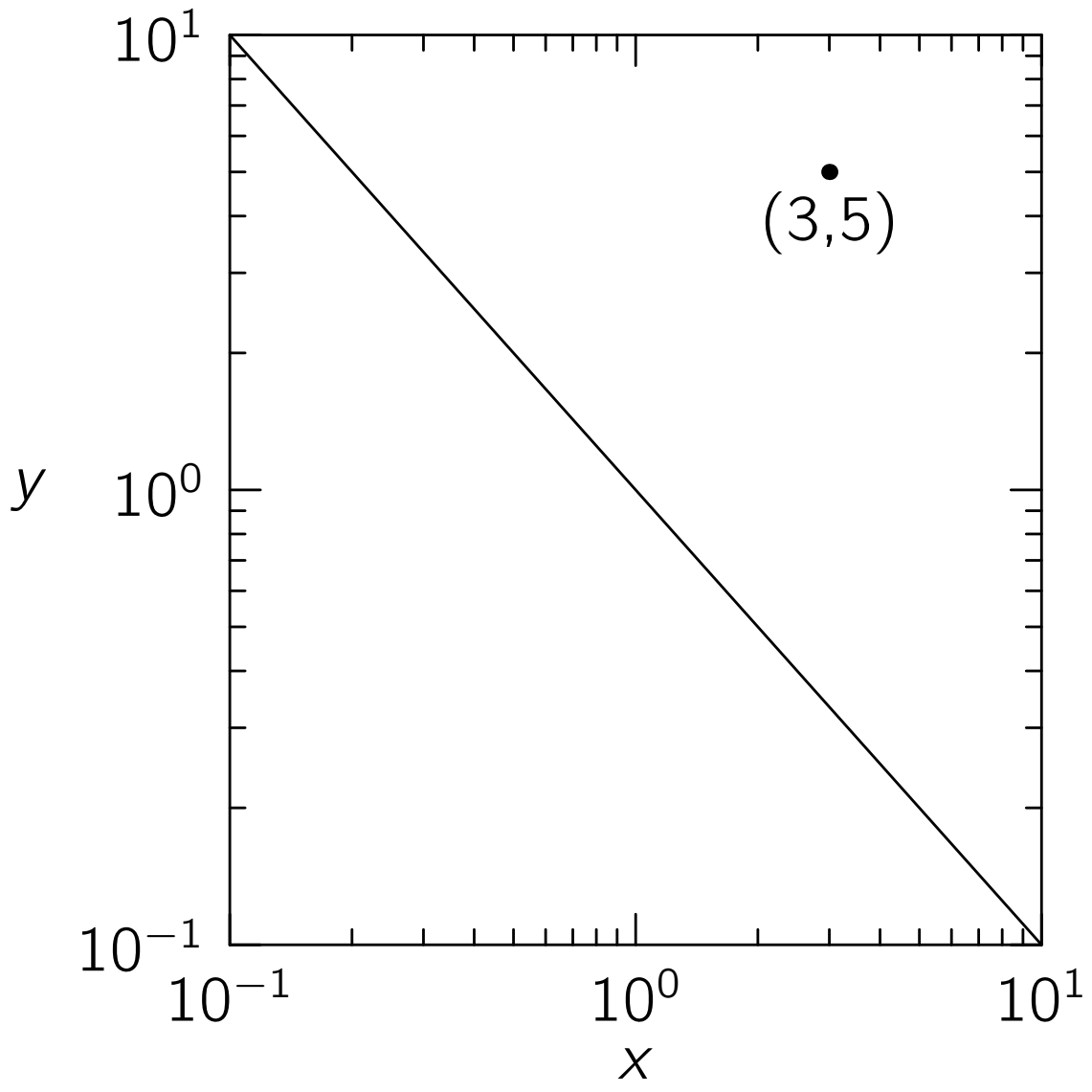
Importation de données

```
import graph ;

size(200,150,IgnoreAspect) ;

file in=input("filegraph.dat").line() ;
real[][] a=in.dimension(0,0) ;
a=transpose(a) ;

real[] x=a[0] ;
real[] y=a[1] ;

draw(graph(x,y),red) ;

xaxis("$x$",BottomTop,LeftTicks) ;
yaxis("$y$",LeftRight,RightTicks) ;
```


