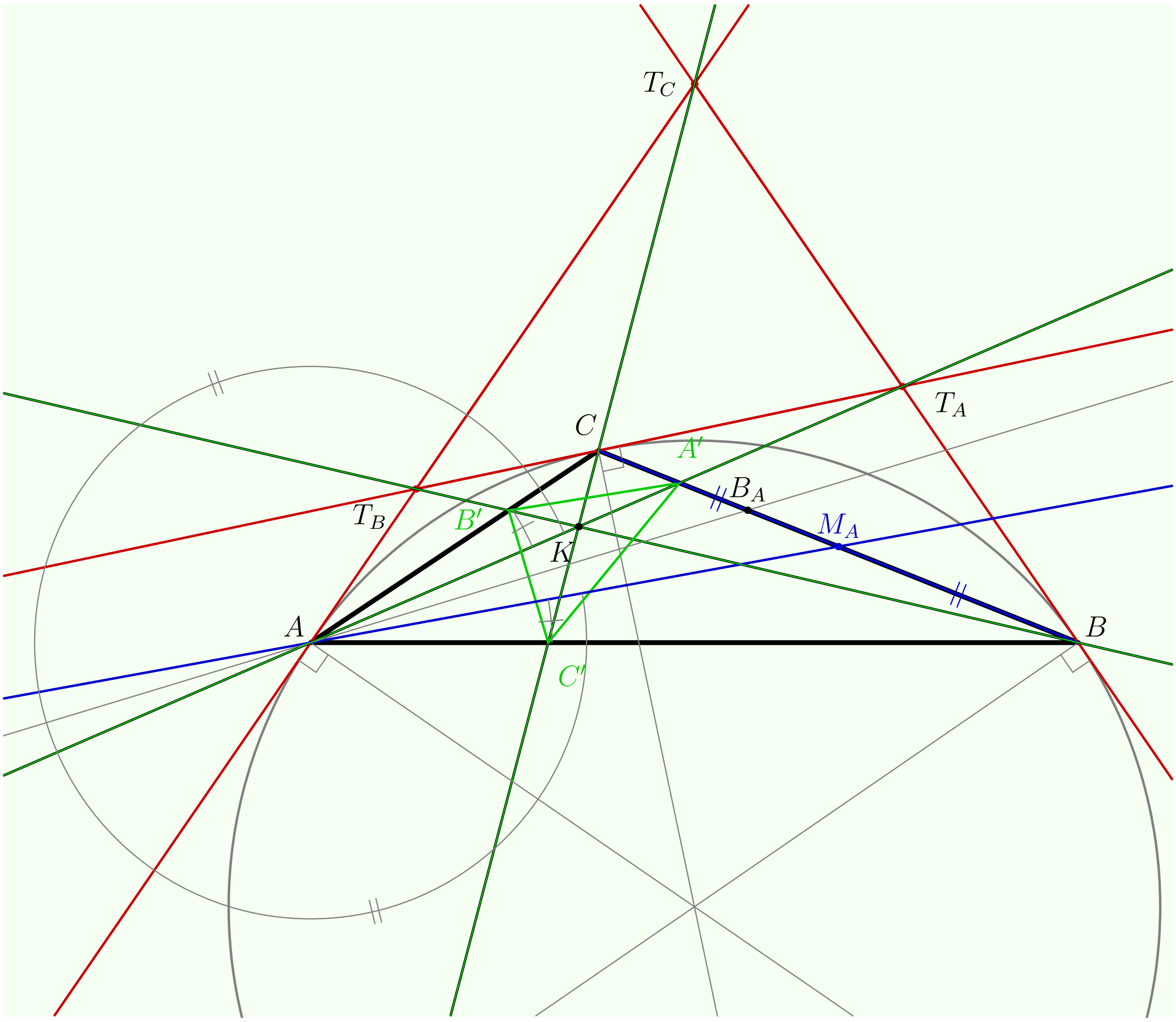
`geometry.asy`*

Géométrie euclidienne avec ASYMPTOTE

Philippe IVALDI

Compilé avec ASYMPTOTE version 2.14svn-r5318
le 7 décembre 2021



Résumé

Ce document décrit l'utilisation de l'extension *geometry.asy* qui facilite la création de figures de géométrie plane euclidienne en définissant de nouveaux types et de nouvelles routines pour le logiciel ASYMPTOTE.
Après avoir dressé la liste des nouveaux types accompagnés d'une brève description, nous les étudierons séparément et détaillerons les routines et opérateurs qui leurs sont associés.

Remerciements

Je remercie particulièrement les personnes suivantes :

— Olivier GUIBÉ pour son aide précieuse dans les algorithmes de mathématiques algébriques, ses encouragements et son écoute toujours attentive ;

— John BOWMAN et Andy HAMMERLINDL sans qui ASYMPTOTE n'existerait pas ;

— MB qui a testé l'extension pendant son développement ce qui a permis de corriger, d'améliorer et d'ajouter des fonctionnalités.

Table des matières

1. Introduction

1.1. Liste des types d'objets

L'extension *geometry.asy* définit de nombreux types d'objets couramment utilisés en géométrie plane Euclidienne ; ces objets peuvent être instanciés comme peut l'être un réel, instancié par le type `real`, ou un chemin, instancié par le type `path`.
Dans la suite de ce document il est important de distinguer l'objet de son type. Par exemple l'objet « équation quadratique à deux variables » est du type `bqe` (pour "Bivariate Quadratic Equation") et possède lui-même un objet nommé `a` de type `real[]` ; on y accède par `un_objet_bqe.a`.
Voici la liste exhaustive des types définis par l'extension *geometry.asy* :

coordsys instancie un repère Cartésien ; ce type est décrit dans la section Système de coordonnées et son utilisation est détaillée dans la section Points et vecteurs ;

point et **vector** point et vecteur relatifs à un repère Cartésien.
Ces types sont décrits dans la section Points et vecteurs dont la lecture peut être différée si l'on ne souhaite pas utiliser un autre repère que celui par défaut ; dans ce cas, on peut considérer que les types `point` et `vector` sont identiques au type `pair` ;

mass point massique relatif à un repère Cartésien (cf. Points massiques) ;

line et **segment** le type `line` instancie une droite ou une demi-droite ou un segment de droite. Le type `segment`, qui instancie un segment de droite, est un dérivé (un fils) du type `line` ; son existence ne se justifie que pour la clarté du

code.
Ces types seront décrits dans la section Droites, demi-droites et segments ;

conic instancie n'importe quelle conique (cf. Coniques).
Pour plus de lisibilité et d'optimisation du code [1] les types dérivés circle, ellipse, parabola, hyperbola, bqe (pour "Bivariate Quadratic Equation") sont aussi définis.

Il est ainsi conseillé de toujours utiliser le type de conique le plus restreint suivant l'usage qui doit en être fait ; il est tout à fait possible par la suite de convertir une conique particulière en une conique quelconque comme le montre l'exemple suivant :

```
ellipse un_cercle=circle((point)(0,0), 3);
...
conic une_conique=un_cercle;
...
```

arc instancie un arc d'ellipse (cf. Arcs) ;

abscissa instancie une abscisse sur une droite (au sens large) ou une conique (cf. Abscisses) ;

triangle instancie un triangle (cf. Triangles).
Les objets relatifs à un triangle, accessibles par `un_triangle.objet`, sont de type

side instancie un côté du triangle (cf. Côtés) ;

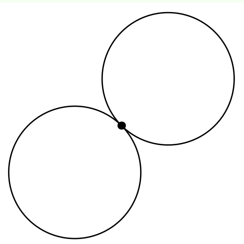
vertex instancie un sommet du sommet du triangle (cf. Sommets).

trilinear instancie des coordonnées trilinéaires relatives à un triangle (cf. Coordonnées trilinéaires).

1.2. Fonctionnement interne

Les calculs portant sur un objet instancié par un des types définis par l'extension *geometry.asy* s'effectuent d'après la nature même de l'objet, non d'après sa représentation graphique qui n'est finalement qu'un `path` ou un `pair`.

Ainsi le code suivant, qui marque l'intersection de deux cercles tangents, se compile en cinq fois moins de temps que le code équivalant utilisant le type `path` à la place du type `circle`.



```
import geometry;
size(3cm,0);
circle cle1=circle((point)(0,0), 1);
circle cle2=circle((point)(sqrt(2),sqrt(2)), 1);
draw(cle1); draw(cle2);
dot(intersectionpoints(cle1, cle2));
```

1.3. Index et exemples externes à ce document

Un index et une galerie d'exemples de toutes les routines, types, et opérateurs créés par l'extension *geometry.asy* permettent une exploration détaillée de ce module :

— index ordonné par nom de fonction ;

— index ordonné par type de fonction ;

— galerie d'exemples.

1. la détermination d'une tangente à un cercle diffère de celle d'une hyperbole

1.4. Conversion automatique des types (« casting »)

Les objets de types précédemment énumérés peuvent être traités par des routines qui leurs sont propres, celles définies par l'extension *geometry.asy*, ou par des routines natives d'ASYMPTOTE grâce à la conversion de type. Par exemple un cercle, de type `circle`, peut être tracé directement avec la routine standard `draw` grâce à la conversion automatisée `circle` vers `path` alors que le code `dot(un_cercle);` renverra un message d'erreur car la routine `dot` attend un type `path` **exactement** ; il faut alors forcer la conversion de type en écrivant `dot((path)un_cercle);`.

2. Systèmes de coordonnées

La lecture de cette section peut être différée si l'on ne souhaite pas utiliser un autre repère du plan que celui par défaut. Dans ce cas il suffit de savoir que l'extension *geometry.asy* utilise le type `point` à la place du type `pair` et que le type `vector` est équivalent au type `pair`.

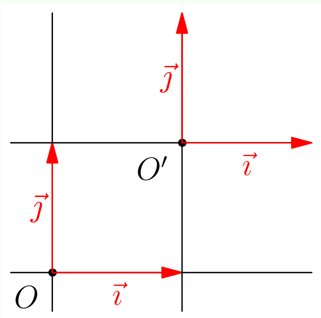
Les paragraphes suivants traitent des routines de base sur les repères Cartésiens, l'utilisation effective des repères étant détaillée dans la section Points et vecteurs.

2.1. Le type coordsys

L'extension *geometry.asy* permet de définir des objets dans un repère Cartésien du plan quelconque ; un tel repère est de type `coordsys`. Comme le montre l'exemple suivant :

— le repère par défaut est `defaultcoordsys`, c'est celui utilisé nativement par ASYMPTOTE ;

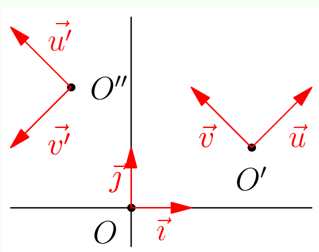
— le repère courant est `currentcoordsys` dont la valeur par défaut est `defaultcoordsys`.




```
import geometry;
size(4cm,0);
show(defaultcoordsys);
show("$O'$", shift((1,1))*currentcoordsys);
```

2.2. Définir un système de coordonnées

Pour définir un repère on peut utiliser la commande `cartesiansystem` ou appliquer une transformation à un repère existant comme illustré dans l'exemple suivant :




```
import geometry;
size(4cm,0);
coordsys R=cartesiansystem((2,1), i=(1,1), j=(-1,1));

show("$O'$","$\vec{u}$", "$\vec{v}$", R, xpen=invisible);
show("$O''$","$\vec{u'}$", "$\vec{v'}$",
     rotate(90)*R, xpen=invisible);
show(defaultcoordsys);
```

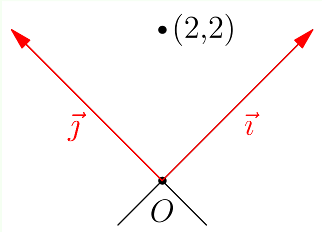
2.3. Changer un objet pair de repère

Les exemples de cette section sont donnés à titre indicatif, le moyen le plus efficace pour définir, modifier et convertir des coordonnées dans un repère étant d'utiliser le type `point` (voir Points et vecteurs).

Les deux principales routines pour définir ou convertir des coordonnées, de type `pair`, dans un repère sont en fait des opérateurs :

— `pair operator *(coordsys R, pair m);`

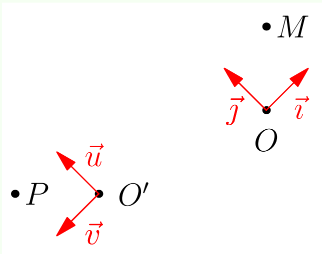
Permet de convertir les coordonnées de `m` données dans le repère `R` en coordonnées relatives au repère par défaut. Ainsi, dans l'exemple suivant, le point `M` a pour coordonnées $(0,5\,;\,0,5)$ dans `R` et $(2\,;\,2)$ dans `defaultcoordsys` :




```
import geometry;
size(4cm,0);
coordsys R=cartesiansystem((2,1), i=(1,1), j=(-1,1));
pair M=R*(0.5,0.5);
dot("", M);
show(R);
```

— `pair operator /( pair m, coordsys R);`

Permet de convertir les coordonnées de `m` données dans le repère par défaut en coordonnées relatives au repère `R`. Ainsi, dans l'exemple suivant, les points `M` et `P` ont les mêmes coordonnées dans `R` et `Rp` respectivement :




```
import geometry;
size(4cm,0);
coordsys R=cartesiansystem((2,1), i=(1,1), j=(-1,1));
coordsys Rp=cartesiansystem((-2,-1), i=(-1,1), j=(-1,-1));
pair M=R*(1,1);
dot("$M$", M);
pair P=Rp*(M/R);
dot("$P$", P);
show(R, xpen=invisible);
show("$O'$", "$\vec{u}$", "$\vec{v}$", Rp, xpen=invisible);
```

2.4. Autres routines

— `path operator *(coordsys R, path g);`

Autorise le code du type `coordsys*path` qui renvoie la reconstruction du chemin `g` comme si chaque nœud du chemin était donné dans le repère `R`.

— `coordsys operator *(transform t, coordsys R);`

Autorise le code `transform*coordsys`. Noter que `shiftless(t)` est appliqué à `R.i` et `R.j`.

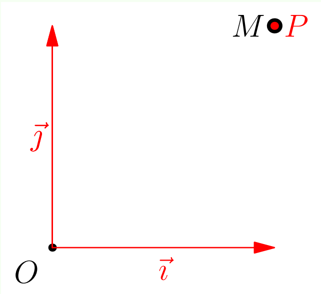
3. Points et vecteurs

3.1. Les points

3.1.1. Principes de base

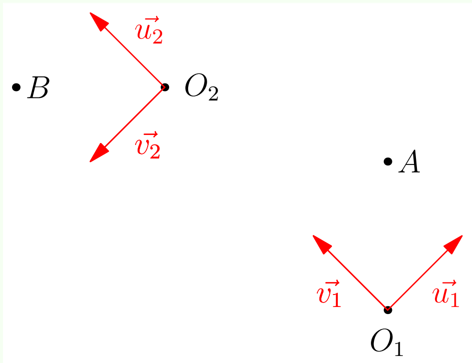
À la différence du type `pair` qui permet de repérer un point dans le repère par défaut, le type `point` permet de repérer un point dans n'importe quel repère Cartésien (voir coordsys) ; un objet de type `point` fait toujours référence au repère dans lequel il est défini.

Grâce au « casting », un `point` peut être globalement assimilé à un `pair` si l'on utilise seulement le repère par défaut. Ainsi, dans l'exemple suivant, le `pair M` et le `point P` marquent le même point :



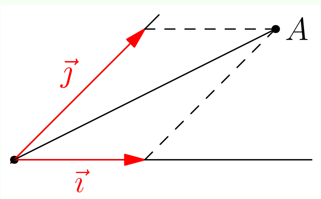

```
import geometry;
size(4cm,0);
show(currentcoordsys, xpen=invisible);
pair M=(1,1); dot("$M$", M, W, linewidth(2mm));
point P=(1,1); dot("$P$", P, red);
```

L'exemple suivant montre comment la modification du repère courant influe sur le « casting » `pair` vers `point` dans le cas du point `A` et comment définir un point dans un repère spécifique grâce à la routine `point point(coordsys R, pair m)` dans le cas du point `B`.



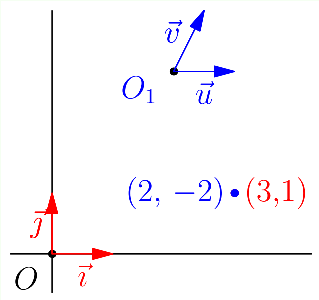

```
import geometry;
size(6cm,0);
currentcoordsys=cartesiansystem((3,0), i=(1,1), j=(-1,1));
show("$O_1$", "$\vec{u_1}$", "$\vec{v_1}$", currentcoordsys, xpen=invisible);
point A=(1,1);
dot("$A$", A);
coordsys Rp=rotate(90)*currentcoordsys;
show("$O_2$", "$\vec{u_2}$", "$\vec{v_2}$", Rp, xpen=invisible);
point B=point(Rp, (1,1));
dot("$B$", B);
```

import geometry ; size(6cm,0) ; currentcoordsys=cartesiansystem((3,0), i=(1,1), j=(-1,1)) ; show("O_1", "$\overrightarrow{u_1}$", "$\overrightarrow{v_1}$", currentcoordsys, xpen=invisible) ; point A=(1,1) ; dot("A", A) ; coordsys Rp=rotate(90)*currentcoordsys ; show("O_2", "$\overrightarrow{u_2}$", "$\overrightarrow{v_2}$", Rp, xpen=invisible) ; point B=point(Rp, (1,1)) ; dot("B", B) ; L'utilisation de la routine `point locate(pair m);` permet aussi de convertir directement, sans nommer de point, un `pair` en `point` :



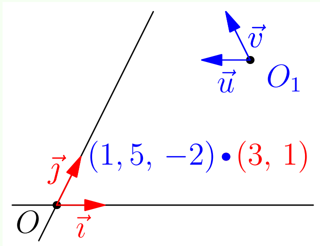

```
import geometry;
size(4cm,0);
currentcoordsys=cartesiansystem((3,0), (1,0), (1,1));
show("", currentcoordsys);
point A=(1,1);
dot("$A$", A); draw(locate(0)--A);
draw(locate((0,1))--A, dashed); draw(locate((1,0))--A, dashed);
```

L'exemple suivant montre comment convertir un type `pair` en type `point` de telle sorte qu'ils représentent le même point et ainsi obtenir ses coordonnées dans deux repères distincts.




```
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);
```

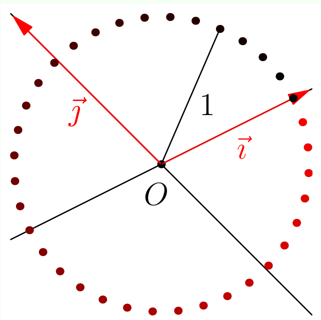
La routine `changecoordsys` permet de changer facilement le repère d'un `point` :




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

Comme pour le type `pair` les opérateurs `+`, `-`, `*`, `/` sont disponibles pour le type `point`. Il est à noter toutefois qu'une opération effectuée avec deux points définis relativement à des repères différents renvoie un `point` défini dans le repère par défaut `defaultcoordsys` ; l'utilisateur est alors prévenu de cette conversion automatique par un avertissement.

Pour repérer un point à l'aide de coordonnées polaires on peut utiliser la méthode `pair polar(real r, real angle)` d'un objet de type `coordsys` comme le montre l'exemple suivant :




```
import geometry;
size(4cm,0);
coordsys R=cartesiansystem((1,2), i=(1,0.5), j=(-1,1));
show(R);

for (int i=0; i < 360; i += 10) {
  pen p=(i/360)*red;
  dot(point(R, R.polar(1,radians(i))), p);
}

point A=point(R, R.polar(1,radians(40)));
draw((string)abs(A), R.O--A);
```

3.1.2. Autres routines

Maintenant que les routines de base concernant le type `point` sont définies, passons en revue les autres routines le concernant :

— `point origin(coordsys R=currentcoordsys);`

Retourne l'origine du repère `R` en tant que point.
La constante `point origin` est l'origine du repère par défaut.

— `point point(coordsys R, explicit point M, real m=M.m);`

Retourne le point de masse `m` dont les coordonnées relatives à `R` ont les mêmes valeurs que celles de `M`.
Ne pas confondre cette routine avec `changecoordsys`.

— `pair coordinates(point M);`

Renvoie les coordonnées de `M` relatives à son repère.

— `bool samecoordsys(bool warn=true ... point[] M);`

Renvoie `true` si et seulement si tous les points sont relatifs au même repère.
Si `warm` vaut true et si les repères sont différents un avertissement est généré.

—
```
point[] standardizecoordsys(coordsys R=currentcoordsys,
bool warn=true ... point[] M);
```

Renvoie les points, sous forme de tableau, relatifs au même repère `R`.
Si `warm` vaut `true` et si les repères sont différents un avertissement est généré.

— `pair[] operator cast(point[] P);`

« Casting » `point[]` vers `pair[]`.

— `pair locate(point P);`

Renvoie les coordonnées de `P` dans le repère par défaut.

— `point operator *(transform t, explicit point P);`

Définit `transform*point`.
Noter que les transformations `scale`, `xscale`, `yscale` et `rotate` sont définies relativement au repère par défaut ce qui n'est en général pas souhaité quand le repère courant est modifié.
Pour pallier cet inconvénient on peut utiliser les routines `scale(real,point)`, `xscale(real,point)`, `yscale(real,point)`, `rotate(real,point)`, `scaleO(real)`, `xscaleO(real)`, `yscaleO(real)` et `rotateO(real)` qui sont décrites dans la section Transformations (partie 1).

— `point operator *(explicit point P1, explicit pair p2);`

Définit `point*pair`.
`p2` est supposé représenter les coordonnées d'un point relativement au repère dans lequel `P1` est défini.

— `bool operator ==(explicit point M, explicit point P);`

Définit le test `M == N` qui renvoie `true` si et seulement si `MN < EPS`.

— `bool operator !=(explicit point M, explicit point N);`

Définit le test `M != N` qui renvoie `true` si et seulement si `MN >= EPS`.

— `real abs(coordsys R, pair m);`

Renvoie le module |`m`| relativement au repère `R`.

— `real abs(explicit point M)`

Renvoie le module |`M`| relativement au repère dans lequel `M` est défini.

— `real length(explicit point M)`

Renvoie le module |`M`| relativement au repère dans lequel `M` est défini.

— `point conj(explicit point M)`

Renvoie le conjugué de `M` relativement au repère dans lequel il est défini.

—
```
real degrees(explicit point M, coordsys,
R=M.coordsys, bool warn=true)
```

Renvoie l'angle de `M` en degrés relativement au repère `R`.

— `real angle(explicit point M, coordsys R=M.coordsys, bool warn=true)`

Renvoie l'angle de `M` en radians relativement au repère `R`.

— `bool finite(explicit point p)`

Même fonctionnement que `finite(pair m)` mais évite des calculs avec des coordonnées infinies.

— `real dot(point A, point B)`

Renvoie le produit scalaire `A.B` relativement au repère dans lequel est défini `A`.

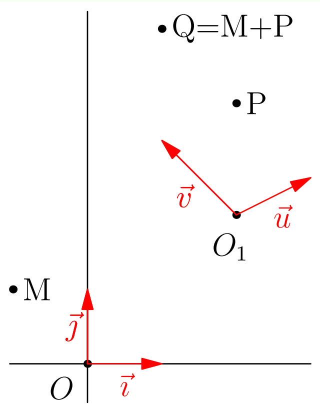
— `real dot(point A, explicit pair B)`

Renvoie le produit scalaire `A.B` après conversion des coordonnées de `A` dans le repère par défaut. `dot(explicit pair,point)` est aussi défini.

3.2. Les vecteurs

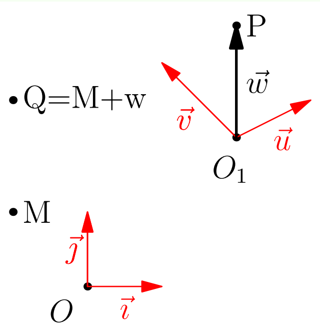
3.2.1. Principes de base

Dans l'exemple suivant les points `M` et `P` sont définis relativement à des repères distincts. Le point `Q`, somme de `M` et `P`, s'obtient en additionnant les coordonnées de `M` et de `P` **après leur conversion dans le repère par défaut** ; ainsi $\overrightarrow{\texttt{OQ}} = \overrightarrow{\texttt{OM}} + \overrightarrow{\texttt{OP}}$.



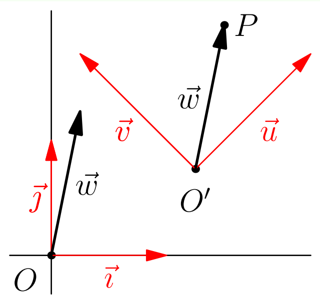

```
import geometry;
size(4cm,0);
show(currentcoordsys);
coordsys R=cartesiansystem((2,2), i=(1,0.5), j=(-1,1));
show("$O_1$", "$\vec{u}$", "$\vec{v}$", R, xpen=invisible);

point M=(-1,1); dot("M", M);

point P=point(R, (1,1)); dot("P", P);

point Q=M+P; dot("Q=M+P", Q);
```

En définissant le vecteur $\overrightarrow{\texttt{w}}$ par `vector w=vector(R, P.coordinates);` ou, plus simplement, par `vector w=P;` on a $\overrightarrow{\texttt{w}} = \overrightarrow{\texttt{O}_1\texttt{P}}$ et le code `point Q=M+w;` définit `Q` tel que $\overrightarrow{\texttt{OQ}} = \overrightarrow{\texttt{OM}} + \overrightarrow{\texttt{O}_1\texttt{P}}$; ce qui est le résultat souhaité. L'exemple suivant en est une illustration :



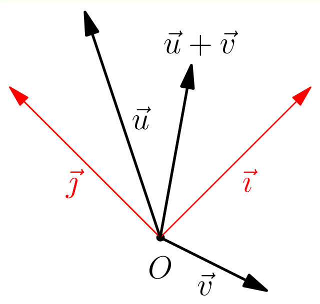

```
import geometry;
size(4cm,0);
show(currentcoordsys, xpen=invisible);
coordsys R=cartesiansystem((2,2), i=(1,0.5), j=(-1,1));
show("$O_1$", "$\vec{u}$", "$\vec{v}$", R, xpen=invisible);

point M=(-1,1); dot("M", M);
point P=point(R, (1,1)); dot("P", P);
vector w=P; show("$\vec{w}$", w, linewidth(bp), Arrow(3mm));
point Q=M+w; dot("Q=M+w", Q);
```

Un objet `u` de type `vector` fonctionne comme un objet de type `point` mais sa conversion en `pair` ou en `point M` relatif au repère par défaut s'effectue de telle façon que $\overrightarrow{\texttt{u}} = \overrightarrow{\texttt{OM}}$ comme le montre cette exemple qui utilise la routine `pair locate(vector v)` :

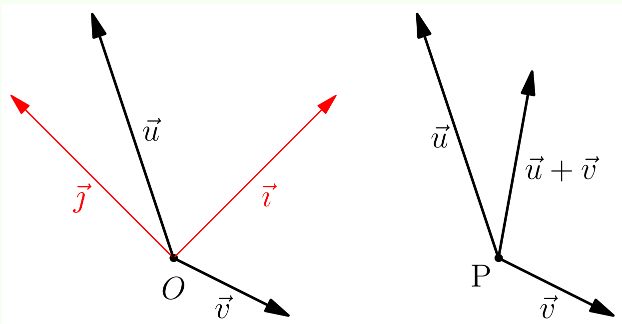



```
import geometry; size(4cm,0);
currentcoordsys=cartesiansystem((1.25,0.75), i=(1,1), j=(-1,1));
coordsys Rp=currentcoordsys; coordsys R=defaultcoordsys;
show(R);
show("$O'$","$\vec{u}$","$\vec{v}$", Rp, xpen=invisible);
point P=(0.75,0.5); dot("$P$",P); vector w=P;
pen bpp=linewidth(bp);
draw("$\vec{w}$",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

Les routines décrites pour le type `point` sont aussi disponibles pour le type `vector`. Voici quelques exemple simples qui en illustrent le comportement :




```
import geometry;
size(4cm,0);
pen bpp=linewidth(bp);
currentcoordsys=cartesiansystem((0,0), i=(1,1), j=(-1,1));
show(currentcoordsys, xpen=invisible);

vector u=(0.5,1), v=rotate(-135)*u/2;
show("$\vec{u}$", u, bpp, Arrow(3mm));
show("$\vec{v}$", v, bpp, Arrow(3mm));
show(Label("$\vec{u}+\vec{v}$",EndPoint), u+v, bpp, Arrow(3mm));
```



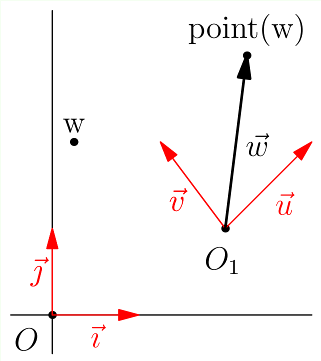

```
import geometry;
size(8cm,0);
pen bpp=linewidth(bp);
currentcoordsys=cartesiansystem((0,0), i=(1,1), j=(-1,1));
show(currentcoordsys, xpen=invisible);

vector u=(0.5,1), v=rotate(-135)*u/2;
show("$\vec{u}$", u, bpp, Arrow(3mm));
show("$\vec{v}$", v, bpp, Arrow(3mm));
point P=(1,-1); dot("P", P, SW);
draw(Label("$\vec{u}$",align=W), P--(P+u), bpp, Arrow(3mm));
draw("$\vec{v}$", P--(P+v), bpp, Arrow(3mm));
draw("$\vec{u}+\vec{v}$", P--(P+(u+v)), bpp, Arrow(3mm));
```

3.2.2. « Casting » vector/point

Par le jeu du « casting » un objet `point` peut être converti en objet `vector` et réciproquement un objet `vector` peut être converti en `point`. Au risque de se répéter, il faut insister sur le fait que la distinction entre `point` et `vector` existe seulement lorsque le repérage s'effectue dans un autre repère que le repère par défaut.

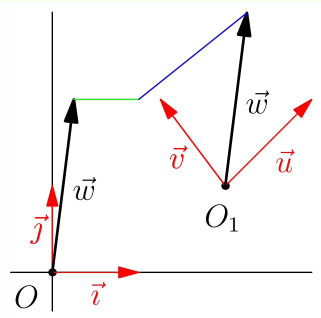
Remarquer dans l'exemple suivant la différence entre `dot(w)` et `dot(point(w))` ; dans le deuxième cas le vecteur est converti en point, celui « pointé par le vecteur » alors que dans le premier cas le vecteur est converti en `pair` comme expliqué

précédemment.
Noter enfin qu'il est possible d'écrire `point M=w;` au lieu de `point M=point(w);`.



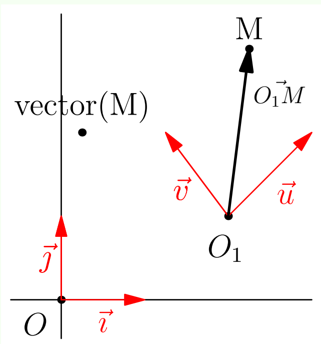

```
import geometry; size(4cm,0);
coordsys R=cartesiansystem((2,1), i=(1,1), j=(-0.75,1));
show("$O_1$", "$\vec{u}$", "$\vec{v}$", R, xpen=invisible);
show(currentcoordsys);

vector w=vector(R, (1,1));
show("$\vec{w}$", w, linewidth(bp), Arrow(3mm));
dot("w", w, N); dot("point(w)", point(w), N);
```

L'exemple suivant se passe donc de commentaires :



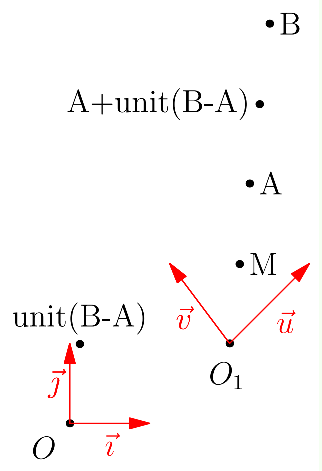

```
import geometry; size(4cm,0); pen bpp=linewidth(bp);
coordsys R=cartesiansystem((2,1), i=(1,1), j=(-0.75,1));
show("$O_1$", "$\vec{u}$", "$\vec{v}$", R, xpen=invisible);
show(currentcoordsys); vector w=vector(R, (1,1));
show("$\vec{w}$", w, bpp, Arrow(3mm));
show("$\vec{w}$", locate(w), bpp, Arrow(3mm));
draw((1,2)--locate(w), green);
draw((1,2)--point(w), blue);
```

Le pendant de la routine `point point(explicit vector)` est `vector vector(point)`




```
import geometry; size(4cm,0);
coordsys R=cartesiansystem((2,1), i=(1,1), j=(-0.75,1));
show("$O_1$", "$\vec{u}$", "$\vec{v}$", R, xpen=invisible);
show(currentcoordsys);
point M=point(R, (1,1)); dot("M", M, N);
dot("vector(M)", vector(M), N);
show(Label(scale(0.75)*"$\vec{O_1M}$", Relative(0.75)),
     M, linewidth(bp), Arrow(3mm));
```

Enfin une attention particulière doit être portée sur les routines `vector unit(point)` et `vector unit(vector)` qui renvoient toujours un objet `vector`. Ainsi, dans l'exemple suivant le comportement de `point P=unit(B-A` ne surprend pas alors que le comportement de `dot(unit(B-A))` peut laisser dubitatif...




```
import geometry;
size(4cm,0);
coordsys R=cartesiansystem((2,1), i=(1,1), j=(-0.75,1));
show("$O_1$", "$\vec{u}$", "$\vec{v}$", R, xpen=invisible);
show(currentcoordsys, xpen=invisible);

point A=point(R, (1,1)); dot("A", A); point B=point(R, (2,2));
dot("B", B); point M=unit(B-A); dot("M", M);
dot("unit(B-A)", unit(B-A), N);
dot("A+unit(B-A)", A+unit(B-A), W);
```

3.2.3. Autres routines

Comme déjà dit, toutes les routines s'appliquant au type `point` s'appliquent aussi au type `vector`. Mentionnons de plus la routine suivante dont le nom parle de lui-même : `bool collinear(vector u, vector v)`

4. Points massiques

4.1. Principes de base

Un objet de type `point` possède une masse à laquelle on peut accéder par `un_point.m` et une routine permet de calculer le barycentre d'un ensemble de points.
« Parfait ! »
Non... pas tout à fait :

— si l'on définit un point `M` par `M=P1+P2;`, le point `M` hérite de la somme des masses de `P1` et `P2`, ce qui est une bonne chose, mais si le point `M` est défini par `M=P/2` les coordonnées de `M` sont égales à la moitié de celles de `P`, ce qui est attendu et entendu, mais la masse reste inchangée. Ainsi, pour définir un point dont la masse est la moitié de celle d'un autre point il **faudrait** écrire :
```
point P=point((1,1), 3);// point de masse 3

point Q=P;
Q.m=P.m/2;
```
ce qui peut rapidement devenir pénible ;

— faire apparaître la masse de façon homogène dans toutes les figures lors de l'utilisation des routines `dot` et `label` risque fort de devenir aussi rapidement pénible.

Essentiellement pour ces deux raisons, l'extension *geometry.asy* définit un nouveau type, le type `mass`, dont le comportement se rapproche au mieux de ce que l'on peut attendre d'un « point massique ». Par exemple `mass M=objet_mass/2` définit le point massique `M` avec les mêmes coordonnées que `objet_mass` mais d'un poids moitié, le code `point M=objet_mass/2` a le même comportement mais le résultat est automatiquement converti en `point`.

Les routines `mass mass(explicit point)` et `point point(explicit mass)` permettent de basculer facilement entre les deux types `mass` et `point` ; la division de la masse d'un objet de type `point` peut alors se traiter de façon assez élégante :
```
point P=point((1,1), 3);// Point de masse 3

point Q=mass(P)/2;// Division de la masse, les coordonnées sont inchangées
```
La division des coordonnées d'un objet de type `mass` se traite de même :
```
mass P=mass((1,1), 3);// Masse de poids 3

mass Q=point(P)/2;// Division des coordonnées
```

4.2. Autres routines

Grâce au « casting » toutes les fonctionnalités du type `point` sont disponibles pour le type `mass` avec les nuances qui ont été mentionnées dans le paragraphe précédent. Voici la liste d'autres routines en rapport avec les points massiques :

— `mass mass(coordsys R, explicit pair p, real m)`

Renvoie un objet de type `mass` de poids `m` dont les coordonnées dans `R` sont `p`.

— `mass mass(point M, real m)`

Convertit le point `M` en masse de poids `m`.

— `point(explicit mass m)`

Convertit une masse de type `mass` en point, de type `point`.

— `mass mass(explicit point P)`

Convertit un point, de type `point`, en une masse de type `mass`.

— `mass masscenter(... mass[] M)`

Barycentre des masses `M`. Noter que, grâce au « casting » de `point[]` vers `mass[]`, cette routine fonctionne aussi avec comme paramètre un type `point[]`.

— `string defaultmassformat;`

Format par défaut utilisé pour construire les labels des masses.
Sa valeur par défaut est `"$\left(%L;%.4g\right)$"` dans laquelle %L sera remplacé par le label de la masse. Un exemple est sûrement plus parlant :

$\bullet (M;1)$ $\bullet M(2)$

```
import geometry;
size(4cm,0);
mass M=mass((0,0), 1); dot("M", M);

defaultmassformat="$%L(%.4g)$";
dot("M", M+(1,0));
```

— `string massformat(string format=defaultmassformat, string s, mass M)`

Renvoie une chaîne formatée par la commande `format` avec comme paramètre `format`, dans laquelle %L est remplacé par `s`, et le poids de `M`.

```
import geometry;

write(massformat(s="foo", mass((0,0),1000)));
// Renvoie $\left(foo;1000\right)$


write(massformat("%L\_%e", "foo", mass((0,0),1000)));
// Renvoie foo\_1\!\times\!10^{3\phantom{+}}
```

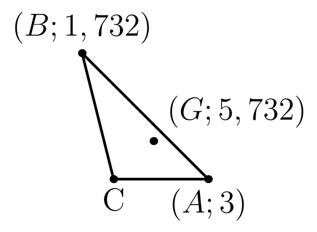
— `void label(picture pic=currentpicture, Label L, mass M, align align=NoAlign, string format=defaultmassformat, pen p=nullpen, filltype filltype=NoFill)`

Place en `M` le label renvoyé par `massformat(format,L,M)`.

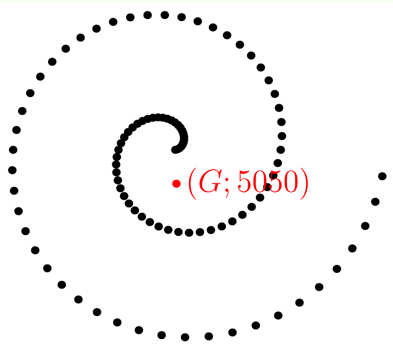
— `void dot(picture pic=currentpicture, Label L, mass M, align align=NoAlign, string format=defaultmassformat, pen p=currentpen)`

Place en `M` une marque de point et le label renvoyé par `massformat(format,L,M)`.

Pour terminer cette section voici trois exemples faisant intervenir quelques routines précédemment décrites.

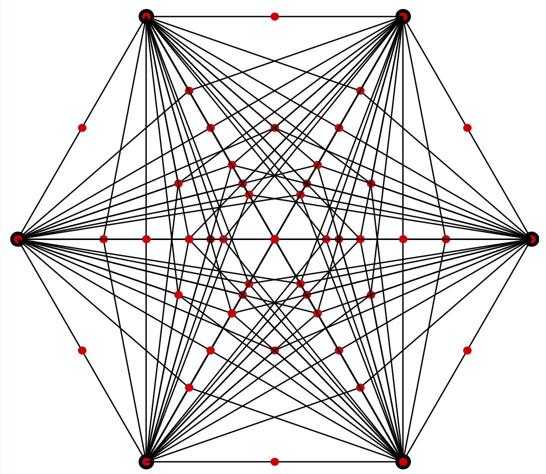



```
import geometry;
size(4cm,0);
mass A=mass((1,0), 3);
mass B=mass((0,1), sqrt(3));
point C=(0.25,0); // C inherits of a weight of 1 by default.

dot("B", B, N); dot("C", C, S); dot("A", A, S);
draw(A--B--C--cycle, linewidth(bp));
dot("G", masscenter(A,B,mass(C)), 2NE);
```




```
import geometry;
size(5cm,0);
int n=50;
mass[] M;
real m, step=360/n;
pair dir;
for (int i=0; i < 2*n; ++i) {
  dir=dir(i*step);
  m=i+1;
  M.push(mass(m*dir, m));
  dot(locate(M[i]));
}
dot("G",masscenter(... M), red);
```

L'exemple suivant montre comment l'on peut construire tous les barycentres partiels de `n` points, chaque barycentre étant relié aux points du système dont il est issu :



```
import geometry;
size(7cm,0);
int[][] parties(int n) {
  int[][] oi;

  void loop(int[] arr, int i) {
    oi.push(arr);
    for (int j=i; j < arr.length; ++j) {
      int[] tt=copy(arr);
      tt[j]=1;
      loop(tt, j+1);}}
  loop(sequence(new int(int n){return 0;}, n), 0);
  return oi;}

int n=6;
real step=360/n;
point[] M;

for (int i=0; i < n; ++i) {
  M[i]=mass(dir(i*step), 1);
  dot(M[i],linewidth(2mm));}

int[][] part=parties(n); int l=part.length;
point[][] group=new point[l][];

for (int i=0; i < l; ++i)
  for (int j=0; j < n; ++j)
    if(part[i][j] == 1) group[i].push(M[j]);

point[][] partbar=new point[l][2];

for (int i=0; i < l; ++i) {
  if(group[i].length > 0) partbar[i][0]=masscenter(...group[i]);
  for (int j=0; j < group[i].length; ++j)
    draw(group[i][j]--partbar[i][0]);
  if(group[i].length > 0) dot(partbar[i][0], 0.8*red);}
```

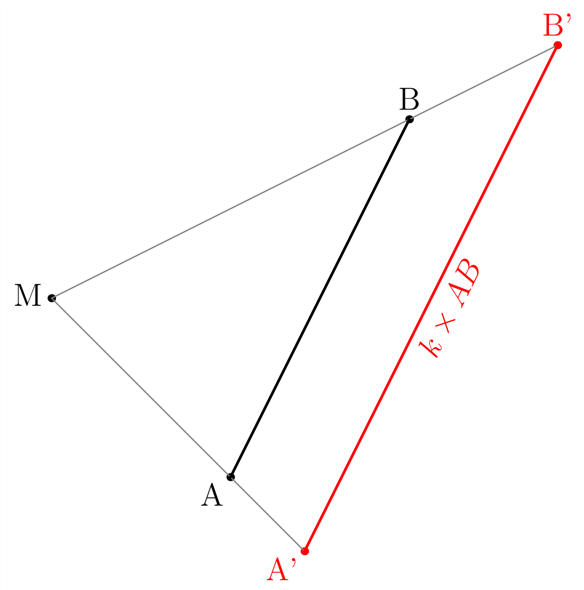
5. Transformations affines (partie 1)

En plus des transformations affines natives l'extension *geometry.asy* définit d'autres transformations. Certaines de ces transformations ont un comportement spécifique suivant le repère courant utilisé. Ainsi, afin de ne pas imposer au lecteur ne travaillant que dans le repère par défaut la description des routines spécifiques aux repères, cette section est divisée en deux sous-sections.

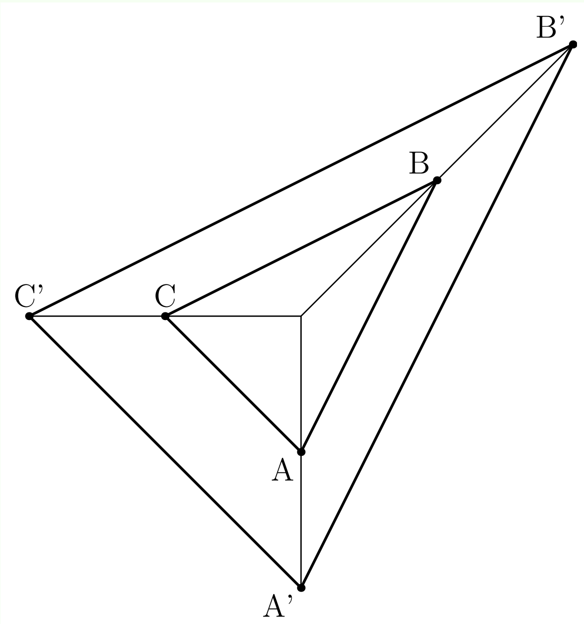
5.1. Transformations indépendantes du repère courant

— `transform scale(real k, point M)`

Homothétie de centre M et de rapport k.



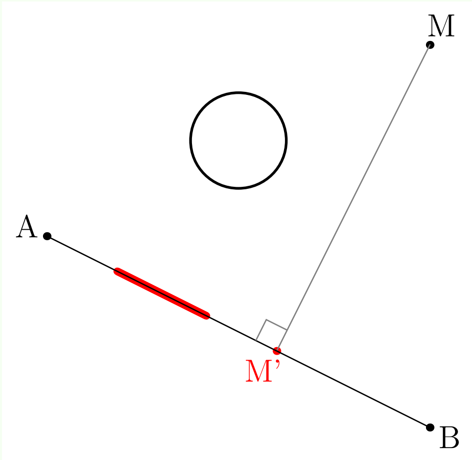

```
import geometry;
size(7.5cm,0);
pen bpp=linewidth(bp); real k=sqrt(2);

point A=(0,0); dot("A", A, SW);
point B=(1,2); dot("B", B, N);
point M=(-1,1);
dot("M", M, -dir(M--A,M--B));

point Ap=scale(k, M)*A;
dot("A'", Ap, SW, red);
point Bp=scale(k, M)*B;
dot("B'", Bp, N, red);

draw(M--Ap, grey); draw(M--Bp, grey);
draw(A--B, bpp);
draw(rotate(unit(Bp-Ap))*"$k\times AB$",
     Ap--Bp, bpp+red);
```




```
import geometry;
size(7.5cm,0);
pen bpp=linewidth(bp);
point A=(0,0); dot("A", A, SW);
point B=(1,2); dot("B", B, NW);
point C=(-1,1); dot("C", C,N);
path g=A--B--C--cycle; draw(g, bpp);
point M=(0,1);
path gp=scale(2, M)*g; draw(gp, bpp);
for (int i=0; i < 3; ++i) draw(M--point(gp,i));
dot("A'", point(gp,0), SW);
dot("B'", point(gp,1), NW);
dot("C'", point(gp,2), N);
```

— `transform projection(point A, point B)`

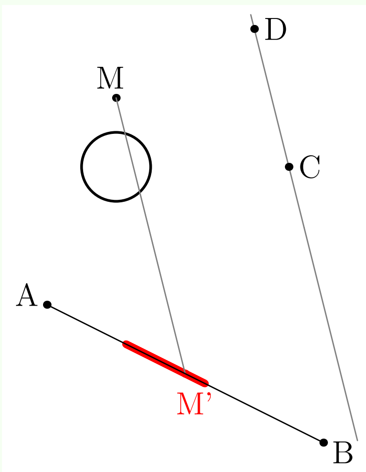
Projection orthogonale sur la droite `(AB)`.




```
import geometry;
size(6cm);
point A=(2,2); point B=(4,1); point M=(4,3);
path cle=shift(3,2.5)*scale(.25)*unitcircle;
draw(cle, linewidth(bp));

transform proj=projection(A,B);
point Mp=proj*M;

draw(proj*cle, 1mm+red);
dot("A", A, unit(A-B)); dot("B", B, unit(B-A));
dot("M", M, unit(M-Mp));
dot("M'", Mp, unit(Mp-M), red);
draw(M--Mp, grey); draw(A--B);
markrightangle(M,Mp,A, grey);
```

— `transform projection(point A, point B, point C, point D, bool safe=false)`

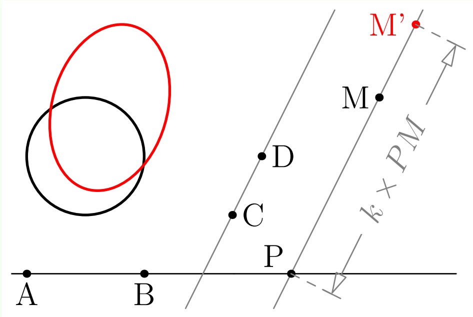
Projection sur la droite `(AB)` parallèlement à `(CD)`.
Si `safe` vaut `true` et `(AB)` est parallèle à `(CD)`, l'identité est renvoyée.
Si `safe` vaut `false` et `(AB)` est parallèle à `(CD)`, l'homothétie de centre `O` et de rapport infini est renvoyée.




```
import geometry;
size(6cm);
point A=(2,2); point B=(4,1); point C=(3.75,3);
point D=(3.5,4); point M=(2.5,3.5);
path cle=shift(2.5,3)*scale(0.25)*unitcircle;
draw(cle, linewidth(bp)); draw(line(C,D), grey);

transform proj=projection(A,B,C,D);
point Mp=proj*M;

draw(proj*cle, 1mm+red);
dot("A", A, unit(A-B)); dot("B", B, unit(B-A));
dot("C", C); dot("D", D); dot("M", M, unit(M-Mp));
dot("M'", Mp, 2*unit(Mp-M), red);
draw(M--Mp, grey); draw(A--B);
```

— `transform scale(real k, point A, point B, point C, point D, bool safe=false)`

Affinité de rapport `k`, d'axe `(AB)` et de direction `(CD)`. Si `safe` vaut `true` et `(AB)` est parallèle à `(CD)`, l'identité est renvoyée.
Si `safe` vaut `false` et `(AB)` est parallèle à `(CD)`, l'homothétie de centre `O` et de rapport infini est renvoyée.

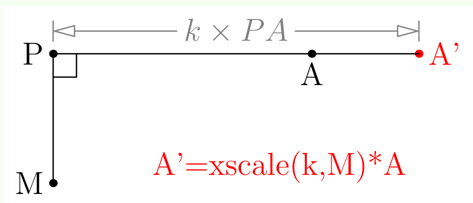



```
import geometry;
size(6cm,0);
pen bpp=linewidth(bp); real k=sqrt(2);
point A=(0,0), B=(2,0), C=(3.5,1);
point D=(4,2), M=(6,3);
path cle=shift(1,2)*unitcircle;
draw(cle, bpp);
draw(line(A,B));
draw(line(C,D), grey);

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);
```

5.2. Transformations dépendantes du repère courant

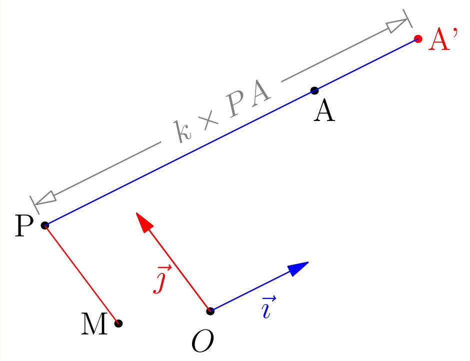
— `transform xscale(real k, point M)`

Affinité de rapport `k`, d'axe « l'axe passant par `M` et parallèle à `(Oy)` » et de direction `(Ox)`.




```
import geometry;
size(6cm,0);
real k=sqrt(2);
point A=(1,2); dot("A", A, S);
point M=(-1,1); dot("M", M, W);

point Ap=xscale(k, M)*A; dot("A'", Ap, red);
label("A'=xscale(k,M)*A", (0.75,1.125), red);

point P=extension(A, Ap, M, M+N);
dot("P", P, W); draw(M--P); draw(P--Ap);
perpendicularmark(P, dir(-45));
distance("$k\times PA$", P, Ap, -3mm, grey);
```

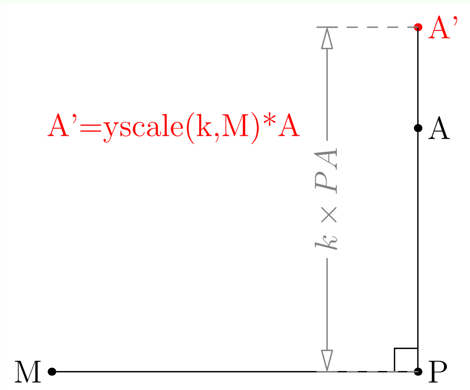
Le même exemple dans un repère quelconque :




```
import geometry;
size(6cm,0);
currentcoordsys=cartesiansystem((2,1), i=(1,0.5), j=(-0.75,1));
show(currentcoordsys, ipen=blue, jpen=red, xpen=invisible);

real k=sqrt(2);
point A=(2,1.25);
point M=(-0.75,0.25); dot("M", M, W);

point Ap=xscale(k, M)*A;
dot("A'", Ap, red); dot("A", A, I*unit(A-Ap));

point P=intersectionpoint(line(A,Ap), line(M,M+N));
dot("P", P, W); draw(M--P, red); draw(P--Ap, blue);
distance("$k\times PA$", P, Ap, -3mm, grey);
```

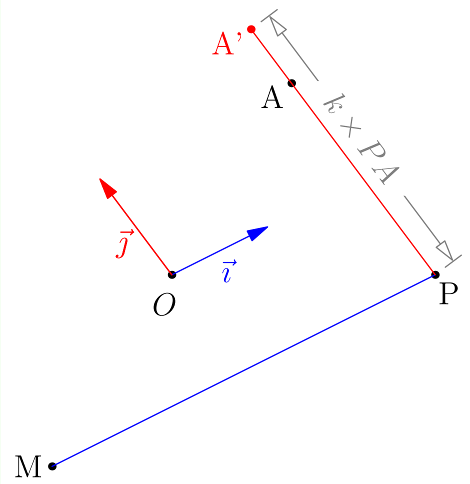
— `transform yscale(real k, point M)`

Affinité de rapport k, d'axe « l'axe passant par M et parallèle à (Ox) » et de direction (Oy).