2
y
1
0
50
70
90
110
x

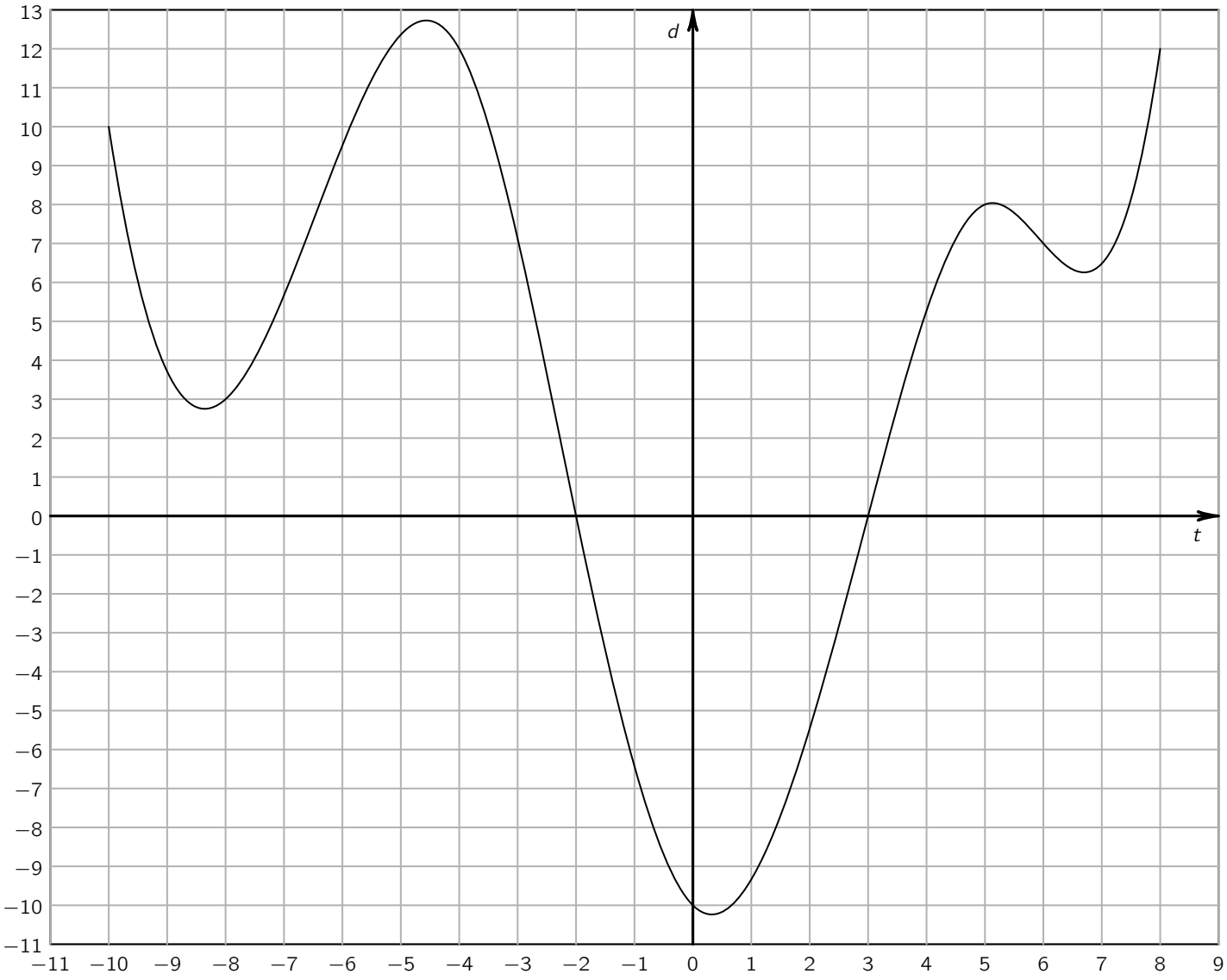
Échelle logarithmique

```
import graph ;

size(200,200,IgnoreAspect) ;

real f(real t) {return 1/t ;}

scale(Log,Log) ;

draw(graph(f,0.1,10)) ;

//xlimits(1,10,Crop) ;
//ylimits(0.1,1,Crop) ;

dot(Label("(3,5)",align=S),Scale((3,5))) ;

xaxis("$x$",BottomTop,LeftTicks) ;
yaxis("$y$",LeftRight,RightTicks) ;
```