```
import geometry;
size(6cm,0);
real k=sqrt(2);
point A=(2,1);
point M=(-1,-1); dot("M", M, W);

point Ap=yscale(k, M)*A;
dot("A'", Ap, red); dot("A", A, I*unit(A-Ap));
label("A'=yscale(k,M)*A", (0,1), red);

point P=intersectionpoint(line(A,Ap), line(M,M+E));
dot("P", P); draw(M--P); draw(P--Ap);
perpendicularmark(P, dir(135));
distance("$k\times PA$",P,Ap,-12mm,grey,grey+dashed);
```

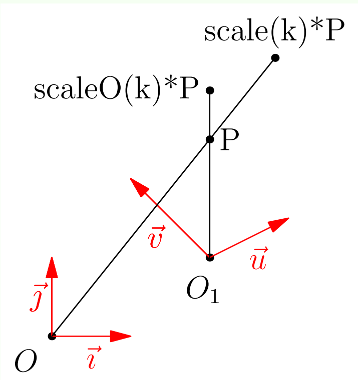
Le même exemple dans un repère quelconque :




```
import geometry;
size(6cm,0);
currentcoordsys=cartesiansystem((2,1), i=(1,0.5), j=(-0.75,1));
show(currentcoordsys, ipen=blue, jpen=red, xpen=invisible);

real k=sqrt(2);
point A=(2,1);
point M=(-2,-1); dot("M", M, W);

point Ap=yscale(k, M)*A;
dot("A'", Ap, -I*unit(A-Ap), red); dot("A", A, -I*unit(A-Ap));
point P=intersectionpoint(line(A,Ap), line(M,M+E));
dot("P", P, locate(unit(A-Ap))); draw(M--P, blue); draw(P--Ap, red);
distance("$k\times PA$", P, Ap, 3mm, grey);
```

— `transform scaleO(real x)`

Homothétie de rapport x et de centre « l'origine du repère courant ». Cette transformation est identique à `scale(x, origin())`. Dans l'exemple suivant, on notera la différence entre `scale(k)*P` et `scaleO(k)*P`.




```
import geometry; size(4.5cm,0);
currentcoordsys=cartesiansystem((2,1), i=(1,0.5), j=(-1,1));
show("$O_1$", "$\vec{u}$", "$\vec{v}$", currentcoordsys,
     xpen=invisible); show(defaultcoordsys, xpen=invisible);

real k=sqrt(2); point P=(1,1); dot("P", P);

point P1=scale(k)*P, P2=scaleO(k)*P; dot("scale(k)*P", P1, N);
dot("scaleO(k)*P", P2, W); draw((0,0)--locate(P1));
draw(origin()--P2);
```

— `transform xscaleO(real x)`

Identique à `xscale(x, origin())` (voir xscale(real,point)).

— `transform yscaleO(real x)`

Identique à `yscale(x, origin())` (voir yscale(real,point)).

— `transform rotateO(real angle)`

Identique à `rotate(angle, origin())`.

6. Droites, demi-droites et segments

6.1. Le type « line »

Un objet de type `line` représente une droite, une demi-droite ou un segment de droite suivant la valeur de ses propriétés `bool extendA,extendB;` accessible via `line.extendA` et `line.extendB`. La description complète des méthodes et propriétés du type `line` est accessible ici.

6.1.1. Droites définies par deux points, routines de base

— `line line(point A, bool extendA=true, point B, bool extendB=true)`

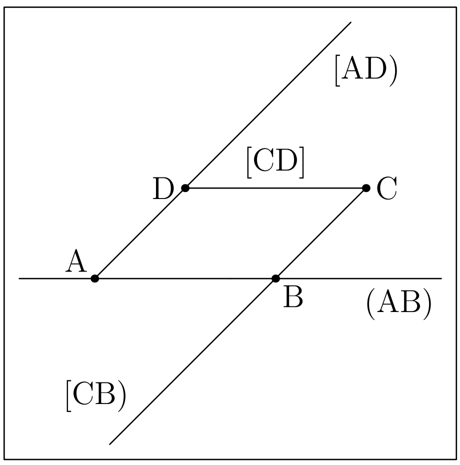
Définit un objet de type `line` passant par les deux points `A` et `B`, orientée de `A` vers `B`. Si `extendA` vaut `true` la « droite » s'étend du côté de `A`.
Un objet de type `line` appartient au repère dans lequel sont définis les deux points `A` et `B` mais si ces deux points sont définis dans des repères distincts, ils sont automatiquement redéfinis relativement au repère par défaut et un message d'avertissement est généré.

— `line Ox(coordsys R=currentcoordsys)`

Renvoie l'axe des abscisses du repère `R`.
La routine `line Oy(coordsys R=currentcoordsys)` est aussi définie.
Les constantes `Ox` et `Oy` sont les axes du repère par défaut.

—
```
void draw(picture pic=currentpicture, Label L="",
line l, bool dirA=l.extendA, bool dirB=l.extendB,
align align=NoAlign, pen p=currentpen,
arrowbar arrow=None, Label legend="", marker marker=nomarker)
```

Trace dans `pic` la « droite » `l` sans altérer la taille de l'image si le label est correctement positionné et la variable `linemargin` positive.
Les paramètres booléens `dirA` et `dirB` contrôlent la section infinie à afficher.
Noter qu'il est possible de contrôler la marge entre le bord de l'image et la trace des droites en modifiant la variable réelle `linemargin` dont la valeur par défaut est 0 ; dans le cas où cette marge est négative, la taille de l'image sera modifiée.




```
import geometry;
size(6cm,0);
linemargin=2mm;
point A=(0,0), B=(2, 0), C=(3,1), D=(1,1);
dot("A", A, NW); dot("B", B, SE); dot("C", C);
dot("D", D, W);

line AB=line(A, B);
line CB=line(C, false, B);
line CD=line(C, false, D, false);
line AD=line(A, false, D);

draw("(AB)", AB); draw("[CB)", CB);
draw(Label("[CD]",Relative(0.5),align=N), CD);
draw("[AD)", AD); draw(box((-1,-2),(4,3)));
```

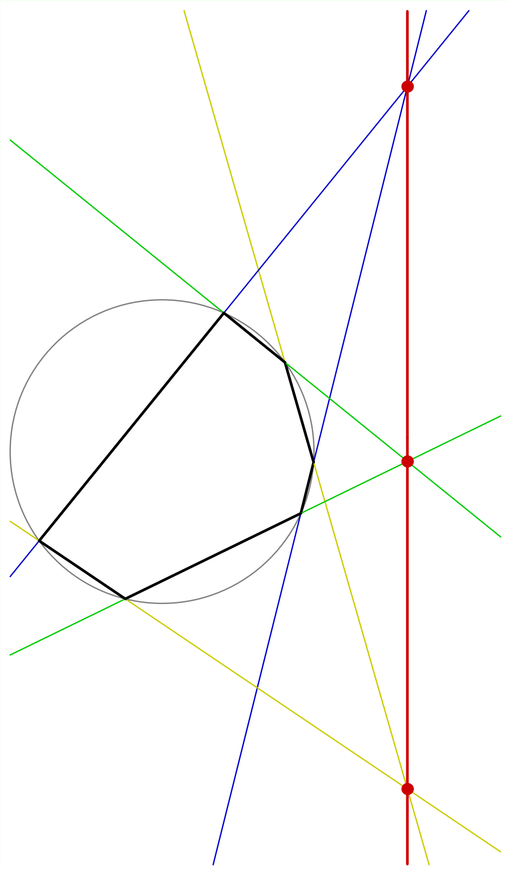
— `void show(picture pic=currentpicture, line l, pen p=red)`

Affiche dans `pic` les points qui ont servi à définir la droite `l` ainsi que le vecteur directeur et le vecteur normal.

— `point intersectionpoint(line l1, line l2)`

Renvoie le point d'intersection des objets `l1` et `l2`.
S'il n'y a aucun point d'intersection ou s'il y en a une infinité, c'est le point de coordonnée `(infinity,infinity)` qui est renvoyé.
Noter que si les deux droites sont définies relativement à des repères différents, le point d'intersection est défini relativement au repère pas défaut `defaultcoordsys` et un avertissement est généré.

L'exemple suivant est une illustration du célèbre théorème de PASCAL qui affirme que « Les points de concours des côtés opposés de tout hexagone inscrit dans un cercle sont alignés. »



```
import geometry;
size(6.5cm,0);
draw(unitcircle, grey);
point[] P;
real[] a=new real[]{0, 20, 60, 90, 240, 280};
real cor=24.0036303043338;
for (int i=0; i < 6; ++i) {
  P.push((Cos(a[i]-cor),Sin(a[i]-cor)));
}
pen[] p=new pen[] {0.8*blue, 0.8*yellow, 0.8*green};
line[] l;
for (int i=0; i < 6; ++i) {
  l.push(line(P[i],P[(i+1)%6]));
  draw(l[i], p[i%3]);
  draw(P[i]--P[(i+1)%6], linewidth(bp));
}
point[] inter;
for (int i=0; i < 3; ++i) {
  inter.push(intersectionpoint(l[i],l[(i+3)%6]));
  dot(inter[i], 1.5*dotsize()+0.8*red);
}
draw(line(inter[0],inter[1]), bp+0.8*red);
draw(box((-1,-2.722), (2.229,2.905)), invisible);
```

— `point[] intersectionpoints(line l, path g)`

Renvoie, sous forme de tableau, les points d'intersections de la « droite » `l` avec le chemin `g`.

6.1.2. Droites définies par équations

— `line line(coordsys R=currentcoordsys, real a, real b, real c)`

Renvoie la droite d'équation $\mathtt{a}x + \mathtt{b}y + \mathtt{c} = 0$ dans le repère `R`.

— `line line(coordsys R=currentcoordsys, real slope, real origin)`

Renvoie la droite de pente `slope` et d'ordonnée à l'origine `origin` donnés relativement au repère `R`.

6.1.3. Droites et parallélisme

— `line parallel(point M, line l)`

Renvoie la droite parallèle à `l` passant par `M`.

— `line parallel(point M, explicit vector dir)`

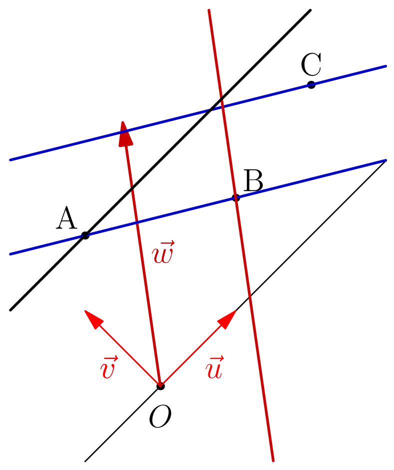
Renvoie la droite de vecteur directeur `dir` et passant par `M`.

— `line parallel(point M, explicit pair dir)`

Renvoie la droite de vecteur directeur `dir` donné dans le repère courant `currentcoordsys` et passant par `M`.

— `bool parallel(line l1, line l2, bool strictly=false)`

Renvoie `true` si `l1` et `l2` sont parallèles (strictement si `stricly` vaut `true`).

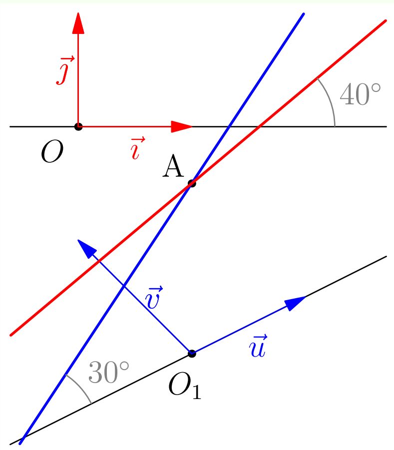



```
import geometry;
size(5cm,0);
coordsys R=cartesiansystem((1,-2), i=(1,1), j=(-1,1));
show("$O$","$\vec{u}$","$\vec{v}$", R, ypen=invisible);

pen bpp=linewidth(bp);
point A=(0,0), B=(2, 0.5), C=(3,2);
vector w=vector(R, (1.5,2)); line AB=line(A,B);

dot("A", A, NW); dot("B", B, NE); dot("C", C, N);
show("$\vec{w}$", w, bpp+0.8*red, Arrow(3mm));
draw(AB, bpp+0.8*blue);
draw(parallel(C, AB), bpp+0.8*blue);
draw(parallel(B, w), bpp+0.8*red);
draw(parallel(A, R.i), bpp);
draw(box((-1,-3),(4,3)), invisible);
```

6.1.4. Droites et angles

— `line line(real a, point A=point(currentcoordsys,(0,0)))`

Renvoie la droite passant par `A` et faisant un angle de `a` degrés avec l'axe des abscisses du repère dans lequel est défini `A`.
La routine `line(point,real)` est aussi définie.



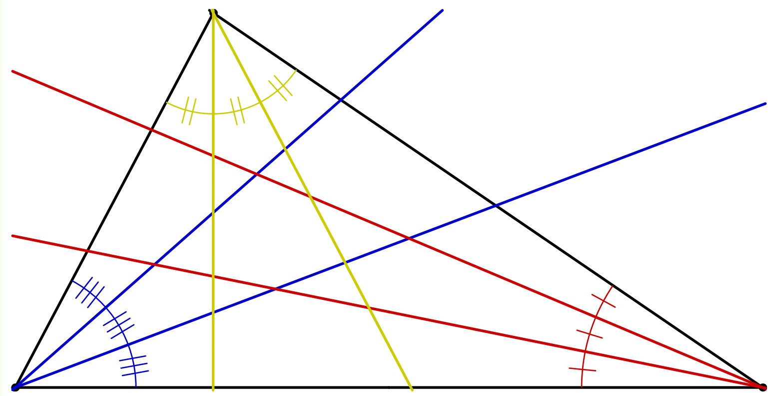

```
import geometry;
size(5cm,0);
coordsys R=cartesiansystem((1,-2), i=(1,0.5), j=(-1,1));
show("$O_{1}$", "$\vec{u}$", Label("$\vec{v}$", align=E),
     R, ipen=blue, ypen=invisible);
show(defaultcoordsys, ypen=invisible);
point A=point(R,(1,1)); dot("A", A, NW);

line l=line(A, 30);
draw(l, bp+blue);
markangle("$30^\circ$", Ox(R), l, grey);

A=changecoordsys(defaultcoordsys, A);
line ll=line(A, 40);
draw(ll, bp+red);
markangle("$40^\circ$", Ox, ll, grey);
draw(box((-0.6,-2.8), (2,-0.3)), invisible);
```

— `line bisector(line l1, line l2, real angle=0, bool sharp=true)`

Renvoie l'image de la bissectrice de l'angle formé par les droites orientées `l1` et `l2` par la rotation de centre « l'intersection de `l1` et `l2` » et d'angle `angle`.
Si le paramètre `sharp` vaut `true`, cette routine renvoie la bissectrice de l'angle aigu.
Noter que la droite renvoyée hérite du repère dans lequel est défini `l1`.

— `line sector(int n=2, int p=1, line l1, line l2, real angle=0, bool sharp=true)`

Renvoie l'image de la p-iéme droite qui partage l'angle formé par les droites orientées `l1` et `l2` en `n` parties égales par la rotation de centre « l'intersection de `l1` et `l2` » et d'angle `angle`.
Si le paramètre `sharp` vaut `true`, cette routine considère l'angle aigu.
Noter que la droite renvoyée hérite du repère dans lequel est défini `l1`. Ci-après, un exemple d'utilisation pour partager des angles en trois parties d'égales mesures.



```
import geometry;
size(10cm,0);
point A=(0,0), B=(3,0), C=(0.795,1.5);
dot(A); dot(B); dot(C);
pen pb=0.8*blue, pr=0.8*red, py=0.8*yellow, bpp=linewidth(bp);
line AB=line(A,B), AC=line(A,C), BC=line(B,C);
draw(AB, bpp); draw(AC, bpp); draw(BC, bpp);

line bA1=sector(3,AB,AC), bA2=sector(3,2,AB,AC);
line bB1=sector(3,AB,BC), bB2=sector(3,2,AB,BC);
line bC1=sector(3,AC,BC), bC2=sector(3,2,AC,BC);
draw(bA1, bpp+pb); draw(bA2, bpp+pb);
draw(bB1, bpp+pr); draw(bB2, bpp+pr);
draw(bC1, bpp+py); draw(bC2, bpp+py);

markangleradiusfactor *= 8;
markangle(BC, reverse(AB), pr, StickIntervalMarker(3,1,pr,true));
markangleradiusfactor /= 3;
markangle(reverse(AC), reverse(BC), py, StickIntervalMarker(3,2,py,true));
markangleradiusfactor *= 3/2;
markangle(AB, AC, pb, StickIntervalMarker(3,3,pb,true));
```

— `line perpendicular(point M, line l)`

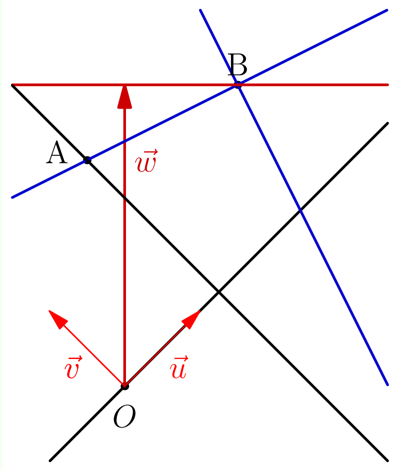
Renvoie la droite perpendiculaire à `l` passant par `M`.

— `line perpendicular(point M, explicit vector normal)`

Renvoie la droite passant par `M` et de vecteur normal `normal`.

— `line perpendicular(point M, explicit pair normal)`

Renvoie la droite passant par `M` et de vecteur normal `normal` donné dans le repère courant `currentcoordsys`.




```
import geometry;
size(5cm,0);
pen bpp=linewidth(bp);
coordsys R=cartesiansystem((0.5,-2), i=(1,1), j=(-1,1));
show("$O$","$\vec{u}$","$\vec{v}$", R, xpen=bpp,
     ypen=invisible);
point A=(0,1), B=(2,2);
vector w=vector(R, (2,2)); line AB=line(A,B);
dot("A", A, 2*dir(165)); dot("B", B, N);
show(Label("$\vec{w}$",Relative(0.75)), w, bp+0.8*red,
     Arrow(3mm));
draw(AB, bp+0.8*blue);
draw(perpendicular(B, AB), bp+0.8*blue);
draw(perpendicular(B, w), bp+0.8*red);
draw(perpendicular(A, R.i), bpp);
draw(box((-1,-3),(4,3)), invisible);
```

— `real angle(line l, coordsys R=coordsys(l))`

Renvoie la mesure de l'angle, en radians dans $]-\pi\,;\pi]$, par rapport au repère `R` de la droite orientée que représente `l`.

— `real degrees(line l, coordsys R=coordsys(l))`

Renvoie la mesure de l'angle, en degrés dans $[0\,;360[$, par rapport au repère `R` de la droite orientée que représente `l`.

— `real sharpangle(line l1, line l2)`

Renvoie la mesure de l'angle aigu orienté, en radians dans $]-\frac{\pi}{2}\,;\frac{\pi}{2}]$, formé par `l1` et `l2`.

— `real sharpdegrees(line l1, line l2)`

Renvoie la mesure de l'angle aigu orienté, en degrés dans $]-90\,;90]$, formé par `l1` et `l2`.

— `real angle(line l1, line l2)`

Renvoie la mesure de l'angle orienté, en radians dans $]-\pi\,;\pi]$, formé par les droites orientées représentées par `l1` et `l2`.

— `real degrees(line l1, line l2)`

Renvoie la mesure de l'angle orienté, en degrés dans $]-180\,;180]$, formé par les droites orientées représentées par `l1` et `l2`.

6.1.5. Droites et opérateurs

— `line operator *(transform t, line l)`

Autorise le code `transform*line`.

— `line operator /(line l, real x)`

Autorise le code `line/real`.
Renvoie la « droite » passant par `l.A/x` et `l.B/x`.
`line operator *(real x, line l)` est aussi défini.

— `line operator *(point M, line l)`

Autorise le code `point*line`.
Renvoie la « droite » passant pas `unit(M)*l.A` et `unit(M)*l.B`.

— `line operator +(line l, vector u)`

Autorise le code `line+vector`.
Renvoie l'image de `l` par la translation de vecteur `u`.
`line operator -(line l, vector u)` est aussi défini.

—
```
line[] operator ^^(line l1, line l2)
```
Autorise le code `line^^line`.
Renvoie le tableau `new line[] l1,l2`.

—
```
bool operator ==(line l1, line l2)
```
Autorise le test `line == line`.
Renvoie `true` si et seulement si `l1` et `l2` représente la même droite.

—
```
bool operator !=(line l1, line l2)
```
Autorise le test `line != line`.
Renvoie `false` si et seulement si `l1` et `l2` représente la même droite.

—
```
bool operator @(point m, line l)
```
Autorise le code `point @ line`.
Renvoie `true` si et seulement si le point `M` appartient à l'objet `l`.

6.1.6. Autres routines

Aux routines décrites dans cette section s'ajoutent des routines pour récupérer l'abscisse d'un point appartenant à un objet de type `line`.

—
```
void draw(picture pic=currentpicture,Label[] L=new Label[], line[] l,
align align=NoAlign, pen[] p=new pen[],
arrowbar arrow=None,
Label[] legend=new Label[], marker marker=nomarker)
```
Dessine chacune des droites représentées par `line[] l` avec le stylo correspondant `pen[] p`.
Si `p` n'est pas spécifié le stylo courant est utilisé.

—
```
void draw(picture pic=currentpicture,Label[] L=new Label[], line[] l,
align align=NoAlign, pen p,
arrowbar arrow=None,
Label[] legend=new Label[], marker marker=nomarker)
```
Dessine chacune des droites représentées par `line[] l` avec le même stylo `p`.

—
```
real distance(point M, line l)
```
Renvoie la distance de `M` à `l`.
`real distance(line l,point M)` est aussi défini.

—
```
bool sameside(point M, point P, line l)
```
Renvoie `true` si et seulement si `M` et `P` sont du même côté de `l`.

—
```
point[] sameside(point M, line l1, line l2)
```
Renvoie un tableau composé de deux points : le premier est le projeté de `M` sur `l1` parallèlement à `l2` et le second est le projeté de `M` sur `l2` parallèlement à `l1`.

—
```
coordsys coordsys(line l)
```
Renvoie le repère dans lequel est défini `l`.

—
```
line changecoordsys(coordsys R, line l)
```
Renvoie la « droite » représentée par `l` relativement au repère `R`.

—
```
line reverse(line l)
```
Renvoie la droite représentée par `l` avec une orientation contraire à celle de `l`.

—
```
line extend(line l)
```
Renvoie la droite portée par `l` qui, rappelons le, peut être une demi-droite ou un segment de droite.

—
```
line complementary(explicit line l)
```
Renvoie la demi-droite complémentaire de `l` ; cette routine ne fonctionne que si `l` représente effectivement une demi-droite.

— `bool concurrent(... line[] l)`

Renvoie `true` si et seulement si les droites représentées par `line[] l` sont concourantes.

— `bool perpendicular(line l1, line l2)`

Renvoie `true` si et seulement si les droites représentées par `l1` et `l2` sont perpendiculaires.

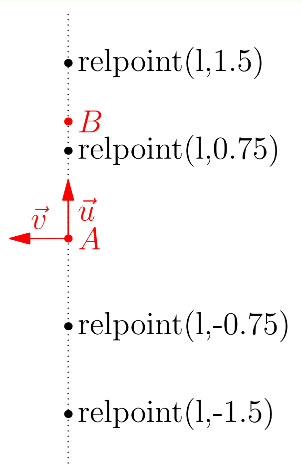
— `bool perpendicular(line l1, line l2)`

— `point point(line l, real x)`

Retourne le point entre `l.A` et `l.B` comme le ferait `point(l.A--l.B,x)`.

— `point relpoint(line l, real x)`

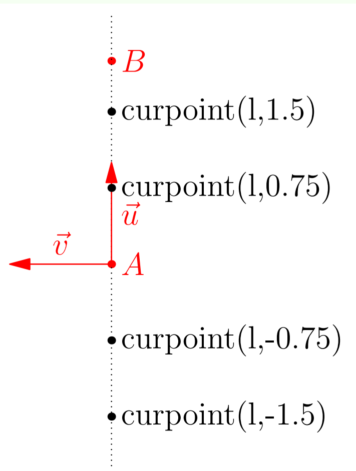
Retourne le point d'abscisse relative `x` donnée par rapport au segment [AB]. Autrement dit `relpoint(l,x)` renvoie `l.A+x*vector(l.B-l.A)`.




```
import geometry;
size(0,6cm);

point A=(0,0), B=(0,2);
line l=line(A,B); show(l);

dot("relpoint(l,0.75)", relpoint(l,0.75));
dot("relpoint(l,-0.75)", relpoint(l,-0.75));
dot("relpoint(l,1.5)", relpoint(l,1.5));
dot("relpoint(l,-1.5)", relpoint(l,-1.5));

addMargins(bmargin=5mm);
```

— `point curpoint(line l, real x)`

Retourne le point d'abscisse `x` donnée par rapport au repère $(\texttt{l.A}\,;\overrightarrow{\texttt{l.u}})$. Autrement dit `curpoint(l,x)` renvoie `l.A+x*unit(l.B-l.A)`.




```
import geometry;
size(0,6cm);

point A=(0,0), B=(0,2);
line l=line(A,B); show(l);

dot("curpoint(l,0.75)", curpoint(l,0.75));
dot("curpoint(l,-0.75)", curpoint(l,-0.75));
dot("curpoint(l,1.5)", curpoint(l,1.5));
dot("curpoint(l,-1.5)", curpoint(l,-1.5));

addMargins(bmargin=5mm);
```

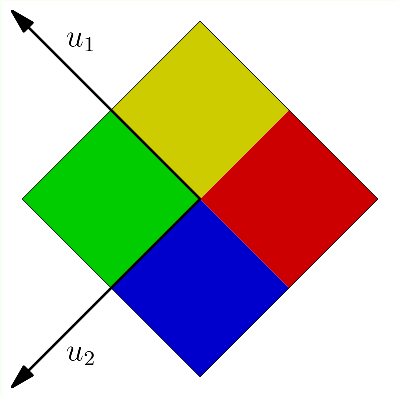
6.1.7. Droites et marqueurs

—
```
void markangle(picture pic=currentpicture,
Label L="", int n=1, real radius=0, real space=0,
line l1, line l2,
arrowbar arrow=None, pen p=currentpen,
margin margin=NoMargin, marker marker=nomarker)
```

Marque par n arcs de cercle l'angle orienté formé par les « droites » l1 et l2. Les arcs sont dessinés dans le sens trigonométrique si radius est positif ou nul, dans le sens horaire sinon.
Se reporter à « cette figure » pour un exemple.

—
```
void perpendicularmark(picture pic=currentpicture, line l1, line l2,
real size=0, pen p=currentpen, int quarter=1,
margin margin=NoMargin, filltype filltype=NoFill)
```

Marque un angle droit au point d'intersection de l1 et l2 dans le quarterième quart de plan compté dans le sens trigonométrique, le premier étant celui formé par les vecteurs l1.u et l2.u.




```
import geometry;
size(5cm,0);
transform t=rotate(135);
line l1=t*line((0,0),E); line l2=t*line((0,0),N);

perpfactor *=5.5;
perpendicularmark(l1,l2, Fill(0.8*green));
perpendicularmark(l1,l2, quarter=2, Fill(0.8*blue));
perpendicularmark(l1,l2, quarter=3, Fill(0.8*red));
perpendicularmark(l1,l2, quarter=4, Fill(0.8*yellow));

pen bpp=linewidth(bp); position pos=Relative(0.75);
show(Label("$u_1$",pos), l1.u, bpp, Arrow(3mm));
show(Label("$u_2$",pos,align=SE), l2.u, bpp, Arrow(3mm));
show("", -l1.u, invisible); show("", -l2.u, invisible);
```

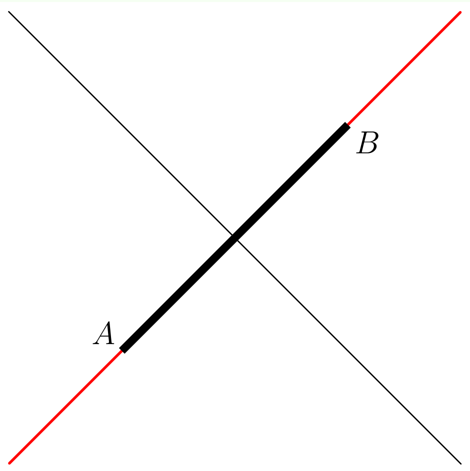
6.2. Le type « segment »

Comme déjà mentionné dans l'introduction, le type segment, qui instancie un segment de droite, est un dérivé (un fils) du type line. Par le jeu du « casting », pratiquement toutes les routines applicables à un objet de type line s'appliquent aussi à un objet de type segment et réciproquement.

Il est cependant important de noter que, lors du tracé d'un segment, la valeur de la variable addpenline est ajoutée au stylo utilisé. Par défaut cette variable a pour valeur squarecap, afin d'avoir les extrémités droites, ce qui rend l'affichage d'un segment en pointillé inefficient.
Pour contourner ce problème il y a trois solutions :

1. écrire draw(un_segment, roundcap+dotted); au lieux de draw(un_segment, dotted); ;
2. affecter la valeur nullpen à addpenline.
3. contacter l'auteur de l'extension *geometry.asy* pour lui faire connaître son désaccord quant à la valeur par défaut de addpenline ;

Enfin, tout comme les types point et mass sont interchangeables, les objets de types line et segment peuvent être converti de l'un vers l'autre en écrivant par exemple segment s=un_obj_line; ou draw(segment(un_obj_line)); ou encore draw(line(un_obj_segment)); comme le montre l'exemple suivant :




```
import geometry;
size(6cm,0);
point A=SW, B=NE;
label("$A$", A, NW); label("$B$", B, SE);

line l=line(A,B);
draw(l, bp+red);

segment s=l;
draw(s, linewidth(3bp));
draw(line(rotate(90,midpoint(s))*s));
draw(box(2*A,2*B), invisible);
```

En dehors des routines définies pour les objets de type `line` voici d'autres routines spécifiques aux objets de type `segment` :

— `segment segment(point A, point B)`

Renvoie le segment de droite d'extrémités `A` et `B`.

— `point midpoint(segment s)`

Renvoie le milieu du segment `s`.

— `line bisector(segment s, real angle=0)`

Renvoie l'image de la médiatrice de `s` par la rotation de centre « le milieu de `s` » et d'angle `angle`.

— `line[] complementary(explicit segment s)`

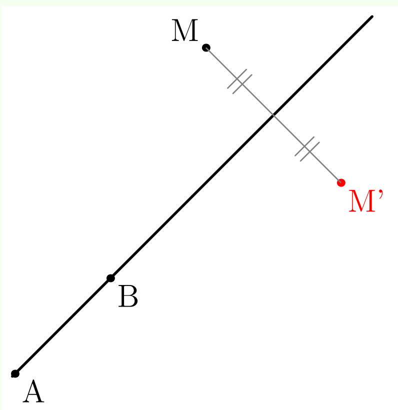
Renvoie sous forme de tableau les deux demi-droites de support `s` et d'extrémités respectives `s.A` et `s.B`.

7. Transformations affines (partie 2)

Certaines transformations décrites dans le section Transformations affines (partie 1), définies à partir de points, peuvent aussi être définies à partir de droites.

— `transform reflect(line l)`

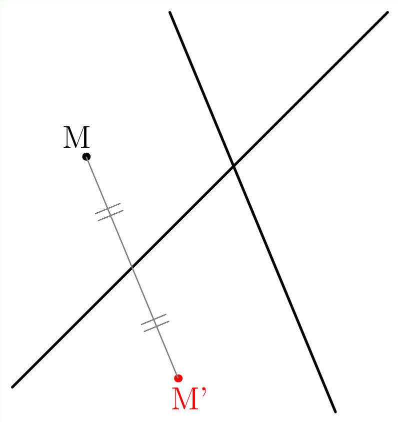
Renvoie la réflexion par rapport à `l`.




```
import geometry;
size(5cm,0);
point A=origin, B=NE, M=2*B+N;
dot("A", A, I*unit(A-B)); dot("B", B, I*unit(A-B));

line AB=line(A,B);
draw(AB, linewidth(bp));
transform reflect=reflect(AB);

point Mp=reflect*M;
dot("M",M, unit(M-Mp)); dot("M'", Mp, unit(Mp-M), red);
draw(segment(M,Mp), grey, StickIntervalMarker(2,2,grey));
```

— `transform reflect(line l1, line l2, bool safe=false)`

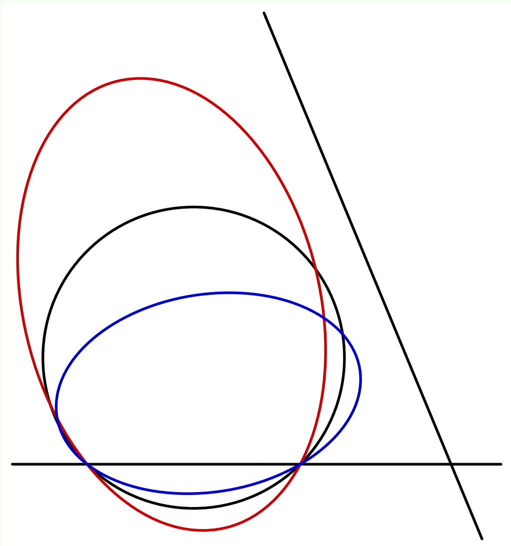
Renvoie la réflexion par rapport à `l1` parallèlement à `l2`.
Si `safe` vaut `true` et `l1` parallèle à `l2`, la routine renvoie l'identité.




```
import geometry;
size(5cm,0);
line AB=line(origin, NE), CD=line(2*NE+N, 2*NE+SE);
draw(AB, linewidth(bp)); draw(CD, linewidth(bp));
transform reflect=reflect(AB,CD);

point M=1.75*NE+0.5N, Mp=reflect*M;
dot("M",M, unit(M-Mp)); dot("M'", Mp, unit(Mp-M), red);
draw(segment(M,Mp), grey, StickIntervalMarker(2,2,grey));
draw(box((1,1), (2.2,2.2)), invisible);
```

— `transform scale(real k, line l1, line l2, bool safe=false)`

Renvoie l'affinité de rapport k, d'axe `l1` et de direction `l2`.
Si `safe` vaut `true` et `l1` parallèle à `l2`, la routine renvoie l'identité.



```
import geometry;
size(6.5cm,0);
pen bpp=linewidth(bp);
line AB=line(origin, E), CD=line(2*NE+N, 2*NE+SE);
draw(AB, bpp); draw(CD, bpp);

transform dilatation=scale(1.5,AB,CD);

path cle=shift(NE)*unitcircle;
draw(cle,bpp);

draw(dilatation*cle, 0.8*red+bpp);
draw(inverse(dilatation)*cle, 0.8*blue+bpp);
draw(box((-0.5,-0.5), (2.75,3)), invisible);
```

— `transform projection(line l)`

Renvoie la projection orthogonale sur `l`.

— `transform projection(line l1, line l2, bool safe=false)`

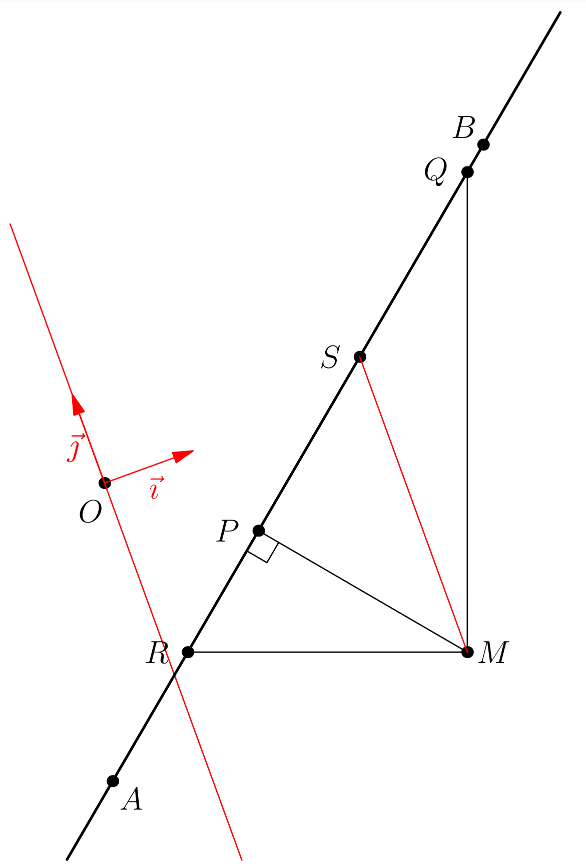
Renvoie la projection sur `l1` parallèlement à `l2`.
Si `safe` vaut `true` et `l1` parallèle à `l2`, la routine renvoie l'identité.

— `transform vprojection(line l, bool safe=false)`

Renvoie la projection sur `l` parallèlement à la verticale. Cette routine est équivalente à `projection(l,line(origin,point(def`
Si `safe` vaut `true` et `l` est une droite verticale, la routine renvoie l'identité.

— `transform hprojection(line l, bool safe=false)`

Renvoie la projection sur `l` parallèlement à l'horizontale. Cette routine est équivalente à `projection(l,line(origin,point(d`
Si `safe` vaut `true` et `l` est une droite horizontale, la routine renvoie l'identité.




```
import geometry;
size(7.5cm,0); dotfactor*=1.5;
currentcoordsys=rotate(20)*defaultcoordsys;
show(currentcoordsys, xpen=invisible, ypen=red);

point A=(-1,-3), B=(5,2);
line l1=line(A,B); draw(l1, linewidth(bp));
dot("$A$", A, SE); dot("$B$", B, NW);
point M=(3,-3); dot("$M$", M);

point P=projection(l1)*M;
dot("$P$", P, 2W); draw(M--P);
markrightangle(l1.A, P, M);

point Q=vprojection(l1)*M;
dot("$Q$", Q, 2W); draw(M--Q);

point R=hprojection(l1)*M;
dot("$R$", R, 2W); draw(M--R);

point S=projection(l1,line((0,0),(0,1)))*M;
dot("$S$", S, 2W); draw(M--S, red);
draw(box((-1,-4),(5,5)), invisible);
```

8. Coniques

8.1. Le type « conic »

8.1.1. Description

L'extension *geometry.asy* définit le type `conic` pour instancier une conique quelconque non dégénérée. S'il est tout à fait possible d'utiliser une instance de ce type, son existence est plutôt destinée au fonctionnement interne de l'extension ; on préférera utiliser directement les types dérivés `circle`, `ellipse`, `parabola` et `hyperbola` décrits ultérieurement.

Attardons nous cependant un peu sur sa structure afin d'en définir précisément les composantes :

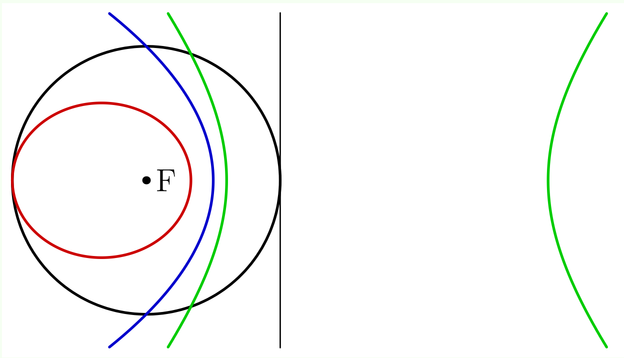
```
struct conic { real e, p, h; point F; line D; }
```

- `e` est l'excentricité ;
- `F` est un foyer et `D` la directrice associée ;
- `h` est la distance de `F` à `D` ;
- `p` est le paramètre, il vérifie l'égalité `p=he`.

Les deux principales routines pour définir une conique quelconque sont :

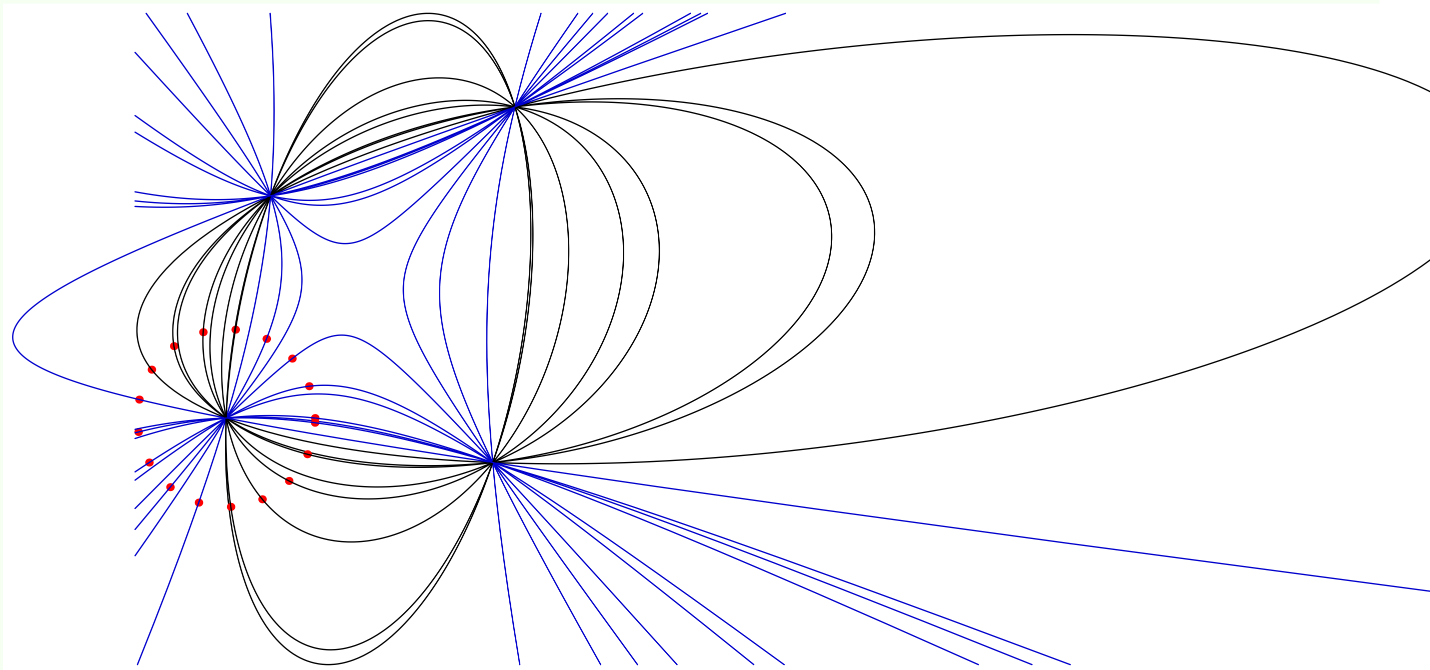
1. `conic conic(point F, line l, real e)`

   Retourne la conique de foyer `F` associé à la directrice `l` et d'excentricité `e` ; en voici un exemple d'utilisation :




```
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);
```

2. `conic conic(point M1, point M2, point M3, point M4, point M5)`
   Retourne la conique non dégénérée passant par les points `M1`, `M2`, `M3`, `M4` et `M5`.



```
import geometry;
size(18cm,0);
point B=(1.75,3), C=(-1,2), D=(-1.5,-0.5), F=(1.5,-1);

for (int i=0; i < 360; i += 21) {
  point A=shift(D)*dir(i);
  dot(A,red);
  conic co=conic(A,B,C,D,F);
  draw(co, co.e < 1 ? black : 0.8*blue);
}
```

On notera qu'il est aussi possible de définir une conique d'après son équation dans un repère spécifique, voir la section Équations de coniques, et que d'autres façons de définir une conique sont implémentées par des routines renvoyant un type spécifique de conique qu'il est ensuite possible de convertir en type `conic` ; voir Coniques et « casting ».

8.1.2. Routines de base

Les routines suivantes peuvent être utilisées en remplaçant un objet de type `conic` par l'un des types `circle`, `ellipse`, `parabola` ou `hyperbola` sauf lorsque le mot clef `explicit` précède le type `conic` dans la définition de la routine.

On notera qu'en plus des routines décrites dans cette section s'ajoutent des routines retournant une abscisse d'un point sur un objet de type `conic`.

— `conic changecoordsys(coordsys R, conic co)`

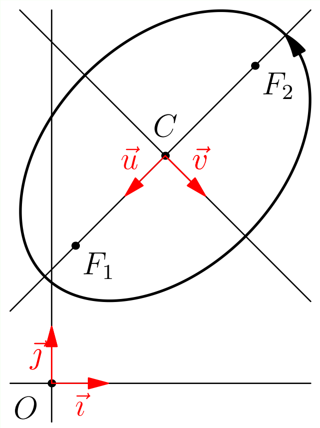
Retourne la même conique que `co` relativement au repère `R`.

— `coordsys coordsys(conic co)`

Retourne le repère dans lequel est définie la conique `co`.

— `coordsys canonicalcartesiansystem(explicit conic co)`

Retourne le repère canonique de la conique `co`.
Les routines `canonicalcartesiansystem(ellipse)`, `canonicalcartesiansystem(parabola)` et `canonicalcartesiansystem(hyperbola)` sont aussi disponibles.
L'exemple suivant en est une illustration dans le cas d'une ellipse.



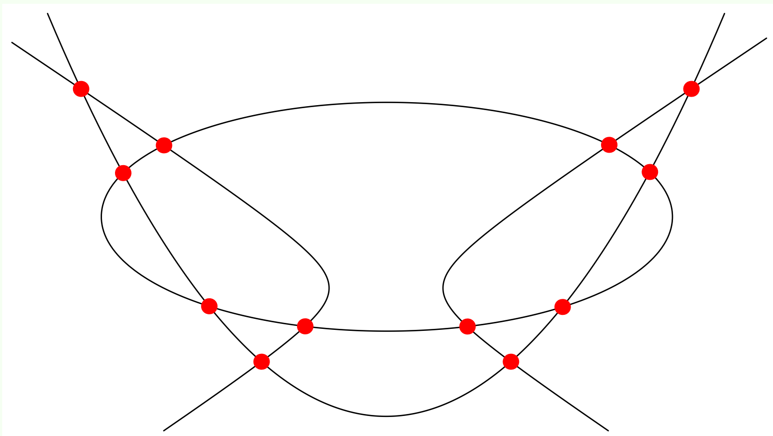

```
import geometry;
size(4cm,0);

show(defaultcoordsys);
ellipse el=ellipse((point)(2,4),3,2,45);
dot("$F_1$", el.F1, dir(-45));
dot("$F_2$", el.F2, dir(-45));
draw(el, linewidth(bp), Arrow(3mm));
show("$C$", "$\vec{u}$" ,"$\vec{v}$",
     canonicalcartesiansystem(el));
```

— `int conicnodesnumber(conic co, real angle1, real angle2, bool dir=CCW)`

Retourne le nombre de nœuds utilisés pour convertir la conique `co` en `path` entre les angles `angle1` et `angle2` donnés dans le sens de parcours `dir`.

— `point[] intersectionpoints(conic co1, conic co2)`

Retourne, sous forme d'un tableau, les points d'intersections des deux coniques `co1` et `co2`.



```
import geometry; size(10cm); conic co[];
co[0]=conic((-4.58,1.25), line((-5.45545,1.25), (-5.45545,2.12287)), 0.9165);
draw(co[0]);
co[1]=conic((0,-1),line((0,-3.5),(-1,-3.5)),1); draw(co[1]);
co[2]=conic((-1.2,0), line((-5/6,0),(-5/6,-1)),1.2); draw(co[2]);
dotfactor *= 2;
for (int i=0; i < 3; ++i)
  for (int j=i+1; j < 3; ++j)
    dot(intersectionpoints(co[i],co[j]), red);
addMargins(lmargin=10mm,bmargin=10mm);
```

— `point[] intersectionpoints(line l, conic co)`

Retourne, sous forme d'un tableau, les points d'intersections de la droite `l` avec la conique `co`.
La routine `intersectionpoints(conic,line)` est aussi définie.

— `point[] intersectionpoints(triangle t, conic co, bool extended=false)`

Retourne, sous forme d'un tableau, les points d'intersections du triangle `t` avec la conique `co`. Si `extended` vaut `true` les côtés du triangle sont considérés comme des droites ; voir la section Triangles.
La routine `intersectionpoints(conic,triangle,bool)` est aussi définie.

8.1.3. Opérateurs

Comme pour les routines précédentes, les opérateurs décrits ici peuvent être utilisées en remplaçant un objet de type `conic` par l'un des types `circle`, `ellipse`, `parabola` ou `hyperbola`.

— `bool operator @(point M, conic co)`

Autorise le code `point @ conic`.
Retourne `true` si et seulement si le point `M` appartient à la conique `co`.

— `conic operator *(transform t, conic co)`

Autorise le code `transform*conic`.

— `conic operator +(conic co, explicit point M)`

Autorise le code `conic+point`.
Retourne le translaté de la conique `co` par le vecteur $\overrightarrow{O\texttt{M}}$.
La routine `-(conic,explicit point)` est aussi définie.

— `conic operator +(conic co, explicit pair m)`

Autorise le code `conic+pair`.
Retourne le translaté de la conique `co` par le vecteur $\overrightarrow{\texttt{Om}}$; m représente alors les coordonnées d'un point défini relativement au repère dans lequel est défini la conique.
La routine `-(conic,explicit pair)` est aussi définie.

— `conic operator +(conic co, explicit vector u)`

Autorise le code `conic+vector`.
Retourne le translaté de la conique `co` par le vecteur $\overrightarrow{\texttt{u}}$.
La routine `-(conic,explicit vector)` est aussi définie.

8.1.4. Équations de coniques

Le type `bqe`, pour *Bivariate Quadratic Equation*, permet d'instancier un objet représentant une équation de conique dans un repère donné. Sa structure est la suivante :

```
struct bqe
{
  real[] a;
  coordsys coordsys;
}
```

où :

— `a` est un tableau des six coefficients d'une équation de la conique donnée sous la forme

$$\texttt{a[0]}x^2 + \texttt{a[1]}xy + \texttt{a[2]}y^2 + \texttt{a[3]}x + \texttt{a[4]}y + \texttt{a[5]} = 0$$

— `coordsys` est le repère dans lequel cette équation est donnée.

Voici la liste des routines concernant les objets de type `bqe` :

— `bqe bqe(coordsys R=currentcoordsys, real a, real b, real c, real d, real e, real f)`

Retourne un objet de type `bqe` représentant l'équation $\texttt{a}x^2 + \texttt{b}xy + \texttt{c}y^2 + \texttt{d}x + \texttt{e}y + \texttt{f} = 0$ relativement au repère `R`.

— `bqe changecoordsys(coordsys R, bqe bqe)`

Retourne un objet de type `bqe` relatif au repère `R` et représentant la même conique que celle représentée par le paramètre `bqe`. Cette routine permet donc d'effectuer un changement de repère dans une équation quadratique à deux variables.

— `bqe bqe(point M1, point M2, point M3, point M4, point M5)`

Retourne l'équation de la conique passant par les cinq points `M1`, `M2`, `M3`, `M4` et `M5`.
Si les points sont définis relativement au même repère, l'équation est relative à ce repère ; dans le cas contraire l'équation est relative au repère par défaut `defaultcoordsys`.

— `string conictype(bqe bqe)`

Retourne le type de conique représentée par `bqe`. Les valeurs retournées possibles sont `"degenerated"`, `"ellipse"`, `"parabola"` et `"hyperbola"`.

— `bqe equation(explicit conic co)`

Retourne, sous forme d'objet de type `bqe`, l'équation de la conique `co`.
Les routines `equation(ellispe)`, `equation(parabola)` et `equation(hyperbola)` sont aussi disponibles.

— `bqe canonical(bqe bqe)`

Retourne l'équation de la conique représentée par `bqe` dans le repère canonique de la dite conique.

— `conic conic(bqe bqe)`

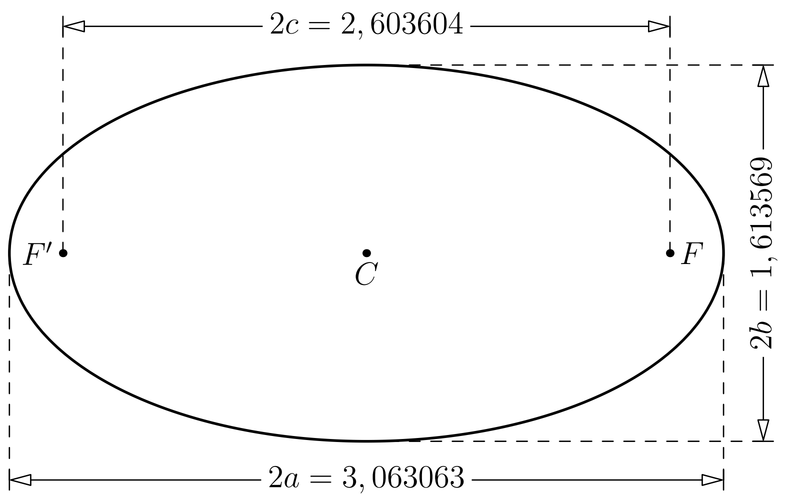
Retourne la conique dont une équation est représentée par `bqe`.

8.1.5. Coniques et « casting »

Comme il a déjà été mentionné dans les sections précédentes, le type `conic` permet d'instancier un objet représentant une conique quelconque. Il est toutefois possible, et souvent recommandé, de convertir un objet représentant une conique quelconque en une conique d'un type spécifique afin d'utiliser les propriétés et routines qui lui sont propres.

Les types spécifiques de coniques sont `circle`, lui même un cas particulier du type `ellipse`, `parabola` et `hyperbola` ; ces type seront décrits dans les sections suivantes.

Ainsi dans l'exemple suivant la conique `co` est définie par un foyer et la directrice correspondante avec une excentricité inférieure à 1. Comme cette conique est une ellipse, on peut affecter la variable `co` à une variable de type `ellipse` pour en récupérer les dimensions.