10^1
10^0
10^{-1}
y
10^{-1}
10^0
10^1
x
(3,5)

Graphique encore

```
import graph ;
unitsize(0.6cm,0.4cm) ; xlimits(-11,9) ; ylimits(-11,13) ;
real[]x={-10,-8,-4,-2,0,3,5,6,8} ;
real[]y={10,3,12,0,-10,0,8,7,12} ;
pen p=fontsize(6pt) ; pen pTick=gray(0.7) ;
arrowbar tempArrow =Arrow(HookHead,3bp) ;
xaxis(BottomTop,p,LeftTicks(extend =true,end=true,
                            Step=1,pTick=pTick)) ;
yaxis(LeftRight,p,RightTicks(extend=true,end=true,
                             Step=1,pTick=pTick)) ;
xequals(shift(3S+2W)*"$d$",0,-11,13,p=p+0.8bp+black,tempArrow) ;
yequals(shift(2S+4W)*"$t$",0,-11,9,p=p+0.8bp+black,tempArrow) ;
draw(graph(x,y,Hermite)) ;
```


d
t
13
12
11
10
9
8
7
6
5
4
3
2
1
0
−1
−2
−3
−4
−5
−6
−7
−8
−9
−10
−11
−11 −10 −9 −8 −7 −6 −5 −4 −3 −2 −1 0 1 2 3 4 5 6 7 8 9

3D

- Chemins/guides définis en 3D : par rapport à la 2D c'est juste une coordonnée supplémentaire
- Différentes projections possibles : orthographique, perspective, oblique
- Asymptote implémente une version 3D de l'algorithme de J. Hobby et D. Knuth pour le choix automatique (si l'utilisateur le veut) des points de contrôle
- La 3D donne beaucoup de complications du point de vue algorithmique, mathématique
- Surface 3D : carreaux de Bézier bicubiques (16 degrés de liberté) et NURBS (récent)
- Représentation de surface de type $z = f(x, y)$ avec (x, y) dans un pavé de $\mathbf{R}^2$: facette ou spline cubique (surface régulière)
- Représentation de surface paramétrique, $(x, y, z) = F(s, t)$ avec (s, t) dans un pavé de $\mathbf{R}^2$: facette ou spline cubique (surface régulière)

3D

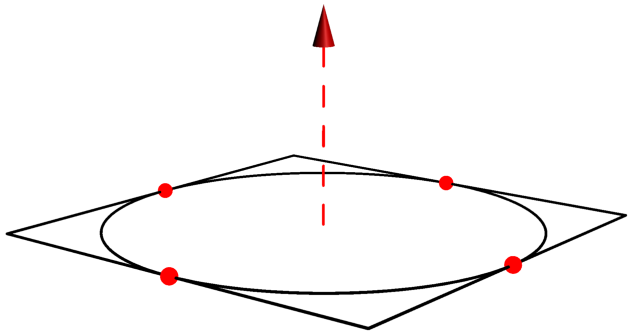
Asymptote gère la 3D de trois/quatre façons