```
import geometry;
size(10cm);
point F=(-1,0); line D=line(N,S);
conic co=conic(F, D, 0.85); dot("$F$", F); draw(co, linewidth(bp));

ellipse el=(ellipse)co; dot("$C$", el.C, S);
distance(format("$2c=%f$", el.c),el.F1, el.F2, 3cm, joinpen=dashed);
distance(format("$2a=%f$", el.a), relpoint(el,0), relpoint(el,0.5), 3cm,
         joinpen=dashed);
distance(format("$2b=%f$", el.b), relpoint(el,0.25), relpoint(el,0.75), 5.25cm,
         joinpen=dashed);
dot("$F'$", el.F2, W);
```

Du point de vue du fonctionnement interne de l'extension *geometry.asy*, certaines routines s'appliquant à un objet de type `conic` font appel en fait à des routines équivalentes s'appliquant à une conique spécifique ; cela permet d'optimiser certains calculs. Inversement, des routines relatives à un type spécifique de conique utilisent de façon sous-jacente des routines relatives à une conique quelconque.
Les types spécifiques de coniques sont décrits ci-après.

8.2. Cercles

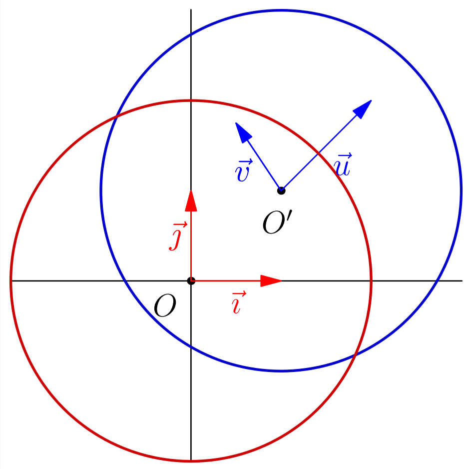
8.2.1. Routines de bases

En dehors des routines concernant les objets de type `conic`, voici d'autres routines permettant de définir un objet de type `circle` :

— `circle circle(explicit point C, real r)`

Renvoie le cercle de centre `C` et de rayon `r`.
Depuis la version 2.10 d'ASYMPTOTE la routine `circle circle(pair C, real r)` n'est plus redéfinie afin de renvoyer un objet de type `circle` comme c'était le cas dans les versions précédentes ; ceci oblige à utiliser le « casting » de `pair` à `point` dans le code `circle cle = circle((point)(1,2), 2)` et permet ainsi d'obtenir le cercle de centre ((1,2)) dans le repère courant `currentcoordsys` et rayon 2.

L'exemple suivant illustre la différence entre le code `circle((point)(0,0),R);` qui définit le cercle bleu dans le repère courant et `circle(point(defaultcoordsys,(0,0)), R);` qui définit le cercle rouge dans le repère par défaut ; évidemment si la variable `currentcoordsys` n'est pas modifiée les deux codes sont équivalents.




```
import geometry;
size(6cm,0);
currentcoordsys=cartesiansystem((1,1), i=(1,1),
                                          j=(-0.5,.75));
show("$O'$","$\vec{u}$", "$\vec{v}$", currentcoordsys,
     ipen=blue, xpen=invisible);
show(defaultcoordsys);
real R=2;
circle C=circle((point)(0,0), R);
draw(C, bp+0.8*blue);
circle Cp=circle(point(defaultcoordsys,(0,0)), R);
draw(Cp, bp+0.8*red);
```

— `circle circle(point A, point B)`

  Renvoie le cercle de diamètre `AB`.

— `circle circle(point A, point B, point C)`

  Renvoie le cercle passant par les points distincts `A`, `B` et `C`.
  Un alias de cette routine est `circle circumcircle(point A, point B, point C)`.

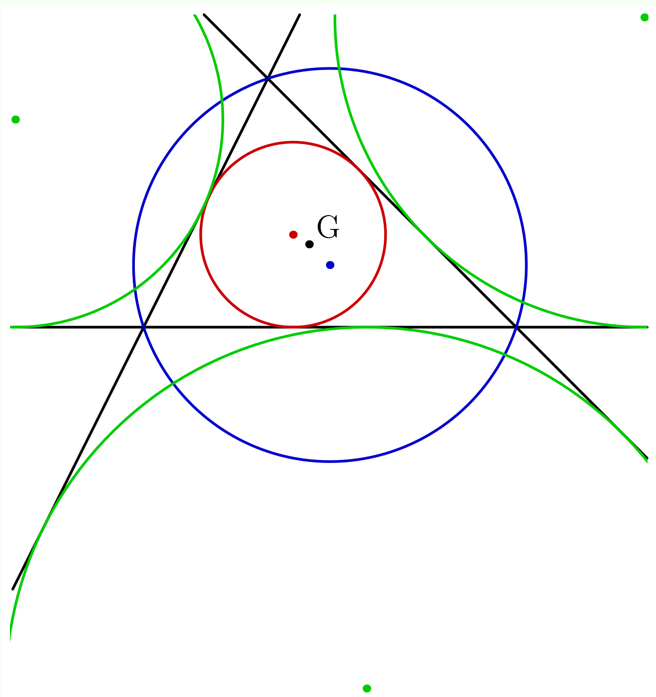
— `circle incircle(point A, point B, point C)`

  Renvoie le cercle inscrit du triangle ABC.

— `circle excircle(point A, point B, point C)`

  Renvoie le cercle exinscrit du triangle ABC tangent à (AB).

Dans l'exemple suivant on remarquera l'utilisation de la routine `clipdraw` qui trace un chemin en se restreignant aux dimensions de l'image finale.




```
import geometry;
size(9cm);
green=0.8green; blue=0.8blue; red=0.8red;
pen bpp=linewidth(bp);
point A=(-1,0), B=(2,0), C=(0,2);
draw(line(A,B),bpp); draw(line(A,C),bpp);
draw(line(B,C),bpp);
circle cc=circle(A,B,C);
draw(cc, bp+blue); dot(cc.C, blue);
circle ic=incircle(A,B,C);
draw(ic, bp+red); dot(ic.C, red);
circle ec=excircle(A,B,C);
clipdraw(ec, bp+green); dot(ec.C, green);
ec=excircle(A,C,B);
clipdraw(ec, bp+green); dot(ec.C, green);
ec=excircle(C,B,A);
clipdraw(ec, bp+green); dot(ec.C, green);
dot("G", centroid(A,B,C), NE);
```

Des routines spécifiques à la géométrie du triangle permettent d'obtenir le même résultat de façon plus élégante, voir la section Triangles.

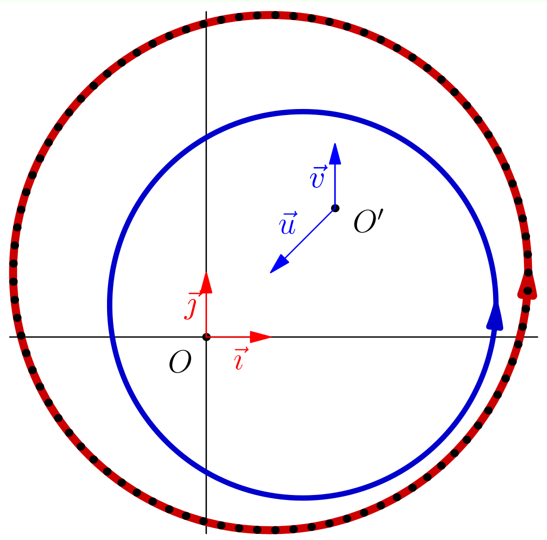
8.2.2. Du type « circle » au type « path »

La conversion d'un objet de type `circle` en `path` s'effectue suivant les règles suivantes :

— le chemin est cyclique, orienté dans le sens trigonométrique ;

— le premier point du chemin, celui renvoyé par la routine `pair point(path g, real t)` pour `t=0`, est le point d'intersection du cercle avec la demi-droite issue du centre et de direction « le premier vecteur du repère dans lequel le cercle est défini » ;

— le nombre de points du chemin est fonction du rayon du cercle ; il est calculé par la routine `int circlenodesnumber(real r)` qui dépend elle-même de la variable `circlenodesnumberfactor`.




```
import geometry;
size(7cm,0);
currentcoordsys=cartesiansystem((2,2), i=(-1,-1),
                                j=(0,1));
show("$O'$","$\vec{u}$", "$\vec{v}$",
     currentcoordsys, ipen=blue, xpen=invisible);
show(defaultcoordsys);
circle C=circle((point)(0.5,-1), 3);
draw(C, 2bp+0.8*blue, Arrow(3mm));
circle Cp=circle(point(defaultcoordsys,(1,1)), 4);
draw(Cp, dotsize()+0.8*red, Arrow(3mm));
dot((path)Cp);
```

8.2.3. Les opérateurs

En dehors des opérateurs s'appliquant aux objets de type `conic`, voici la liste d'autres opérateurs définis pour les objets de type `circle`.

— `circle operator *(real x, explicit circle c)`

Autorise le code `real*circle`.
Renvoie le cercle de même centre que `c` et de rayon `x` fois celui de `c`.
L'opérateur `circle operator /(explicit circle c, real x)` est aussi défini.

— `real operator ^(point M, explicit circle c)`

Autorise le code `point^circle`.
Renvoie la puissance de `M` par rapport à `c`.

— `bool operator @(point M, explicit circle c)`

Autorise le code `point @ circle`.
Renvoie `true` si et seulement si le point `M` appartient au cercle `c`.

— `ellipse operator cast(circle c)`

Permet le « casting » `circle` vers `ellipse`.
Le « casting » de `ellipse` vers `circle` est aussi défini.
On notera que l'opérateur `*(transform t, circle c)` n'existe pas ; par le jeu du « casting », c'est l'opérateur `ellipse operator *(transform t, ellipse el)` qui est utilisé lors de l'exécution du code `transform*circle`.
Ainsi le code `scale(2)*circle` renvoie un objet de type `ellipse` mais il est possible d'écrire `circle=scale(2)*circle` alors que le code `circle=xscale(2)*circle` génère une erreur.

8.2.4. Autres routines

En dehors des routines s'appliquant aux objets de type `conic`, voici la liste des routines spécifiques aux objets de type `circle`.

— `point radicalcenter(circle c1, circle c2)`

  Renvoie le pied de l'axe radical des deux cercles `c1` et `c2`.
  Le repère dans lequel est défini le point renvoyé est celui de `c1`.

— `point radicalcenter(circle c1, circle c2, circle c3)`

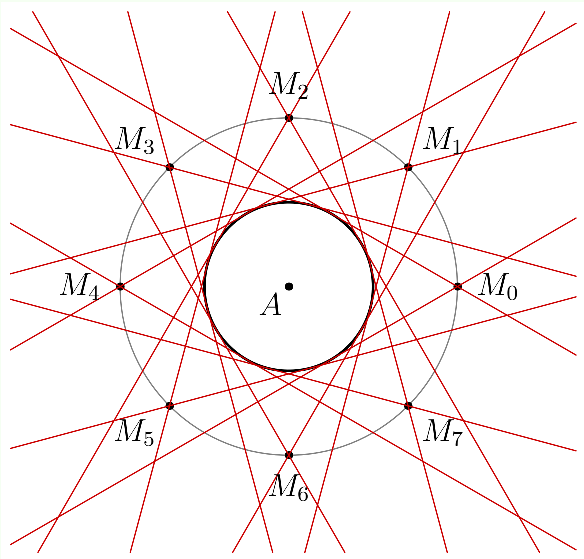
  Renvoie le centre radical des trois cercles `c1`, `c2` et `c3`.

— `line radicalline(circle c1, circle c2)`

  Renvoie l'axe radical des deux cercles `c1` et `c2`.

— `line[] tangents(circle c, point M)`

  Renvoie les tangentes éventuelles à `c` passant par `M`.



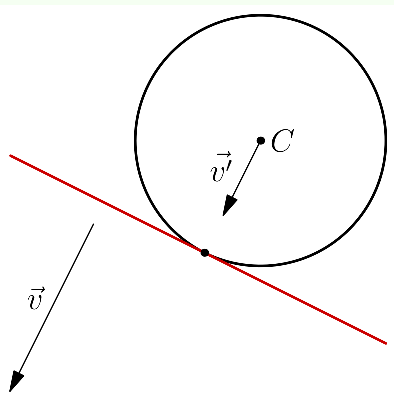

```
import geometry;
size(7.5cm,0);

point A=(2.5,-1); dot("$A$", A, SW);
circle C=circle(A,1); draw(C, linewidth(bp));

path Cp=shift(A)*scale(2)*unitcircle;
draw(Cp, grey);
for (int i=0; i < 360; i+=45) {
  point M=relpoint(Cp, i/360);
  dot(format("$M_%f$", i/45), M, 2*unit(M-A));
  draw(tangents(C, M), 0.8*red);
}
addMargins(10mm,10mm);
```

— `line tangent(circle c, point M)`

  Renvoie la tangente à `c` au point d'intersection de `c` avec la demi-droite d'origine `c.C` passant par `M`. Le point de tangence peut être obtenu avec la routine `point(circle c, point M)`.

— `line tangent(circle c, explicit vector v)`

  Renvoie la tangente à `c` au point d'intersection de `c` avec la demi-droite d'origine `c.C` orientée par le vecteur `v`. Le point de tangence peut être obtenu avec la routine `point(circle c, vector v)`.




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

circle cle=circle((point)(2,1),1.5);
draw(cle, linewidth(bp));
dot("$C$", cle.C);

vector v=(-1,-2);
show("$\vec{v}$",v);

line tgt=tangent(cle,v);
draw(tgt, bp+0.8*red);
draw("$\vec{v'}$",cle.C--(cle.C+tgt.v), Arrow);
dot(point(cle,v));
```

— `line tangent(circle c, abscissa x)`

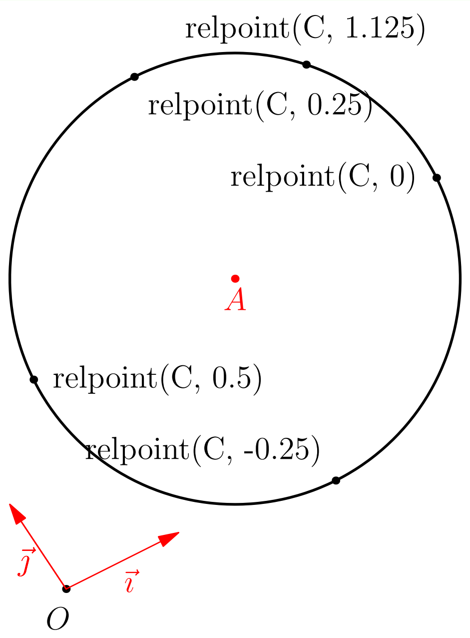
Retourne la tangente à `c` au point de `c` d'abscisse `x`.

— `point point(explicit circle c, real x)`

Retourne le `point` de `c` marquant le même point que le `pair` retourné par le code `point((path)c,x)`.

— `point relpoint(explicit circle c, real x)`

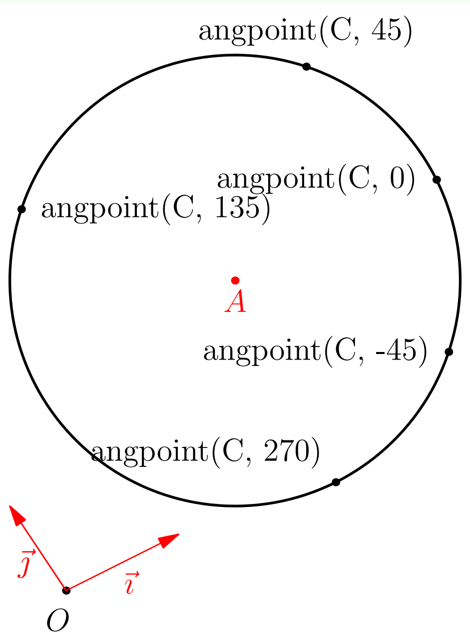
Retourne le point de `c` correspondant à la fraction `x` du périmètre de `c`.




```
import geometry;
size(6cm,0);
currentcoordsys=cartesiansystem((0,0), i=(1,0.5),
                                j=(-0.5,.75));
show(currentcoordsys, xpen=invisible);

point A=(2.5,2); dot("$A$", A, S, red);
real R=2;
circle C=circle(A,R);
draw(C, linewidth(bp));

dot("relpoint(C, 0)", relpoint(C,0), 2W);
dot("relpoint(C, 0.25)", relpoint(C,0.25), 2SE);
dot("relpoint(C, 0.5)", relpoint(C,0.5), 2E);
dot("relpoint(C, -0.25)", relpoint(C, -0.25), 2NW);
dot("relpoint(C, 1.125)", relpoint(C, 1.125), 2N);
```

— `point angpoint(explicit circle c, real x)`

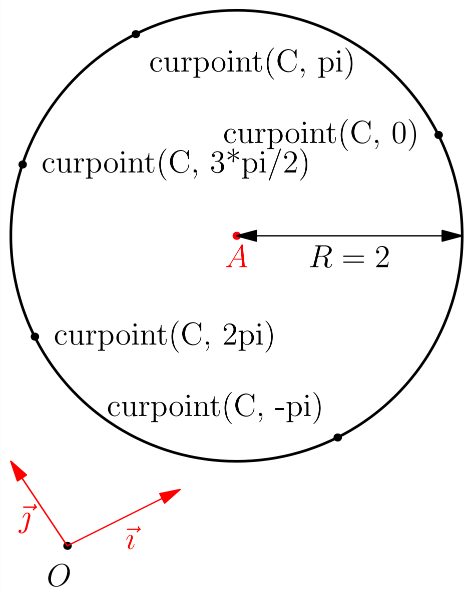
Retourne le point de `c` d'angle `x` degrés.




```
import geometry;
size(6cm,0);
currentcoordsys=cartesiansystem((0,0), i=(1,0.5),
                                j=(-0.5,.75));
show(currentcoordsys, xpen=invisible);
point A=(2.5,2); dot("$A$", A, S, red);
real R=2;
circle C=circle(A,R);
draw(C, linewidth(bp));

dot("angpoint(C, 0)", angpoint(C,0), 2W);
dot("angpoint(C, 45)", angpoint(C,45), 2N);
dot("angpoint(C, 135)", angpoint(C,135), 2E);
dot("angpoint(C, 270)", angpoint(C, 270), 2NW);
dot("angpoint(C, -45)", angpoint(C, -45), 2W);
```

— `point curpoint(explicit circle c, real x)`

Retourne le point de `c` dont l'abscisse curviligne est `x`.



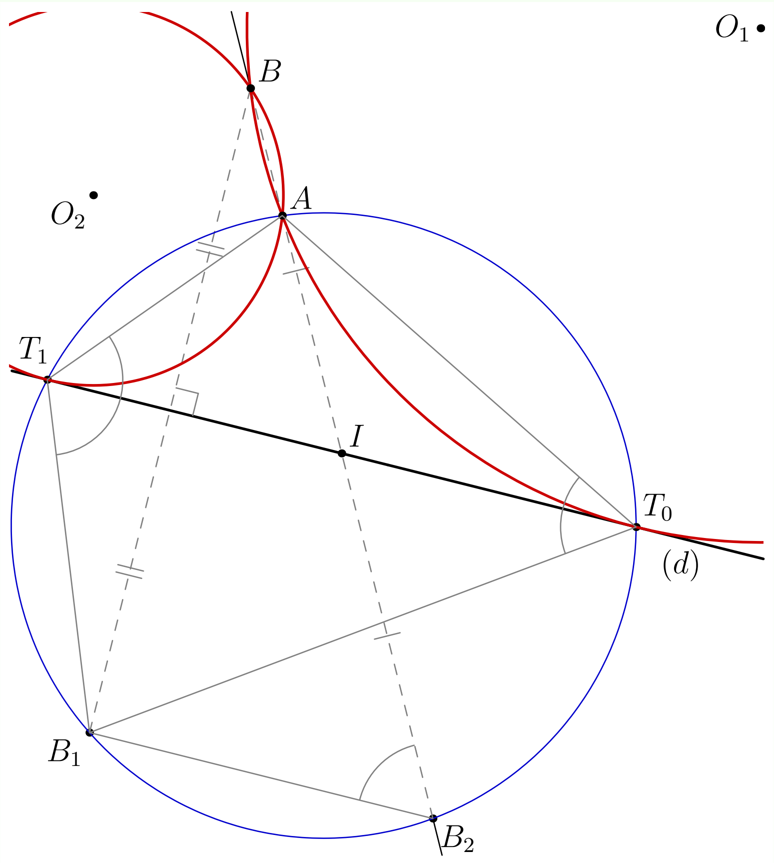

```
import geometry;
size(6cm,0); real R=2;
currentcoordsys=cartesiansystem((0,0), i=(1,0.5),
                                j=(-0.5,.75));
show(currentcoordsys, xpen=invisible);
point A=(2.5,2); dot("$A$", A, S, red);
circle C=circle(A,R); draw(C, linewidth(bp));
draw(rotate(A-point(C,0))*("$R="+(string)R+"$"),
     A--point(C,0), S, Arrows);

dot("curpoint(C, 0)", curpoint(C,0), 2W);
dot("curpoint(C, pi)", curpoint(C,pi), 2SE);
dot("curpoint(C, 3*pi/2)", curpoint(C,3*pi/2), 2E);
dot("curpoint(C, -pi)", curpoint(C, -pi), 2NW);
dot("curpoint(C, 2pi)", curpoint(C, 2*pi), 2E);
```

D'autres routines sont définies pour les objets de type `circle`, elles sont accessibles via des routines utilisant le type `ellipse`.

Pour terminer cette section notons qu'il est possible d'utiliser un objet de type `circle` comme une inversion. On pourra se reporter à la section Inversions pour plus de détails.

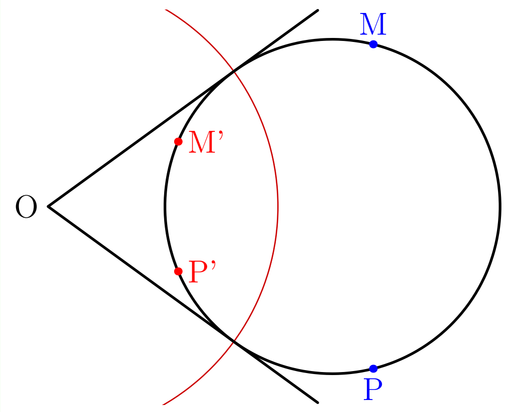
Voici quelques exemples d'utilisations des routines précédemment décrites :

— Construction des deux cercles passant par les points A et B donnés et tangents à la droite (d) donnée.




```
import geometry;
size(10cm,0);
pen bpp=linewidth(bp);
line l=line(origin,(1,-0.25)); draw("$(d)$", l, bpp);
point A=(1,1.5), B=(0.75,2.5);
line AB=line(A,B);
point B1=reflect(l)*B, I=intersectionpoint(l,AB), B2=rotate(180,I)*B;
dot("$I$", I, NE); dot("$B_1$", B1, SW); dot("$B_2$", B2, SE);

draw(B--B1, grey+dashed, StickIntervalMarker(2,2,grey));
markrightangle(B,midpoint(B--B1),I, grey);
draw(B--B2, grey+dashed, StickIntervalMarker(2,1,grey));
draw(complementary(segment(B,B2)));

circle C=circle(A,B1,B2); draw(C, 0.8*blue);
point[] T=intersectionpoints(l,C);
dot("$T_0$",T[0], NE); dot("$T_1$",T[1], N+NW);

circle C1=circle(A,B,T[0]), C2=circle(A,B,T[1]);
clipdraw(C1, bpp+0.8*red); clipdraw(C2, bpp+0.8*red);
dot("$O_1$", C1.C, W); dot("$O_2$", C2.C, SW); dot("$A$", A, NE); dot("$B$", B, NE);

draw(A--T[0]--B1, grey); markangle(A,T[0],B1, grey);
draw(A--T[1]--B1, grey); markangle(B1,T[1],A, grey);
draw(B2--B1, grey); markangle(A,B2,B1, grey);
```

— Deux points du plan et leurs inverses sont cocycliques, sur le cercle othogonal au cercle d'inversion.

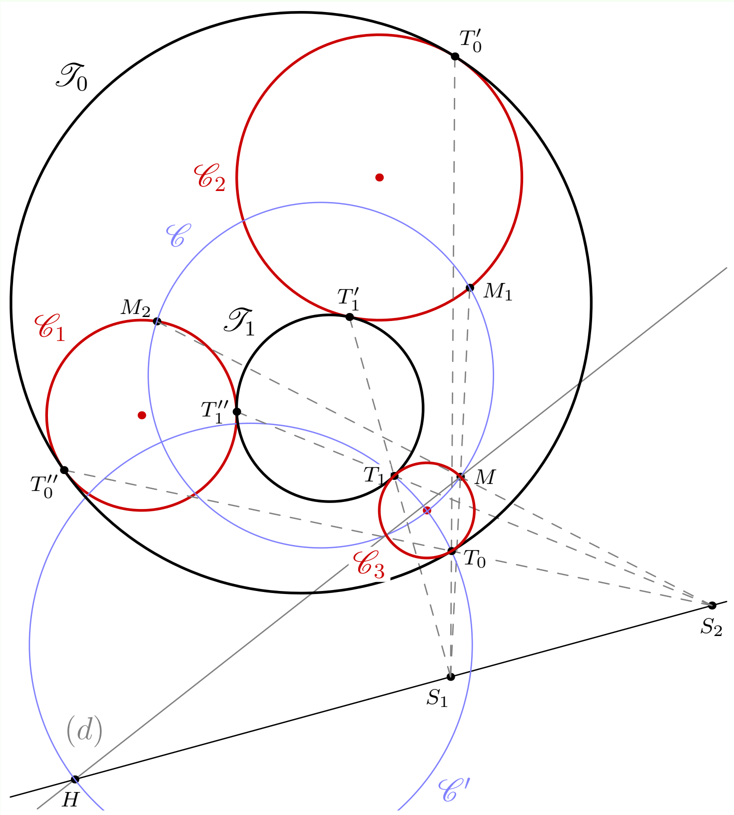



```
import geometry;
size(6.5cm,0); currentpen=linewidth(bp);
point O=origin, M=(2,1), P=(2,-1);
dot("O", O, W);
inversion t=inversion(2,O);
point Mp=t*M, Pt=t*P;
circle C=circle(M,P,Mp); draw(C);
dot("M", M, N, blue); dot("P", P, S, blue);
dot("M'", Mp, red); dot("P'", Pt, red);
circle Ct=circle(t); clipdraw(Ct, 0.8*red);
point[] T=intersectionpoints(C,Ct);
draw(line(O,false,T[0])); draw(line(O,false,T[1]));
```

— Étant donnés trois cercles $\mathscr{C}_1$, $\mathscr{C}_2$ et $\mathscr{C}_3$ tels que $r_3 < r_1$ et $r_3 < r_2$, comment construire des cercles simultanément tangents à ces cercles ?