- projection de la scène en 2D : très limitée du point de vue faces cachée/visibles
- Format PRC embarqué dans un fichier PDF. Uniquement lisible par Adobe Reader sous plate-forme Intel, format très compressé, vue dynamique
- Moteur de rendu OpenGL interne à Asymptote
- Moteur de rendu OpenGL fournit une image à une résolution prédéfinie par l'utilisateur

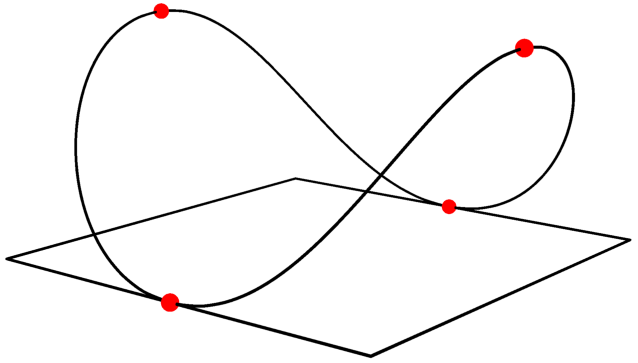
On peut superposer dans un document pdf, la figure 3D version PRC et une image bitmap (résolution à choisir). Ainsi les visionneuses pdf alternatives rendent au moins cette image bitmap.

Selle-3D interactive

- Un cercle unité dans le plan X–Y peut être tracé/rempli par :
  `(1,0,0)..(0,1,0)..(-1,0,0)..(0,-1,0)..cycle`



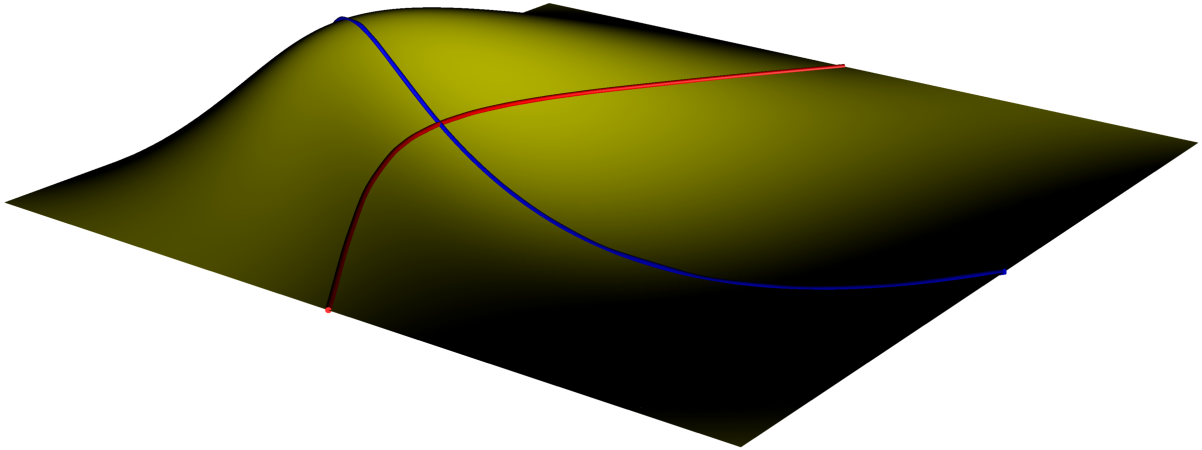
puis être transformé en une selle :
`(1,0,0)..(0,1,1)..(-1,0,0)..(0,-1,1)..cycle`