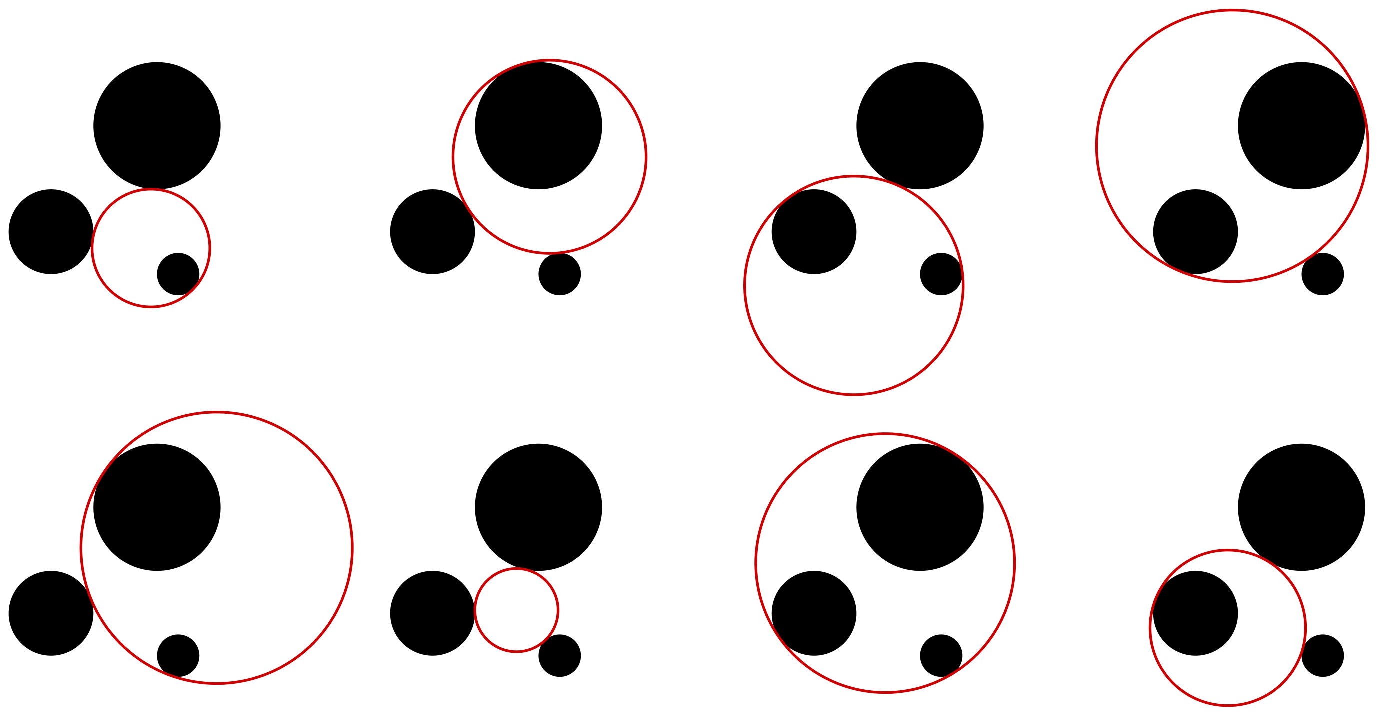
Le principe de la construction illustrée ci-après est le suivant :

- — On note S_1 et S_2 les inversions de rapport positif transformant $\mathscr{C}_2$ en $\mathscr{C}_3$ et $\mathscr{C}_1$ en $\mathscr{C}_3$ respectivement ;
- — on considère un point M sur le cercle $\mathscr{C}_3$ et l'on note M_1 et M_2 son image par S_1 et S_2 respectivement ;
- — on note $\mathscr{C}$ le cercle passant par M, M_1 et M_2 ;
- — l'axe radical (d) des cercles $\mathscr{C}_3$ et $\mathscr{C}$ coupe la droite des centres d'inversions (S_1S_2) en H ; on note $\mathscr{C}'$ le cercle de diamètre $[HO_3]$ où O_3 est le centre de $\mathscr{C}_3$;
- — le cercle $\mathscr{C}'$ coupe $\mathscr{C}_3$ en deux points T_0 et T_1 ;
- — le cercle passant par T_0 et par les images T'_0 et T''_0 de T_0 par S_1 et S_2 respectivement est une solution ;
- — le cercle passant par T_1 et par les images T'_1 et T''_1 de T_1 par S_1 et S_2 respectivement est une autre solution.




```
import geometry;
size(9.5cm,0); usepackage("mathrsfs"); currentpen=fontsize(8); pen bpp=linewidth(bp);
circle C1=circle((point)(0,0),2), C2=circle((point)(5,5), 3), C3=circle((point)(6,-2),1);
draw(Label("$\mathscr{C}_1$",Relative(0.375)), C1, bp+0.8*red);
draw("$\mathscr{C}_2$", C2, bp+0.8*red);
dot(C1.C, 0.8*red); dot(C2.C, 0.8*red); dot(C3.C, 0.8*red);
inversion S1=inversion(C2,C3), S2=inversion(C1,C3);
dot("$S_1$", S1.C, 2S+W); dot("$S_2$", S2.C, 2S);
line cl=line(S1.C,S2.C); draw(cl);
point M=relpoint(C3,0.125), M2=S2*M, M1=S1*M;
dot("$M$", M, 2*E); dot("$M_2$", M2, NW); dot("$M_1$", M1, 2*dir(-10));
draw(segment(S2.C,M2), dashed+grey); draw(segment(S1.C,M1), dashed+grey);
circle C=circle(M,M2,M1);
draw(Label("$\mathscr{C}$", Relative(0.375)), C, lightblue);
line L=radicalline(C,C3); draw("$(d)$", L, grey);
point H=intersectionpoint(L,cl); dot("$H$", H, 2*dir(260));
circle Cp=circle(H,C3.C);
clipdraw(Label("$\mathscr{C'}$", Relative(0.9)), Cp, lightblue);
point[] T=intersectionpoints(Cp,C3);
point[][] Tp= new point[][] {{S2*T[0], S1*T[0]},{S2*T[1], S1*T[1]}};
draw(S2.C--Tp[0][0], dashed+grey); draw(S1.C--Tp[0][1], dashed+grey);
draw(S2.C--Tp[1][0], dashed+grey); draw(S1.C--Tp[1][1], dashed+grey);
dot(Label("$T_0$",UnFill), T[0], 2*dir(-20));
dot(Label("$T_1$",UnFill), T[1], W);
dot("$T''_0$", Tp[0][0], SW); dot("$T'_0$", Tp[0][1], NE);
dot("$T''_1$", Tp[1][0], W); dot("$T'_1$", Tp[1][1], N);
draw(Label("$\mathscr{T}_0$", Relative(0.375)), circle(T[0],Tp[0][0],Tp[0][1]), bpp);
draw(Label("$\mathscr{T}_1$", Relative(0.375)), circle(T[1],Tp[1][0],Tp[1][1]), bpp);
draw(Label("$\mathscr{C}_3$",Relative(0.625),UnFill), C3, bp+0.8*red);
```

— En prenant les quatre combinaisons possibles pour les inversions S_1 et S_2, on obtient de même les huit cercles tangents à trois cercles donnés :



```
import geometry;
size(18cm,0); int shx=18;
circle C1=circle((point)(0,0),2), C2=circle((point)(5,5), 3), C3=circle((point)(6,-2),1);
picture disc;
fill(disc,(path)C1); fill(disc,(path)C2); fill(disc,(path)C3);
transform tv=shift(S), th=shift(E);
int k=0, l=0;
for (int i=0; i < 2 ; ++i)
  for (int j=0; j < 2; ++j) {
    picture[]  tpic; tpic[0]=new picture; tpic[1]=new picture;
    add(tpic[0], disc); add(tpic[1], disc);
    inversion S1=inversion(C2,C3, sgnd(i-1)), S2=inversion(C1,C3, sgnd(j-1));
    line cl=line(S1.C,S2.C);
    point M=relpoint(C3,0.125), M2=S2*M, M1=S1*M;
    circle C=circle(M,M2,M1);
    line L=radicalline(C,C3);
    point H=intersectionpoint(L,cl);
    circle Cp=circle(H,C3.C);
    point[] T=intersectionpoints(Cp,C3);
    point[][] Tp= new point[][] {{S2*T[0], S1*T[0]},{S2*T[1], S1*T[1]}};
    draw(tpic[0], circle(T[0],Tp[0][0],Tp[0][1]), bp+0.8*red);
    draw(tpic[1], circle(T[1],Tp[1][0],Tp[1][1]), bp+0.8*red);
    add(tv^(shx*(i+1))*th^(shx*(l))*tpic[0]);
    l=(l+2)%4; ++k;
    add(tv^(shx*(i+1))*th^(shx*(l+1))*tpic[1]);
  }
```

8.3. Ellipses

Le type `ellipse` ne réserve pas de surprise, il permet d'instancier un objet représentant une ellipse. Comme le type `circle` est un cas particulier du type `ellipse` il est possible d'instancier un cercle en tant qu'une ellipse d'excentricité nulle et, inversement, d'instancier une ellipse d'excentricité nulle en tant qu'un cercle. Enfin, comme il existe une correspondance biunivoque entre les objets de type `ellipse` et ceux de type `conic` ayant une excentricité strictement inférieure à 1, les objets de type `ellipse` héritent des routines et opérateurs définis pour ceux de type `conic`.

8.3.1. Routines de bases

Voici la liste des autres routines permettant de définir un objet de type `ellipse`.

— `ellipse ellipse(point F1, point F2, real a)`

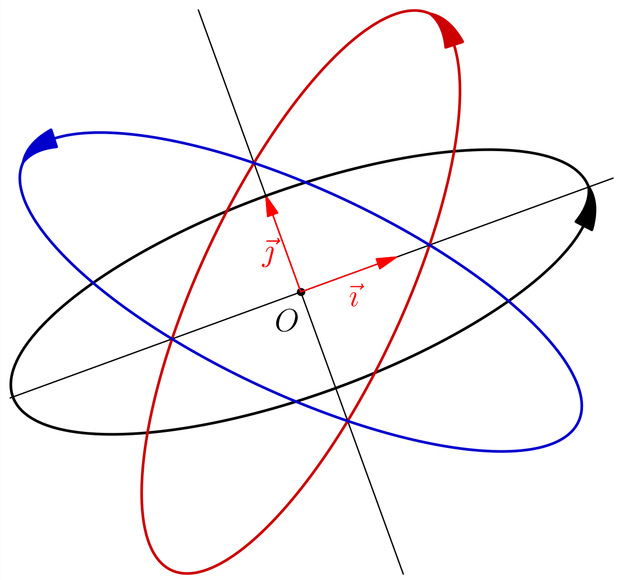
Retourne l'ellipse de foyers `F1` et `F2` ayant pour demi grand axe `a`.

— `ellipse ellipse(point F1, point F2, point M)`

Retourne l'ellipse de foyers `F1` et `F2` et passant par `M`.

— `ellipse ellipse(point C, real a, real b, real angle=0)`

Retourne l'ellipse de centre `C` dont le demi grand axe a pour longueur `a` dans la direction donnée par `dir(angle)` et dont le demi petit axe a pour longueur `b`.




```
import geometry;
size(8cm);

currentcoordsys=rotate(20)*defaultcoordsys;
show(currentcoordsys);

ellipse e0=ellipse((point)(0,0), 3, 1);
draw(e0, linewidth(bp), Arrow);

ellipse e1=ellipse((point)(0,0), 3, 1, 45);
draw(e1, bp+0.8*red, Arrow);

ellipse e2=ellipse((point)(0,0), 1, 3, 45);
draw(e2, bp+0.8*blue, Arrow);
```

8.3.2. Du type « ellipse » au type « path »

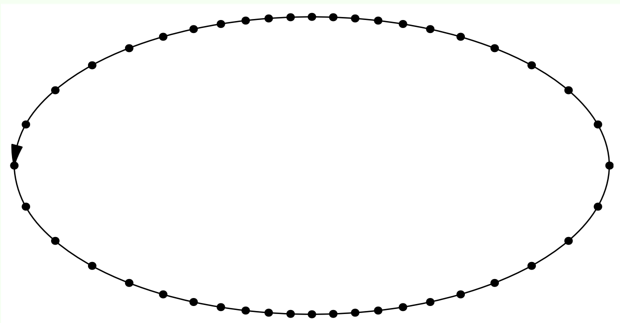
La conversion d'un objet de type `ellipse` en `path` s'effectue suivant les règles suivantes :

— le chemin est cyclique, orienté dans le sens trigonométrique ;

— le premier point, celui renvoyé par la routine `pair point(path g, real t)` pour `t=0`, est le point d'intersection de la demi-droite focale $[F_1F_2)$ avec l'ellipse ;

— le nombre de nœuds du chemin est fonction des longueurs des axes de l'ellipse ; il est calculé par la routine `int ellipsenodesnum` qui dépend elle-même de la variable `ellipsenodesnumberfactor` ;

— les nœuds du chemin sont définis en coordonnées polaires avec des angles donnés relativement au centre de l'ellipse et uniformément répartis dans l'intervalle $[0\,;360[$.



```
import geometry;
size(8cm,0);
ellipsenodesnumberfactor=50;
ellipse e=ellipse(origin, 4, 2, 180);
draw(e, Arrow);
dot((path)e);
```

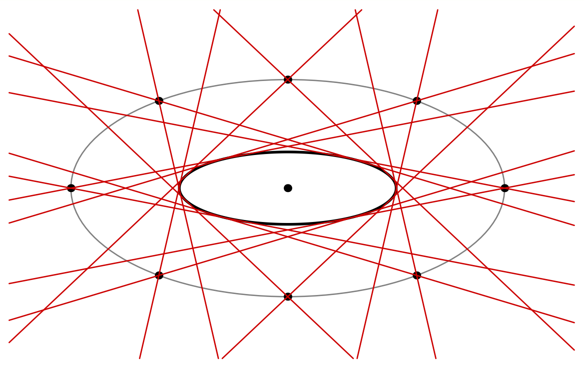
8.3.3. Autres routines

En dehors des routines s'appliquant aux objets de type `conic`, voici la liste des routines spécifiques aux objets de type `ellipse`.

—
```
real centerToFocus(ellipse el, real a)
```
Permet de convertir un angle donné relativement au centre de l'ellipse en l'angle relatif au premier foyer.
La routine `real focusToCenter(ellipse,real)` est aussi définie.

—
```
real arclength(ellipse el, real angle1, real angle2,
bool direction=CCW,
polarconicroutine polarconicroutine=currentpolarconicroutine)
```
Renvoie la longueur de l'arc d'ellipse représenté par `el` entre les angles `angle1` et `angle2` parcouru dans le sens `direction`.
`polarconicroutine` peut prendre les valeurs `arcfromfocus`, qui est la valeur par défaut de `currentpolarconicroutine`, ou `arcfromcenter` ; dans le premier cas les angles sont donnés relativement au premier foyer, dans le second, ils sont donnés relativement au centre de l'ellipse.

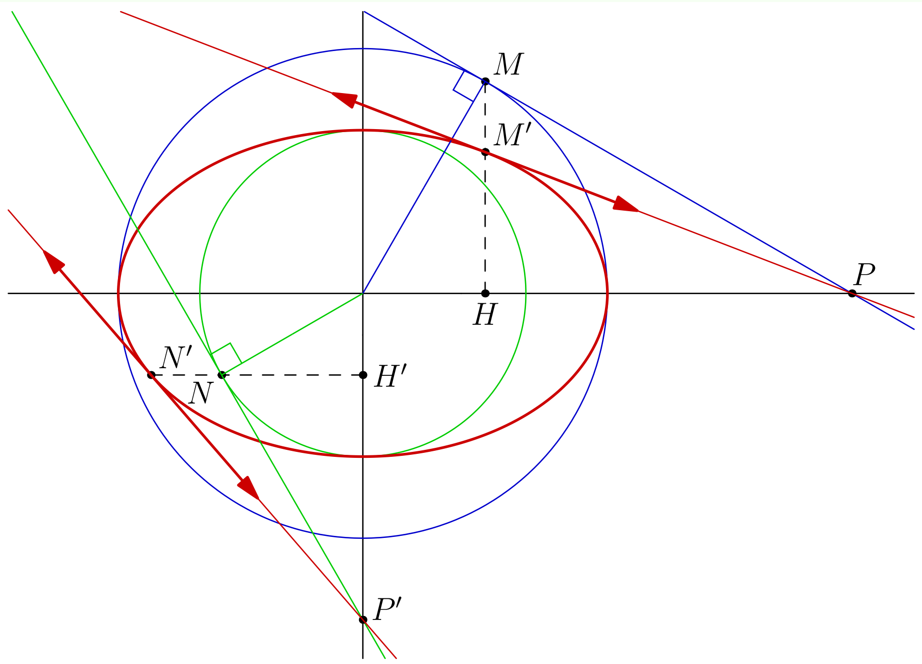
—
```
line[] tangents(ellipse el, point M)
```
Renvoie les tangentes éventuelles à `el` passant par `M`.



```
import geometry; size(7.5cm,0);
point A=(2.5,-1); dot(A);
ellipse C=ellipse(A,3,1); draw(C,linewidth(bp));
path Cp=shift(A)*xscale(2)*scale(3)*unitcircle;
draw(Cp, grey);
for (int i=0; i < 360; i+=45) {
  point M=relpoint(Cp, i/360); dot(M);
  draw(tangents(C, M), 0.8*red);
}
addMargins(10mm,10mm);
```

—
```
line tangent(ellipse el, abscissa x)
```
Retourne la tangente à `el` au point de `el` d'abscisse `x`.

L' exemple suivant illustre la définition d'une ellipse comme image d'un cercle par une affinité et une propriété qui en résulte sur ses tangentes.



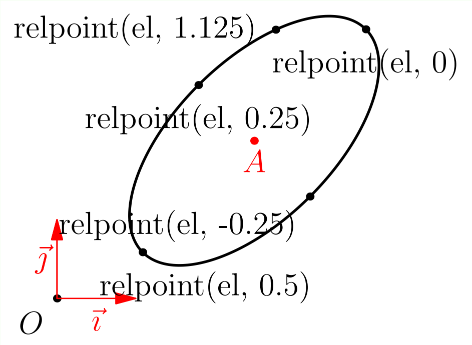

```
import geometry; size(12cm,0); draw(Ox()^^Oy()); real a=3, b=2;
circle C=circle(origin,a), Cp=circle(origin,b);
draw(C, 0.8*blue); draw(Cp, 0.8*green);

transform T=scale(b/a,Ox(),Oy()), Tp=scale(a/b,Oy(),Ox());
ellipse e=T*C; draw(e, bp+0.8*red);

point H=(a/2,0), Hp=(0,-b/2); dot("$H$", H, S); dot("$H'$", Hp);
line L=line(H,false,H+N), Lp=line(Hp,false,Hp+W);
point M=intersectionpoints(L,C)[0], NN=intersectionpoints(Lp,Cp)[0];
point Mp=T*M, NNp=Tp*NN; L=segment(H,M); Lp=segment(Hp,NNp);
dot("$M$", M, NE); dot("$M'$", Mp, NE); dot("$N$", NN, SW); dot("$N'$", NNp, NE);
draw(L, dashed); draw(Lp, dashed);

segment SS=segment(origin,M), SSp=segment(origin,NN);
draw(SS, 0.8*blue); draw(SSp, 0.8*green);

line tgM=tangents(C, M)[0]; point P=intersectionpoint(tgM,Ox());
draw(tgM, 0.8*blue); dot("$P$", P, dir(60));

line tgN=tangents(Cp, NN)[0]; point Pp=intersectionpoint(tgN,Oy());
draw(tgN, 0.8*green); dot("$P'$", Pp, dir(30));

perpendicularmark(tgM,SS, 0.8*blue); perpendicularmark(tgN,SSp, quarter=2, 0.8*green);

line tgMp=line(P, Mp), tgNp=line(Pp, NNp);
draw(tgMp, 0.8*red); draw(tgNp, 0.8*red);

draw(Mp+2tgMp.u--Mp-2tgMp.u, bp+0.8*red, Arrows(3mm));
draw(NNp+2tgNp.u--NNp-2tgNp.u, bp+0.8*red, Arrows(3mm));
addMargins(5mm,5mm);
```

— `point point(explicit ellipse el, real x)`

Retourne le `point` de `el` marquant le même point que le `pair` retourné par le code `point((path)el,x)`.

— `point relpoint(explicit ellipse el, real x)`

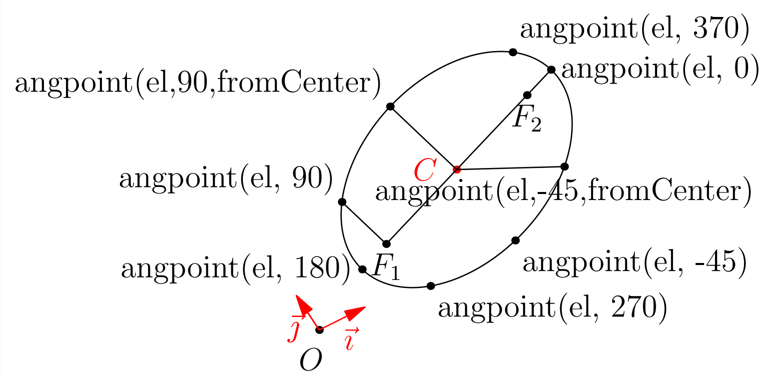
Retourne le point de `el` correspondant à la fraction `x` du périmètre de `el`.




```
import geometry;
size(6cm,0);
show(currentcoordsys, xpen=invisible);
point A=(2.5,2); dot("$A$", A, S, red);
ellipse el=ellipse(A,2,1,45);
draw(el, linewidth(bp));

dot("relpoint(el, 0)", relpoint(el,0), 2S);
dot("relpoint(el, 0.25)", relpoint(el,0.25), 2S);
dot("relpoint(el, 0.5)", relpoint(el,0.5), 2S+E);
dot("relpoint(el, -0.25)", relpoint(el, -0.25), 2SW);
dot("relpoint(el, 1.125)", relpoint(el, 1.125), 2W);
```

— `point angpoint(explicit ellipse el, real x, polarconicroutine polarconicroutine=currentpolarconicroutine)`

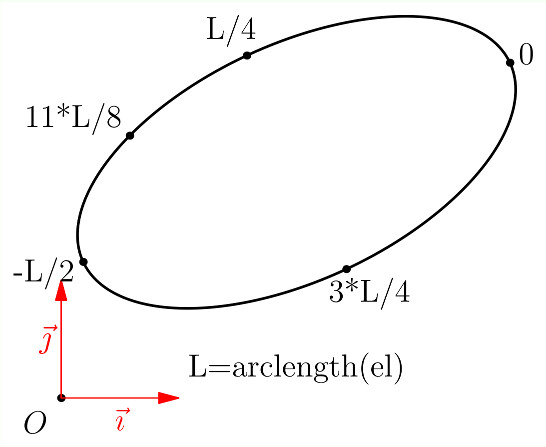
Retourne le point de `el` d'angle `x` degrés depuis le centre de l'ellipse si `polarconicroutine=fromCenter`, depuis le premier foyer si `polarconicroutine=fromFocus`.




```
import geometry; size(10cm,0);
currentcoordsys=cartesiansystem((0,0),i=(1,0.5),j=(-0.5,.75));
show(currentcoordsys, xpen=invisible);
ellipse el=ellipse((point)(4,2),3,2,20);
draw(el); dot("$C$",el.C,2W,red); dot("$F_1$",el.F1,S); dot("$F_2$",el.F2,S);
point P=angpoint(el, 0);  dot("angpoint(el, 0)", P,E);   draw(el.F1--P);
point M=angpoint(el, 90); dot("angpoint(el, 90)", M,NW); draw(el.F1--M);
dot("angpoint(el, 180)", angpoint(el,180), W);
dot("angpoint(el, 270)", angpoint(el,270), SE);
dot("angpoint(el, 370)", angpoint(el,370), NE);
dot("angpoint(el, -45)", angpoint(el,-45), SE);
point P=angpoint(el, 90, fromCenter);  dot("angpoint(el,90,fromCenter)", P,NW);
point Q=angpoint(el, -45, fromCenter); dot("angpoint(el,-45,fromCenter)", Q,S);
draw(el.C--P); draw(el.C--Q);
```

— `point curpoint(explicit ellipse c, real x)`

Retourne le point de `c` dont l'abscisse curviligne est `x`.




```
import geometry; size(7cm,0);
show(currentcoordsys, xpen=invisible);
ellipse el=ellipse((point)(2,2),2,1,25);
draw(el, linewidth(bp));
real L=arclength(el);
dot("0", curpoint(el,0), dir(25));
dot("L/4", curpoint(el,L/4), dir(115));
dot("3*L/4", curpoint(el,3*L/4), -dir(115));
dot("-L/2", curpoint(el, -L/2), -dir(25));
dot("11*L/8", curpoint(el, 11*L/8), dir(145));
label("L=arclength(el)",(2,0.25));
```

8.4. Paraboles

C'est le type `parabola` qui permet d'instancier une parabole. Comme il existe une correspondance biunivoque entre les objets de type `parabola` et ceux de type `conic` ayant une excentricité égale à 1, les objets de type `parabola` héritent des routines et opérateurs définis pour ceux de type `conic`.

8.4.1. Routines de bases

Les routines disponibles pour définir une parabole sont :

— `parabola parabola(point F, line l)`

Renvoie la parabole de foyer `F` et de directrice `l`.

— `parabola parabola(point F, point vertex)`

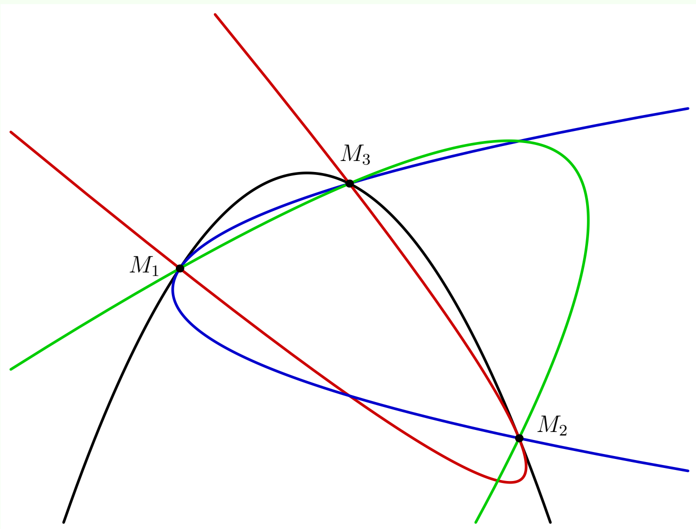
Renvoie la parabole de foyer `F` et de sommet `vertex`.

— `parabola parabola(point F, real a, real angle)`

Renvoie la parabole de foyer `F`, de latus rectum `a` (longueur de la corde focale perpendiculaire à l'axe de la parabole) et dont l'axe fait un angle de `angle` avec l'axe des abscisses du repère dans lequel est défini le point `F`.

— `parabola parabola(point M1, point M2, point M3, line l)`

Renvoie la parabole passant par les points `M1`, `M2`, `M3` et dont la directrice est parallèle à la droite `l`.




```
import geometry;
size(9cm,0);

draw(box((-2,-3),(6,3)), invisible);
point M1=(0,0), M2=(4,-2), M3=(2,1);
pen[] p=new pen[] {black,red,blue,green};
parabola P;
for (int i=0; i < 4; ++i) {
  P=parabola(M1,M2,M3,rotate(45*i)*Ox());
  draw(P, bp+0.8*p[i]);
}
dot(scale(0.75)*"$M_1$", M1, 2*dir(175));
dot(scale(0.75)*"$M_2$", M2, 2*dir(25));
dot(scale(0.75)*"$M_3$", M3, 2*dir(80));
```

8.4.2. Du type « parabola » au type « path »

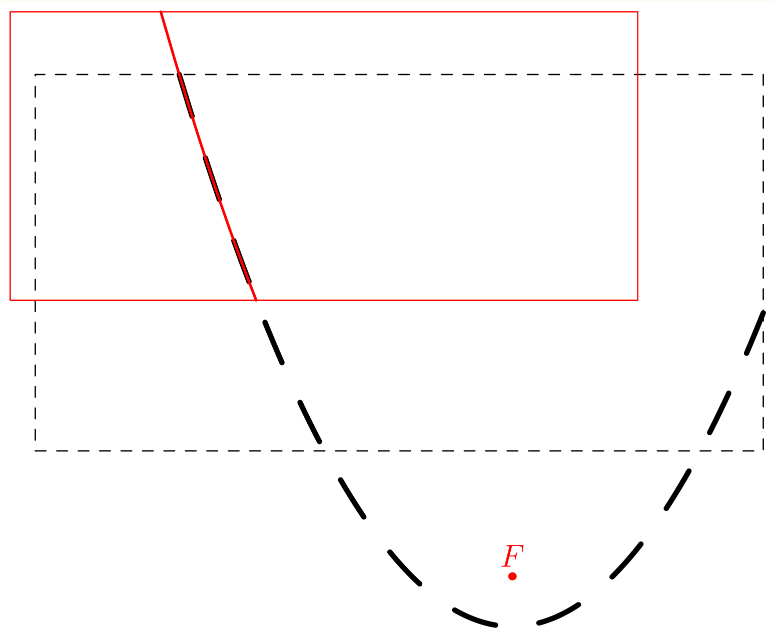
La conversion d'un objet `P` de type `parabola` en `path` s'effectue suivant les règles suivantes :

— le chemin est orienté dans le sens trigonométrique ;

— le chemin est contenu, si c'est possible :

1. dans l'image courante si les variables `P.bmin` et `P.bmax`, de type `pair`, n'ont pas été modifiées ;
2. dans le rectangle `box((P.bmin),box(P.bmax))` dans le cas contraire.

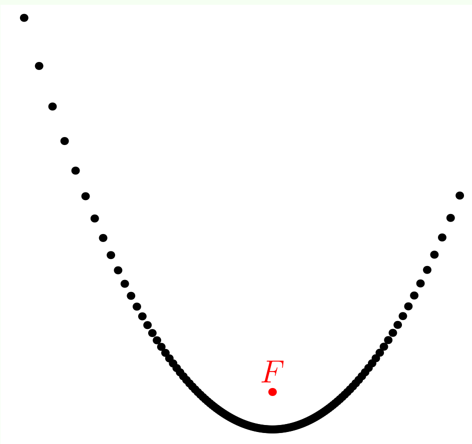
Ainsi dans l'exemple suivant, au moment de la première conversion en chemin, la taille de l'image est symbolisée en pointillé et le chemin ne peut pas contenir dans ce rectangle. Lors de la deuxième conversion, la modification des variables `p.bmin` et `p.bmax` redéfinit la zone de conversion ; elle est tracée en rouge avec la portion de parabole correspondante.




```
import geometry;
size(10cm);

point F=(2,-1.5);
dot("$F$",F,N,red);
parabola p=parabola(F,0.2,90);

draw(box((0.1,-1),(3,0.5)), dashed);
draw((path)p, 2*bp+dashed);

p.bmin=(0,-0.4);
p.bmax=(2.5,0.75);
draw(box(p.bmin,p.bmax), red);
draw((path)p, bp+red);
```

— le nombre de nœuds du chemin est fonction des angles, donnés relativement au foyer en degrés, des extrémités du chemin ; il est calculé par la routine `int parabolanodesnumber(parabola p, real angle1, real angle2)` qui dépend elle-même de la variable `parabolanodes`

— les nœuds du chemin sont définis en coordonnées polaires avec des angles donnés relativement au foyer de la parabole et uniformément répartis dans l'intervalle dont les extrémités sont retournés par la routine `real[] bangles(picture pic=curre`



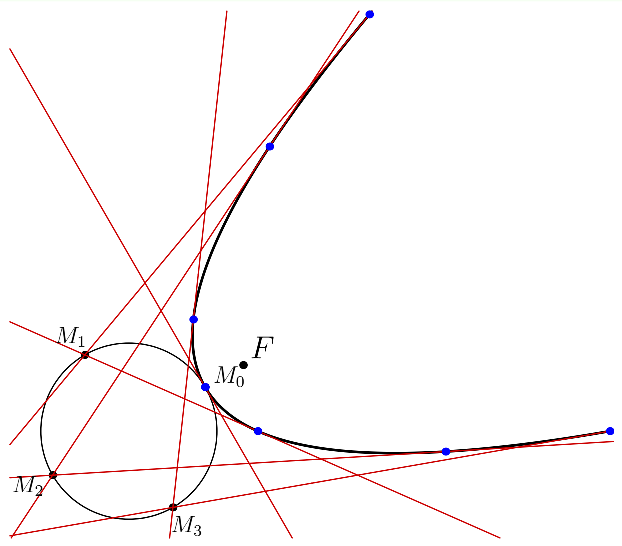

```
import geometry;
size(6cm);

point F=(2,-1.5);
dot("$F$",F,N,red);
parabola p=parabola(F,0.2,90);

draw(box((0.6,-1.75),(3,0.5)), invisible);
parabolanodesnumberfactor=50;
dot((path)p);
```

8.4.3. Autres routines

En dehors des routines s'appliquant aux objets de type `conic`, voici la liste des routines spécifiques aux objets de type `parabola`.

— `line[] tangents(parabola p, point M)`

Retourne les tangentes éventuelles à `p` passant par `M`.




```
import geometry; size(8cm,0);
point F=(0,0); dot("$F$", F, NE);
parabola p=parabola(F, 0.1, 30);
draw(p, linewidth(bp));
point C=shift(2*(p.V-p.F))*p.V;
circle cle=circle(C, 0.2);
draw(cle);
for (int i=0; i < 360; i+=90) {
  point M=C+0.2*dir(i+30);
  dot(scale(0.75)*("$M_"+(string)(i/90)+"$"),
      M, unit(M-C));
  line[] tgt=tangents(p, M);
  draw(tgt, 0.8*red);
  for (int i=0; i < tgt.length; ++i) {
    dot(intersectionpoints(p, tgt[i]), blue);
  } }
```

— `line tangent(parabola p, abscissa x)`

Retourne la tangente à `p` au point de `p` d'abscisse `x`.

— `point point(explicit parabola p, real x)`

Retourne le `point` de `p` marquant le même point que le `pair` retourné par le code `point((path)p,x)`.

— `point relpoint(explicit parabola p, real x)`

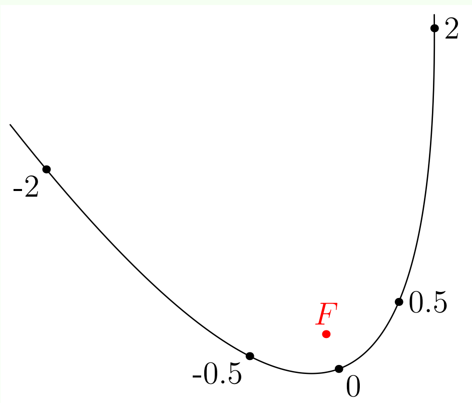
Retourne le `point` marquant le même `pair` retourné par le code `relpoint((path)p,x)`.

— `point angpoint(explicit parabola p, real x)`

Retourne le point de `p` d'angle `x` degrés.

— `point curpoint(explicit parabola p, real x)`

Retourne le point de `p` dont l'abscisse curviligne est `x`, l'origine étant le sommet de la parabole.

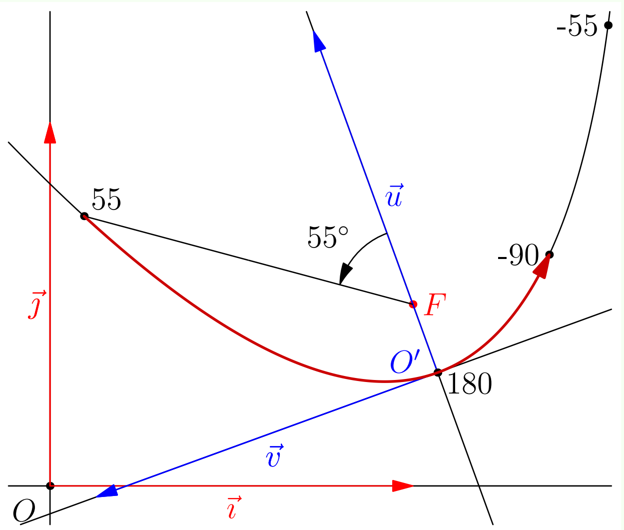



```
import geometry; size(6cm);

point F=(1,-1.5); dot("$F$",F,N,red);
parabola p=parabola(F,0.2,110); draw(p);

dot("0",curpoint(p,0),SE);
dot("0.5",curpoint(p,0.5));
dot("-0.5",curpoint(p,-0.5),SW);
dot("-2",curpoint(p,-2),SW);
dot("2",curpoint(p,2),E);
```

— Il est possible de récupérer un arc de parabole sous forme de `path` grâce à la routine suivante : `path arcfromfocus(conic`

Bien que cette routine soit disponible pour tout type de conique son utilisation n'a réellement d'intérêt que pour les paraboles et les hyperboles ; les arcs d'ellipse possèdent un type spécifique décrit dans la section Arcs.

Voici un exemple illustrant l'utilisation de la routine `arcfromfocus` avec un parabole.




```
import geometry;
size(8cm);
show(currentcoordsys);

point F=(1,0.5); dot("$F$",F,E,red);
parabola p=parabola(F,0.2,110); draw(p);

coordsys Rp=canonicalcartesiansystem(p);
show(Label("$O'$",align=NW+W,blue), Label("$\vec{u}$",blue),
     Label("$\vec{v}$",blue), Rp, ipen=blue);

dot("180", angpoint(p,180), dir(-30));
point P=angpoint(p,55); dot("55",P,NE);

segment s=segment(F,P); draw(s);
line l=line(F,F+Rp.i);
markangle("$"+(string)degrees(l,s)+"^\circ$",l,(line)s,Arrow);

dot("-55", point(arcfromfocus(p,-55,-55,1),0), W);
dot("-90", point(arcfromfocus(p,-90,-90,1),0), W);
draw(arcfromfocus(p,55,-90), bp+0.8*red, Arrow(3mm));
```

8.5. Hyperboles

C'est le type `hyperbola` qui permet d'instancier une hyperbole. Comme il existe une correspondance biunivoque entre les objets de type `hyperbola` et ceux de type `conic` ayant une excentricité strictement supérieure à 1, les objets de type `hyperbola` héritent des routines et opérateurs définis pour ceux de type `conic`.

8.5.1. Routines de bases

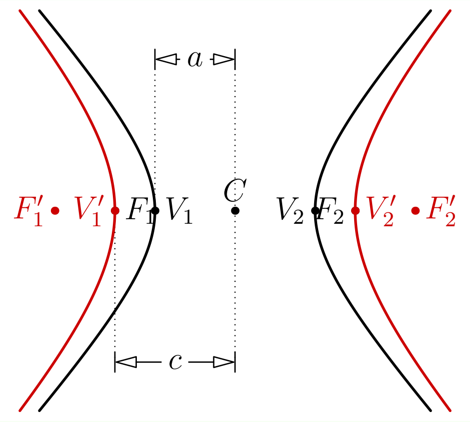
Les routines disponibles pour définir une hyperbole sont :

— `hyperbola hyperbola(point P1, point P2, real ae, bool byfoci=byfoci)`

Si `byfoci=true` : renvoie l'hyperbole de demi grand axe `ae` et de foyers `P1` et `P2` ;

Si `byfoci=false` : renvoie l'hyperbole d'excentricité `ae` et de sommets `P1` et `P2` ;

Pour plus de lisibilité, les constantes `byfoci` et `byvertices` sont définies, elles ont pour valeurs `true` et `false` respectivement.



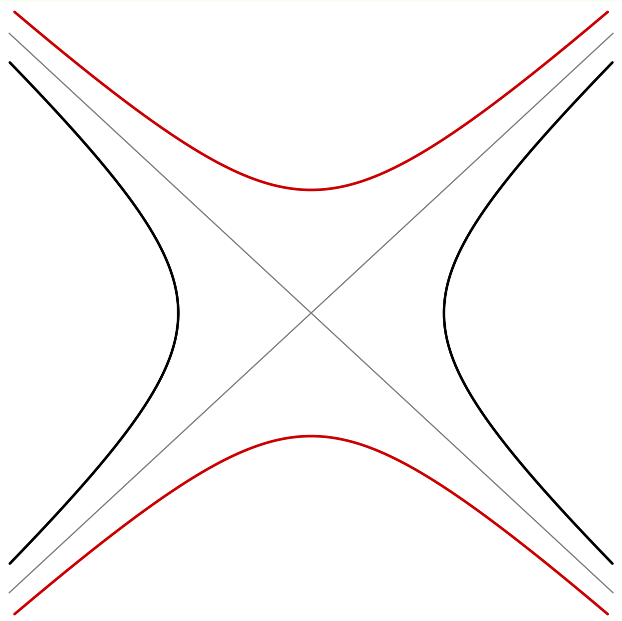

```
import geometry; size(6cm);
pen Red=0.8*red; point P1=(-3,0), P2=(3,0);
draw(box((-5,-5),(5,5)), invisible);
hyperbola Hf=hyperbola(P1,P2,2);
draw(Hf, linewidth(bp)); dot("$C$", Hf.C, N);
dot("$F_1$", Hf.F1); dot("$F_2$", Hf.F2, W);
dot("$V_1$", Hf.V1, E); dot("$V_2$", Hf.V2, W);
distance("$a$", Hf.C, Hf.V1, 2cm, joinpen=dotted);
distance("$c$", Hf.C, Hf.F1, -2cm, joinpen=dotted);
hyperbola Hv=hyperbola(P1,P2,1.5,byvertices);
draw(Hv, bp+Red);
dot("$V'_1$",Hv.V1, W, Red); dot("$V'_2$",Hv.V2, Red);
dot("$F'_1$",Hv.F1, W, Red); dot("$F'_2$",Hv.F2, Red);
```

— `hyperbola hyperbola(point C, real a, real b, real angle=0)`

Renvoie l'hyperbole de centre `C`, de demi grand axe `a` le long de `C--C+dir(angle)` et de « demi petit axe » `b`.

— `hyperbola conj(hyperbola h)`

Retourne l'hyperbole conjuguée de `h`.



```
import geometry;
size(8cm);

point P1=(-3,0), P2=(3,0);
draw(box((-5,-5),(5,5)), invisible);

hyperbola H=hyperbola(P1,P2,2.2);

draw(H, linewidth(bp));
draw(H.A1^^H.A2, grey);

draw(conj(H), bp+0.8*red);
```

8.5.2. Du type « hyperbola » au type « path »

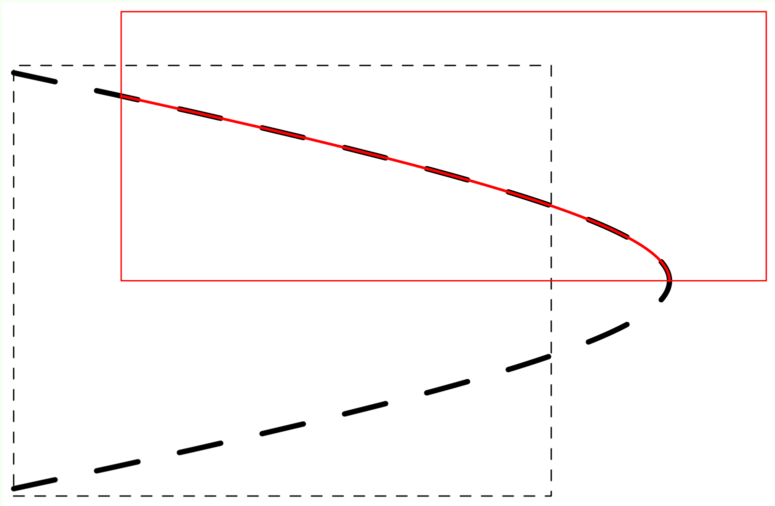
La conversion d'un objet `H` de type `hyperbola` en `path` s'effectue suivant les règles suivantes :

— le chemin est constitué de la branche d'hyperbole de foyer `H.F1`, il est orienté dans le sens trigonométrique ;

— le chemin est contenu, si c'est possible :

1. dans l'image courante si les variables `H.bmin` et `H.bmax`, de type `pair`, n'ont pas été modifiées ;
2. dans le rectangle `box((H.bmin),box(H.bmax))` dans le cas contraire.

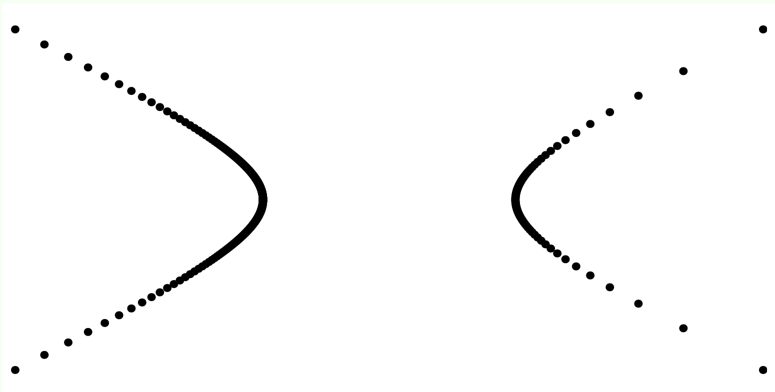
Ainsi dans l'exemple suivant, au moment de la première conversion en chemin, la taille de l'image est symbolisée en pointillé et le chemin ne peut pas contenir dans ce rectangle. Lors de la deuxième conversion, la modification des variables `H.bmin` et `H.bmax` redéfinit la zone de conversion ; elle est tracée en rouge avec la portion d'hyperbole correspondante.



```
import geometry;
size(10cm,0);

point P1=(-3,0), P2=(3,0);
hyperbola H=hyperbola(P1,P2,2.95);

draw(box((-6,-1),(-3.5,1)), dashed);
draw((path)H, 2*bp+dashed);

H.bmin=(-5.5,0);
H.bmax=(-2.5,1.25);
draw(box(H.bmin,H.bmax), red);
draw((path)H, bp+red);
```

— le nombre de nœuds du chemin est fonction des angles, donnés relativement au foyer en degrés, des extrémités du chemin ; il est calculé par la routine `int hyperbolanodesnumber(hyperbola p, real angle1, real angle2)` qui dépend elle-même de la variable `hyperbolano`

— les nœuds du chemin sont définis en coordonnées polaires avec des angles donnés relativement au foyer principale `H.F1` de l'hyperbole et uniformément répartis dans l'intervalle dont les extrémités sont retournés par la routine `real[][] bangles(picture pic=currentpicture, hyperbola p)`.



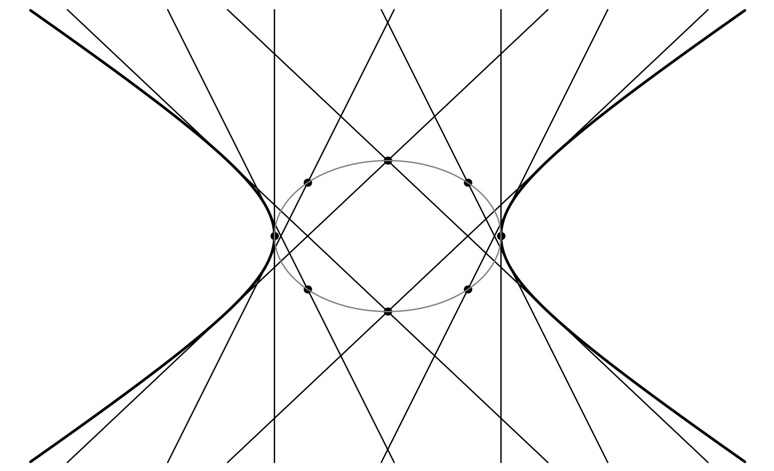
```
import geometry;
size(10cm,0);

point P1=(-3,0), P2=(3,0);
draw(box((-8,-4),(8,4)), invisible);

dot((path)hyperbola(P1,P2,2.7));

hyperbolanodesnumberfactor=30;
dot((path)hyperbola(P2,P1,2.7));
```

8.5.3. Autres routines

En dehors des routines s'appliquant aux objets de type `conic`, voici la liste des routines spécifiques aux objets de type `hyperbola`.

— `line[] tangents(hyperbola h, point M)`

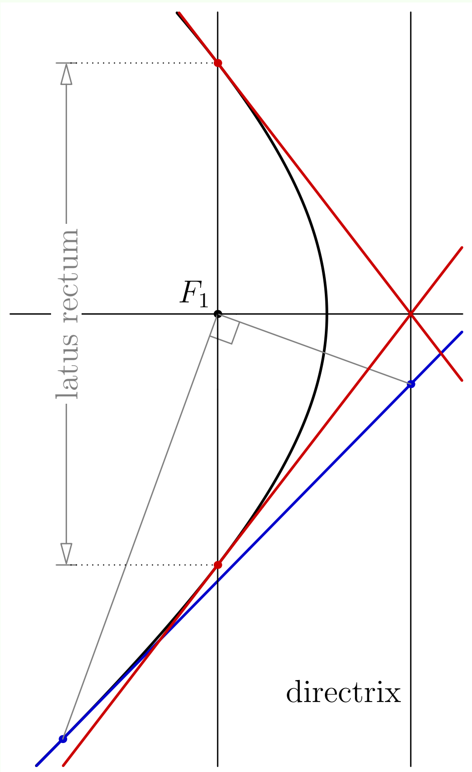
Retourne les tangentes éventuelles à `h` passant par `M`.



```
import geometry; size(10cm,0);
draw(box((-5,-3),(5,3)), invisible);
hyperbola h=hyperbola(origin,1.5,1);
draw(h, linewidth(bp));

for (int i=0; i < 360; i +=45 ) {
  point M=(1.5*Cos(i), Sin(i));
  dot(M); draw(tangents(h,M)); }
draw(ellipse(origin,1.5,1), grey);
```

— `line tangent(hyperbola h, abscissa x)`

Retourne la tangente à `h` au point d'abscisse `x`.




```
import geometry; size(0,10cm);
pen bl=0.8blue, re=0.8*red;
draw(box((-2.25,-1.5),(-0.75,1)), invisible);
hyperbola h=hyperbola(origin,1.2,1);
draw((path)h, linewidth(bp));
draw("directrix", h.D1); dot("$F_1$", h.F1, NW);

line axis=line(h.F1,h.F2); draw(axis);
point M=point(h,angabscissa(70)); dot(M, bl);
line tgt=tangent(h,angabscissa(70)); draw(tgt, bp+bl);
point P=intersectionpoint(tgt,h.D1); dot(P, bl);
draw(P--h.F1--M, grey); markrightangle(P,h.F1,M, grey);

line lr=perpendicular(h.F1, axis); draw(lr);
point[] plr=intersectionpoints(h,lr);
dot(plr, re);
distance(Label("latus rectum",Fill(white)),
         plr[0], plr[1], -2cm, grey, dotted);
for (int i=0; i < 2; ++i) {
  draw(tangents(h,plr[i])[0], bp+re); }
```

— `point point(explicit hyperbola h, real x)`

Retourne le `point` de `h` marquant le même point que le `pair` retourné par le code `point((path)h,x)`.

— `point relpoint(explicit hyperbola h, real x)`

Retourne le `point` marquant le même `pair` retourné par le code `relpoint((path)h,x)`.

— `point angpoint(explicit hyperbola h, real x, polarconicroutine polarconicroutine=currentpolarconicroutine)`

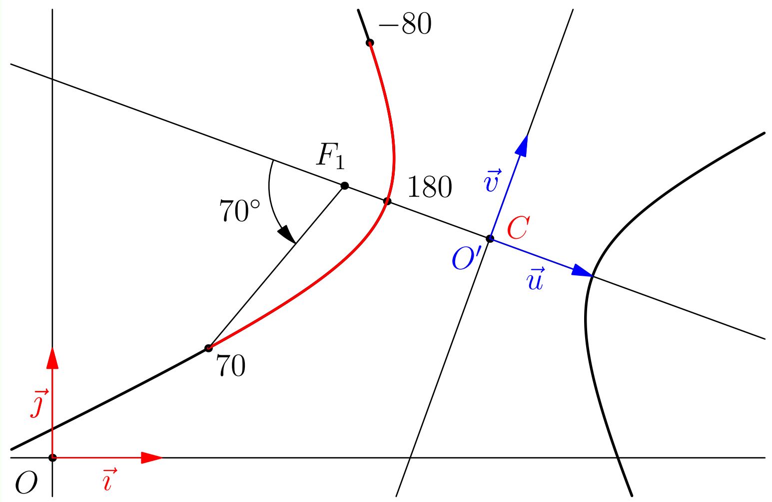
Retourne le point de `h` d'angle `x` degrés depuis le centre de l'hyperbole si `polarconicroutine=fromCenter`, depuis le premier foyer si `polarconicroutine=fromFocus`. Deux exemples sont donnés ci-après.