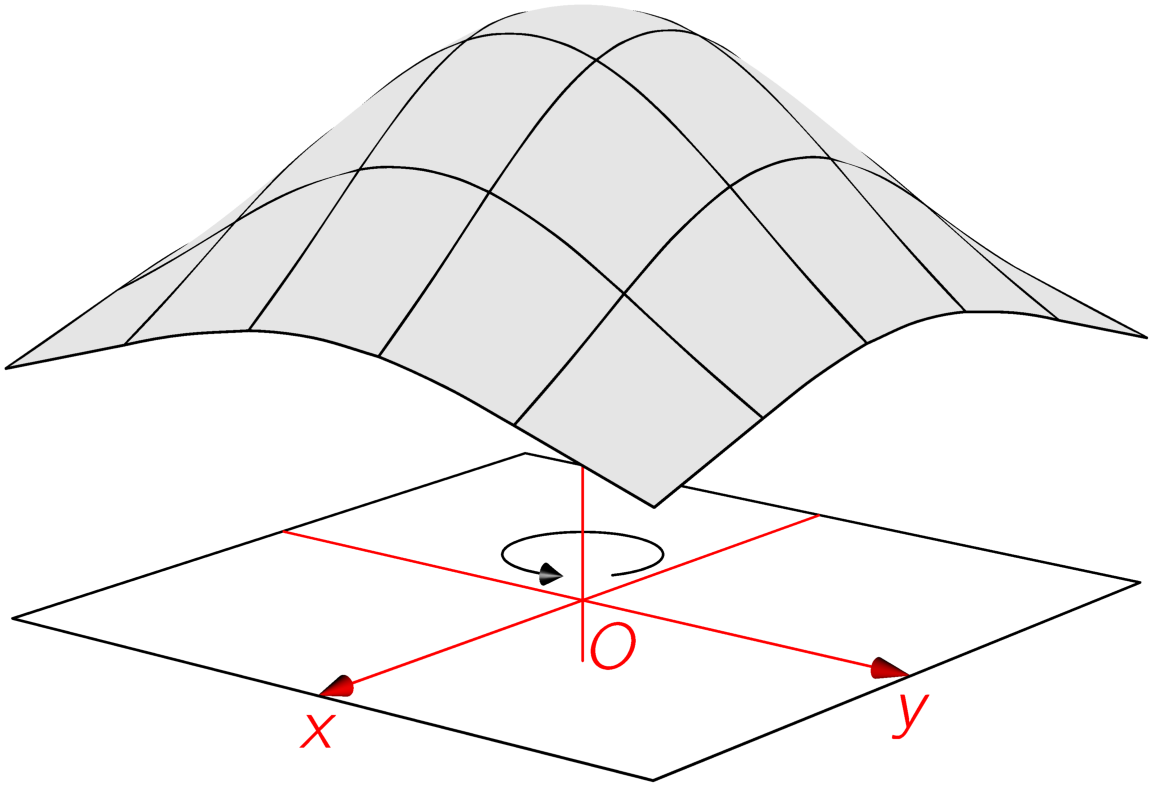
Carreau de Bézier bicubique

- 16 points de contrôle P_{ij} $(0 \leq i, j \leq 3)$, B_i polynôme de base de Bernstein de degré 3

$$S(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} B_i(u) B_j(v) P_{ij}$$



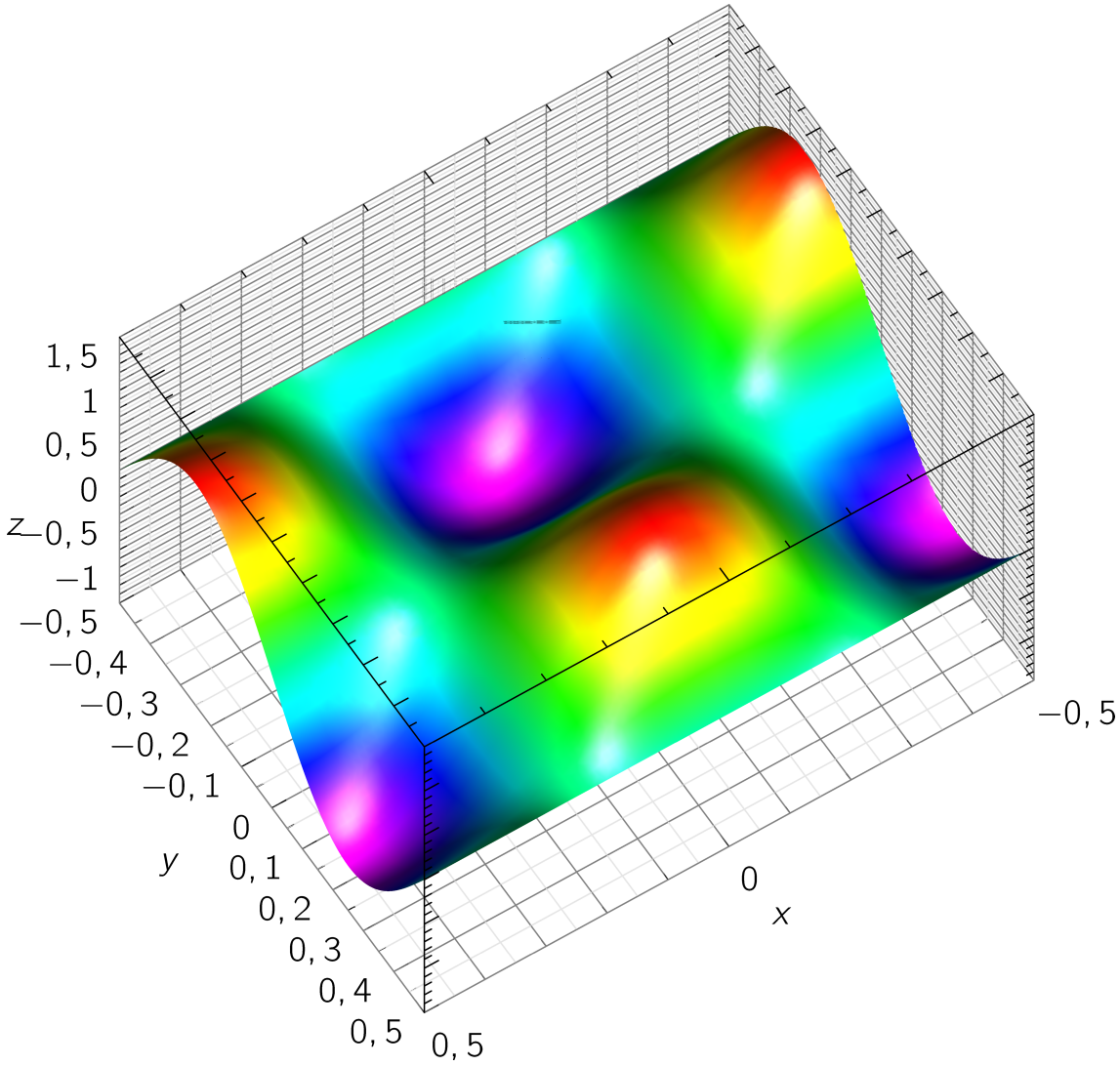
Surfaces 3D lisse (régulière)



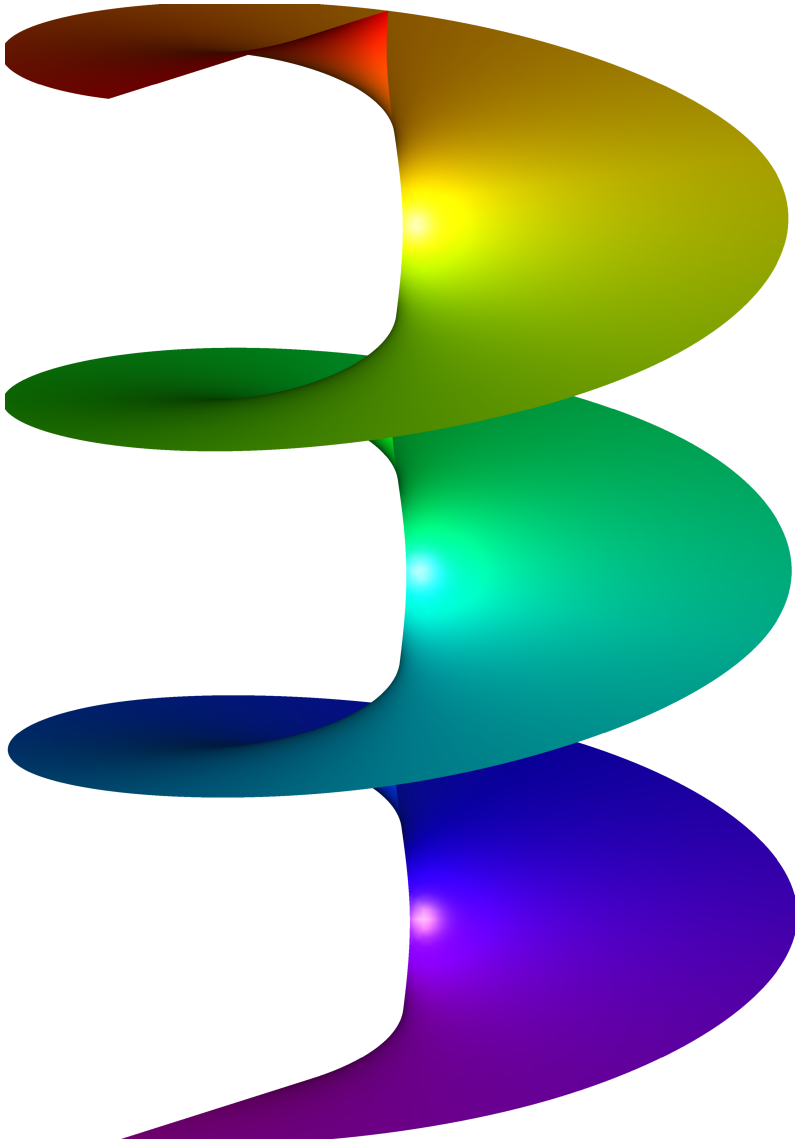
Galerie

$$\int_{-\infty}^{+\infty} e^{-\alpha x^2}\,dx = \sqrt{\frac{\pi}{\alpha}}$$

Surface



Surface paramétrique



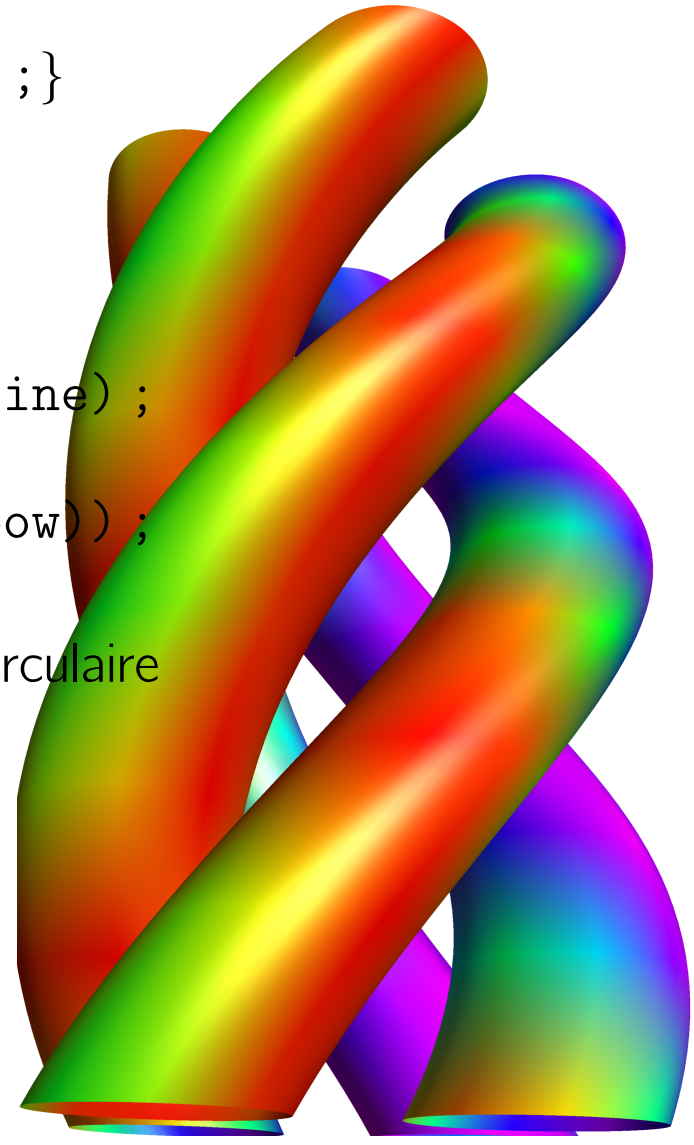
Surface paramétrique

- Le code de la surface de Riemann :

```
import graph3 ;
import palette ;
size(200,300,keepAspect=false) ;
//settings.nothin=true ;

currentprojection=orthographic(10,10,30) ;
currentlight=(10,10,5) ;
triple f(pair t) {return (exp(t.x)*cos(t.y),
                          exp(t.x)*sin(t.y),t.y) ;}

surface s=surface(f,(-4,-2pi),(0,4pi),8,16,Spline) ;
s.colors(palette(s.map(zpart),Rainbow())) ;
draw(s) ;
```

Tubes

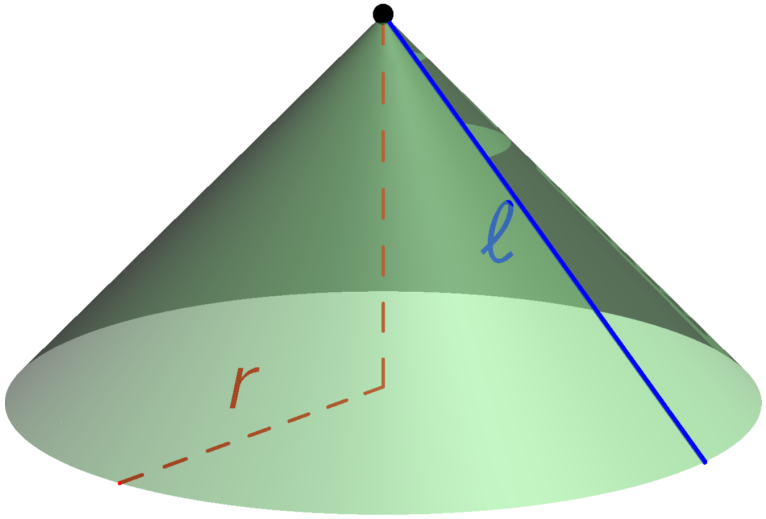
```
real w=0.4 ;
real f(triple t) {return sin(t.x) ;}
triple f1(pair t)
{return (cos(t.x)-2cos(w*t.y),
sin(t.x)-2sin(w*t.y),t.y) ;}
surface s1=
surface(f1,(0,0),(2pi,10),8,8,Spline) ;
pen[] Rainbow=Rainbow() ;
s1.colors(palette(s1.map(f),Rainbow)) ;
draw(s1) ;
```



- On peut aussi faire des tubes à base non circulaire

Solides

- Paquet `solide` pour quelques solides et solides de révolution




```
import solids ;
size(0,75) ;real r=1 ; real h=1 ;
revolution R=cone(r,h) ;
draw(surface(R),lightgreen+opacity(0.5)) ;
pen edge=blue+0.25mm ;
draw("$\ell$",(0,r,0)--(0,0,h),W,edge) ;
draw("$r$",(0,0,0)--(r,0,0),red+dashed) ;
draw((0,0,0)--(0,0,h),red+dashed) ;
dot(h*Z) ;
```