— Il est possible de récupérer un arc de d'hyperbole grâce aux deux routines suivantes :

1. `path arcfromfocus(conic co, real angle1, real angle2, int n=400, bool direction=CCW)`

   Bien que cette routine soit disponible pour tout type de conique son utilisation n'a réellement d'intérêt que pour les paraboles et les hyperboles, les arcs d'ellipse possèdent un type spécifique décrit dans la section Arcs.

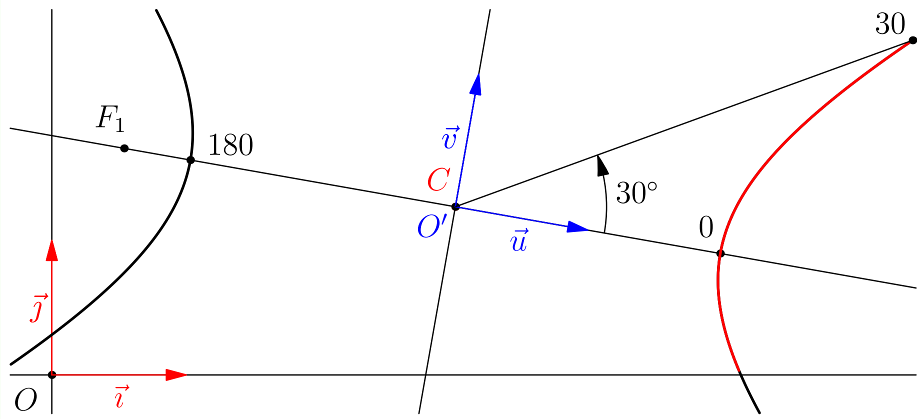
   Voici un exemple illustrant l'utilisation de la routine `arcfromfocus` avec une hyperbole.




```
import geometry; size(10cm,0);
point C=(4,2); dot("$C$", C, E+NE, red);
hyperbola H=hyperbola(C,1,1,-20); draw(H, linewidth(bp));
coordsys R=currentcoordsys; show(R);
coordsys Rp=canonicalcartesiansystem(H);
show(Label("$O'$",align=SW,blue), Label("$\vec{u}$",blue), Label("$\vec{v}$",blue),
     Rp, ipen=blue);
dot("$180$", angpoint(H,180), N+2E);
dot("$-80$", angpoint(H,-80), NE);
point P=angpoint(H,70); dot("$70$", P, SE);
draw(arcfromfocus(H,70,-80), bp+red);
segment s=segment(H.F1,P); draw(s); line l=line(H.F1,H.F1-Rp.i);
dot("$F_1$", H.F1, N+NW); markangle("$70^\circ$",l,(line)s,Arrow);
addMargins(rmargin=3cm);
```

2. `path arcfromcenter(hyperbola h, real angle1, real angle2,`
   `int n=hyperbolanodesnumber(h,angle1,angle2), bool direction=CCW)`

   Voici un exemple illustrant l'utilisation de la routine `arcfromcenter` avec une hyperbole.




```
import geometry; size(12cm);
coordsys R=currentcoordsys; show(R);
point C=(3,1.25); dot("$C$", C, 2*dir(120), red);
hyperbola H=hyperbola(C, 2, 1.5, -10); draw(H, linewidth(bp));
coordsys Rp=canonicalcartesiansystem(H);
show(Label("$O'$", align=SW,blue), Label("$\vec{u}$",blue),
     Label("$\vec{v}$",blue), Rp, ipen=blue);
dot("$0$", angpoint(H,0,fromCenter), 2*dir(120));
dot("$180$", angpoint(H,180,fromCenter), 2*dir(30));
draw(arcfromcenter(H,-20,30), bp+red); dot("$F_1$", H.F1, N+NW);
point P=angpoint(H,30,fromCenter); dot("$30$", P, NW);
segment s=segment(C, P); draw(s);
markangle("$30^\circ$", Ox(Rp), (line) s, radius=2cm, Arrow);
```

9. Arcs

Le type `arc` permet d'instancier un arc orienté d'ellipse. La principale routine pour définir un tel arc est décrite ci-dessous.

```
arc arc(ellipse el, real angle1, real angle2,
polarconicroutine polarconicroutine=polarconicroutine(el),
bool direction=CCW)
```

Retourne un arc de l'ellipse `el` compris entre les angles (en degrés) `angle1` et `angle2` parcouru dans le sens `direction` et donnés relativement au premier foyer si `polarconicroutine=fromFocus`, relativement au centre de l'ellipse si `polarconicroutine=f`

La routine `polarconicroutine polarconicroutine(conic co)` utilisée ici pour déterminer la valeur par défaut du paramètre `polarconicroutine` renvoie dans le cas présent `fromCenter` si `co` représente un cercle, `currentpolarconicroutine`, qui vaut `fromFocus` par défaut, si `co` représente une ellipse.

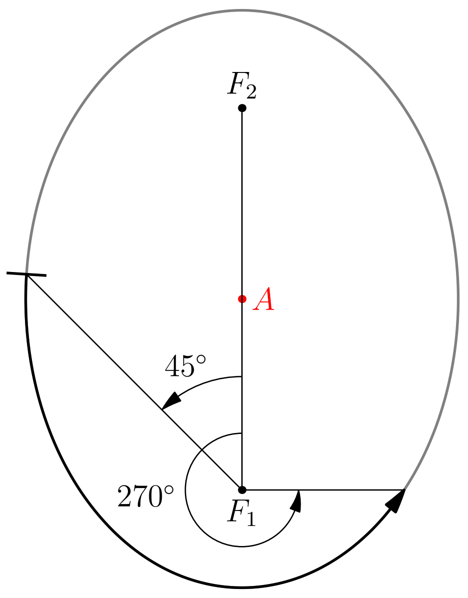
Il est important de noter que, lors du tracé d'un arc, la valeur de la variable `addpenarc` est ajoutée au stylo utilisé. Par défaut cette variable a pour valeur `squarecap`, afin d'avoir les extrémités droites, ce qui rend l'affichage d'un arc en pointillé inefficient.

Pour contourner ce problème il y a trois solutions :

1. écrire `draw(un_arc, roundcap+dotted);` au lieux de `draw(un_arc, dotted);` ;
2. affecter la valeur `nullpen` à `addpenarc`.
3. contacter l'auteur de l'extension *geometry.asy* pour lui faire connaître son désaccord quant à la valeur par défaut de `addpenline` ;

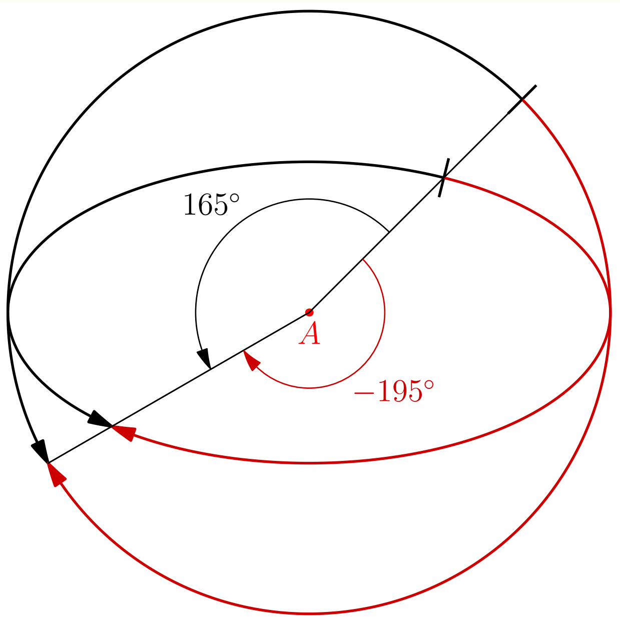
Voici quelques exemples qui illustre l'utilisation de la routine `arc(ellipse,real,real,polarconicroutine,bool)`

— L'exemple suivant montre comment obtenir un arc d'ellipse dont les angles sont donnés relativement à son premier foyer, ce qui est le comportement par défaut. On notera l'utilisation de la routine `markarc` qui sera décrite ultérieurement.



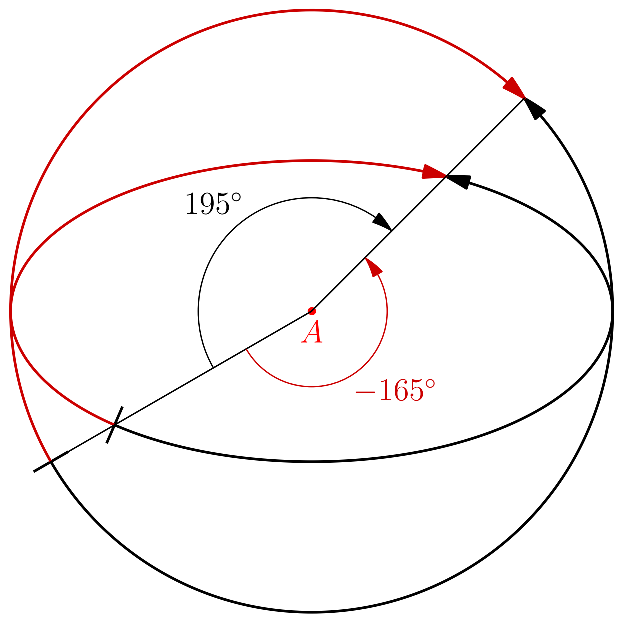

```
import geometry; size(6cm,0);
real a=2, b=1.5;
point A=(1,1); dot("$A$", A, red);
ellipse EL=ellipse(A,a,b,90); draw(EL, bp+grey);
dot("$F_1$", EL.F1, S); dot("$F_2$", EL.F2, N);
draw(EL.F1--EL.F2);
arc AE=arc(EL, 45, 270);
draw(AE, linewidth(bp), Arrow(3mm), BeginBar);
point Bp=point(AE, 0), Ep=relpoint(AE,1);
draw(EL.F1--Bp); draw(EL.F1--Ep);
markangle(format("$%0g^\circ$",AE.angle1),
          EL.F2,EL.F1,Bp, radius=1.5cm, Arrow);
markangle(Label(format("$%0g^\circ$",AE.angle2),
                Relative(0.35)),
          EL.F2, EL.F1, Ep, radius=0.75cm, Arrow);
```

— L'exemple suivant montre l'effet des paramètres `polarconicroutine` et `direction`. On notera l'utilisation de la routine `degrees(arc)` qui sera décrite ultérieurement.




```
import geometry; size(8cm,0);
real a=2, b=1;
point A=(1,1); dot("$A$",A,S,red);
ellipse EL=ellipse(A,a,b);
arc AE=arc(EL, 45, 210, fromCenter);
draw(AE, linewidth(bp), Arrow(3mm), BeginBar);
arc AEp=arc(EL, 45, 210, fromCenter, CW);
draw(AEp, bp+0.8*red, Arrow(3mm));
circle C=circle(A,a); arc AC=arc(C, 45, 210);
draw(AC, linewidth(bp), Arrow(3mm), BeginBar);
arc ACp=arc(C, 45, 210, CW);
draw(ACp, bp+0.8*red, Arrow(3mm));
markarc(format("$%0g^\circ$",degrees(AC)),
        AC, radius=1.5cm, Arrow);
markarc(format("$%0g^\circ$",degrees(ACp)),
        ACp, markpen=0.8*red, Arrow);
```

— L'exemple suivant reprend le code précédant en permutant les angles 45° et 210°.




```
import geometry; size(8cm,0);
real a=2, b=1;
point A=(1,1); dot("$A$",A,S,red);
ellipse EL=ellipse(A,a,b);
arc AE=arc(EL, 210, 45, fromCenter);
draw(AE, linewidth(bp), Arrow(3mm), BeginBar);
arc AEp=arc(EL, 210, 45, fromCenter, CW);
draw(AEp, bp+0.8*red, Arrow(3mm));
circle C=circle(A,a); arc AC=arc(C, 210, 45);
draw(AC, linewidth(bp), Arrow(3mm), BeginBar);
arc ACp=arc(C, 210, 45, CW);
draw(ACp, bp+0.8*red, Arrow(3mm));
markarc(format("$%0g^\circ$",degrees(AC)),
        AC, radius=1.5cm, Arrow);
markarc(format("$%0g^\circ$",degrees(ACp)),
        ACp, markpen=0.8*red, Arrow);
```

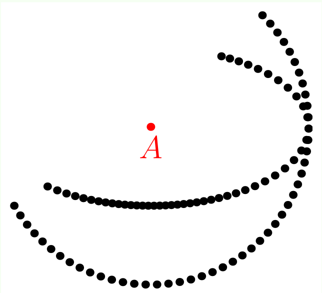
9.1. Du type « arc » au type « path »

La conversion d'un objet `A` de type `arc` en `path` s'effectue suivant les règles suivantes :

— le chemin est orienté dans le sens `A.direction` qui est la direction passée en paramètre pour définir l'arc `A` ;

— le nombre de nœuds du chemin est calculé par la routine `int arcnodesnumber(explicit arc a)` qui dépend elle-même de la variable `ellispenodesnumberfactor` ;

— les nœuds du chemin sont définis en coordonnées polaires avec des angles donnés relativement au premier foyer ou au centre de l'ellipse suivant la valeur de `A.polarconicroutine` et uniformément répartit dans un intervalle adéquat.

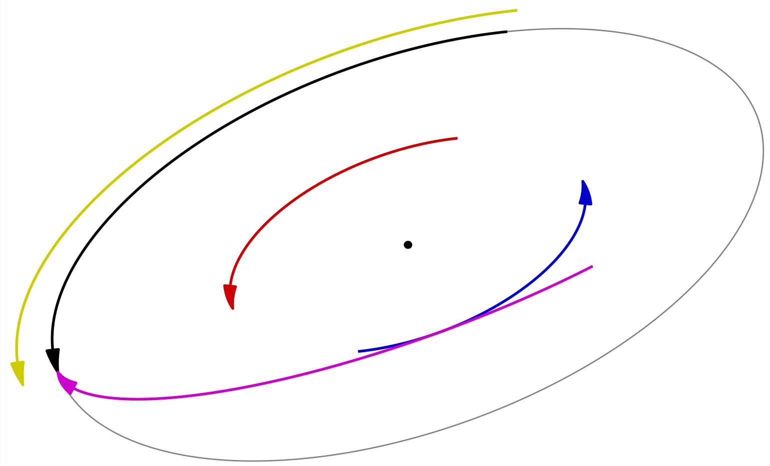



```
import geometry;
size(4cm,0);
ellipsenodesnumberfactor=100;
point A=(1,1); dot("$A$",A,S,red);
ellipse EL=ellipse(A,2,1);
dot((path)arc(EL, 210, 45, fromCenter));
circle C=circle(A,2);
dot((path)arc(C, 210, 45));
```

9.2. Les opérateurs

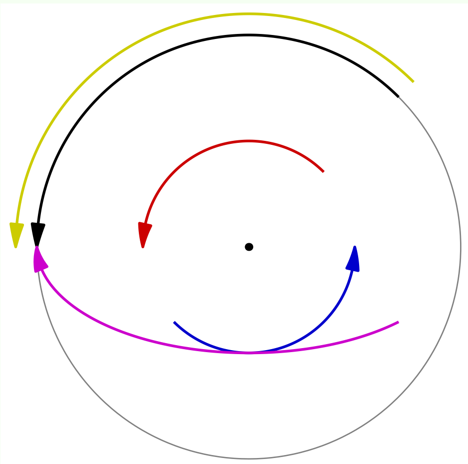
— `arc operator *(transform t, explicit arc a)`

Autorise le code `transfomr*arc` dont le comportement est sans surprise. Dans l'exemple suivant les arcs en couleur sont des images de l'arc noir par des transformations affines.



```
import geometry; size(10cm,0);
currentcoordsys=rotate(20)*defaultcoordsys;
point C=(1,1); dot(C);
ellipse el=ellipse(C,2,1); draw(el, grey);
arc AE=arc(el, 45, 180, fromCenter); draw(AE, linewidth(bp), Arrow(3mm));
draw(scale(0.5,C)*AE, bp+0.8red, Arrow(3mm));
draw(scale(-0.5,C)*AE, bp+0.8blue,Arrow(3mm));
draw(scale(1.1,C)*AE, bp+0.8*yellow, Arrow(3mm));
transform t=scale(-0.5,line(el.F1,el.F2), line(S,N));
draw(t*AE, bp+0.8(red+blue), Arrow(3mm));
```

Le même exemple en partant d'un arc de cercle :



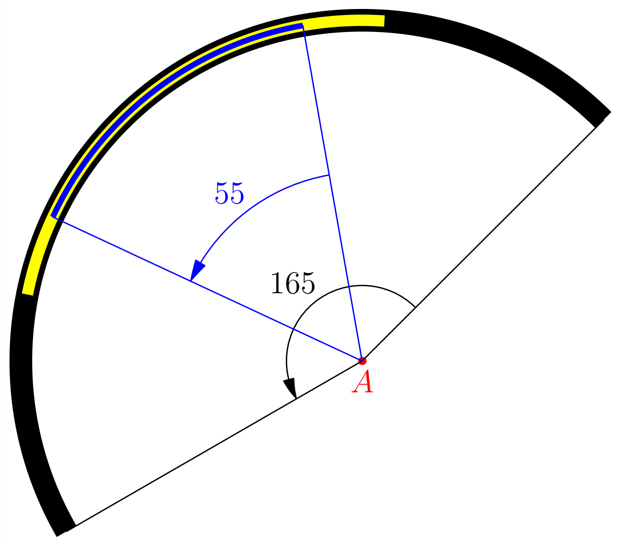
```
import geometry; size(6cm,0);
point C=(0,0); dot(C);
ellipse el=circle(C,2); draw(el, grey);
arc AE=arc(el, 45, 180, fromCenter);

draw(AE, linewidth(bp), Arrow(3mm));
draw(scale(0.5,C)*AE, bp+0.8red, Arrow(3mm));
draw(scale(-0.5,C)*AE, bp+0.8blue,Arrow(3mm));
draw(scale(1.1,C)*AE, bp+0.8*yellow, Arrow(3mm));

transform t=scale(-0.5,Ox(), Oy());
draw(t*AE, bp+0.8(red+blue), Arrow(3mm));
```

— `arc operator *(real x, explicit arc a)`

Autorise le code `real*arc`.

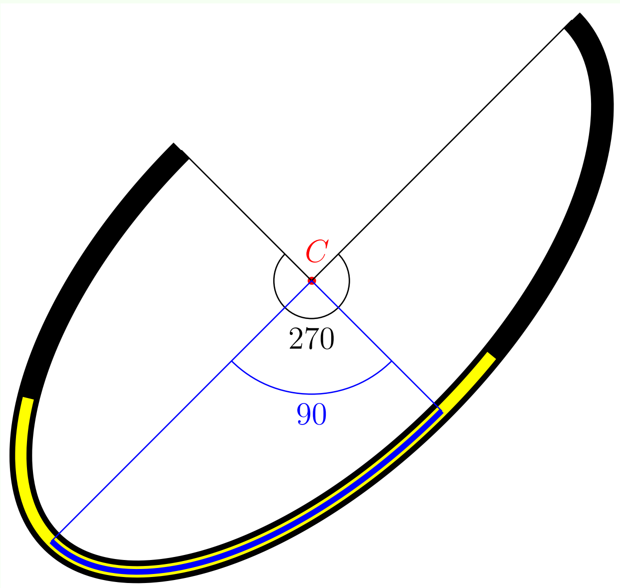
Renvoie l'arc `a` avec les angles `a.angle1-(x-1)*degrees(a)/2` et `a.angle2+(x-1)*degrees(a)/2`. L'opérateur `/(explicit a` est aussi défini.

Dans l'exemple suivant l'arc jaune est obtenu en multipliant l'arc noir par $0,5$ et l'arc bleu en le divisant par 3.



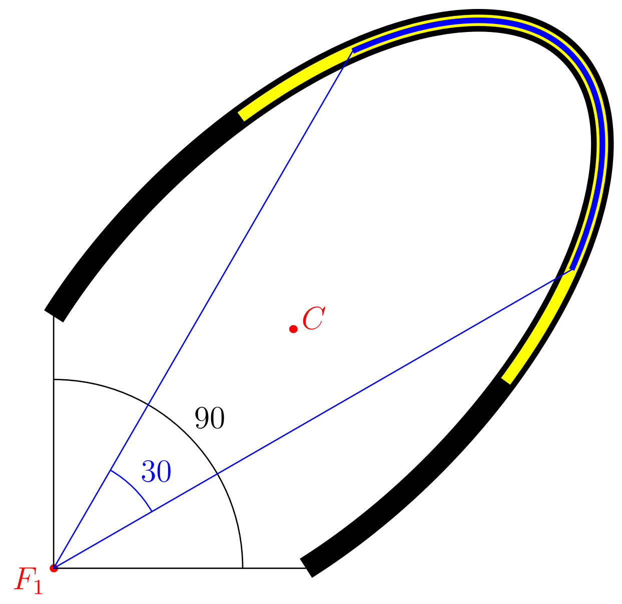

```
import geometry; size(8cm,0);

point A=(1,1); dot("$A$",A,S,red);
arc C=arc(circle(A,2), 45, 210);
draw(C,linewidth(3mm));
markarc(format("%0g",degrees(C)), C, Arrow);

draw(0.5*C,1.5mm+yellow);

arc Cp=C/3;
draw(Cp, 0.75mm+blue);
markarc(format("%0g",degrees(Cp)),
        radius=25mm, Cp, blue, Arrow);
```

La même chose en partant d'un arc d'ellipse défini depuis le centre de l'ellipse :




```
import geometry; size(8cm,0);
point C=(1,1); dot("$C$", C, 2*dir(80), red);
arc a=arc(ellipse(C,2,1,45),90,0,fromCenter);
draw(a, linewidth(3mm));
markarc(format("%0g", degrees(a)),
        radius=-0.5*markangleradius(), a);

draw(0.5*a, 1.5mm+yellow);

arc ap=a/3;
draw(ap, 0.75mm+blue);
markarc(format("%0g", degrees(ap)),
        radius=1.5*markangleradius(),ap,blue);
```

Enfin, dans l'exemple suivant, l'arc est défini depuis le premier foyer de l'ellipse :




```
import geometry; size(8cm,0);
point C=(1,1); dot("$C$", C, dir(30), red);

arc a=arc(ellipse(C,2,1,45), -45, 45);
draw(a, linewidth(3mm));
dot("$F_1$", a.el.F1, dir(210), red);
markarc(format("%0g", degrees(a)),
        radius=2.5*markangleradius(), a);

draw(0.5*a, 1.5mm+yellow);

arc ap=a/3;
draw(ap, 0.75mm+blue);
markarc(format("%0g", degrees(ap)),
        radius=1.5*markangleradius(),ap,blue);
```

— `arc operator +(explicit arc a, point M)`

Autorise le code `arc+point` qui est un alias de `shift(point)*arc`.
Les opérateurs `-(explicit arc,point)`, `+(explicit arc,vector)` et `-(explicit arc,vector)` sont aussi définis.

— `bool operator @(point M, arc a)`

Autorise le code `point @ arc`. Retourne `true` si et seulement si le point `M` appartient à l'arc `a`.

— `arc operator *(inversion i, segment s)`

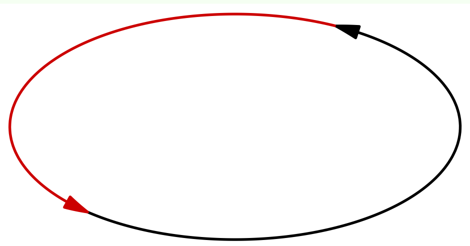
Autorise le code `inversion*segment`. Retourne l'image de `s` par l'inversion `i` ; on peut voir une illustration de `inversion*segment` dans la section Inversions.

9.3. Autres routines

En plus des routines décrites dans cette section s'ajoutent les routines pour localiser un point sur un objet de type `arc` ; elles sont décrites dans la section Abscisses.

— `arc complementary(arc a)`

Retourne le complémentaire de l'arc `a`.



```
import geometry;
size(6cm,0);
ellipse EL=ellipse(origin,2,1);
arc AE=arc(EL, 210, 45, fromCenter);
draw(AE, linewidth(bp), Arrow(3mm));
draw(complementary(AE), bp+0.8*red, Arrow(3mm));
```

— `arc reverse(arc a)`

Retourne l'arc inverse de `a` comme le ferait la routine `reverse(path)`.

— `real degrees(arc a)`

Retourne la mesure en degrés dans $[-360\,;\,360]$ de l'arc orienté représenté par `a`.
La routine `angle(arc)` est aussi définie pour une mesure en radians.

— `real arclength(arc a)`

Retourne la longueur de l'arc représenté par `a`.

—
```
void markarc(picture pic=currentpicture,
Label L="", int n=1, real radius=0, real space=0,
arc a, pen sectorpen=currentpen, pen markpen=sectorpen,
margin margin=NoMargin, arrowbar arrow=None, marker marker=nomarker)
```

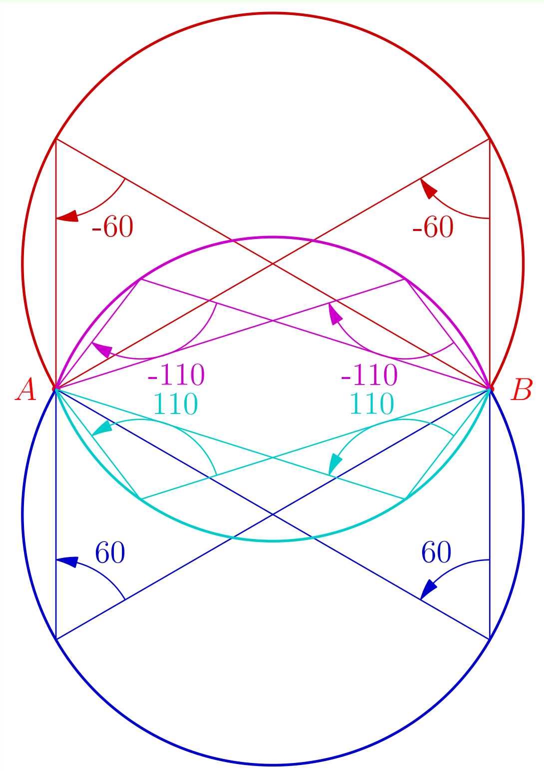
Permet de marquer l'angle représenté par `a` avec un arc de cercle.
Le paramètre `sectorpen` est le stylo utilisé pour marquer les segments qui relient le centre ou le foyer de l'arc avec ses extrémités.
Le paramètre `markpen` est le stylo utilisé pour tracer l'arc de cercle qui peut à son tour être marqué à l'aide du paramètre `marker`.
Des exemples d'utilisation ont déjà été donnés.

—
```
point[] intersectionpoints(arc a1, arc a2)
```

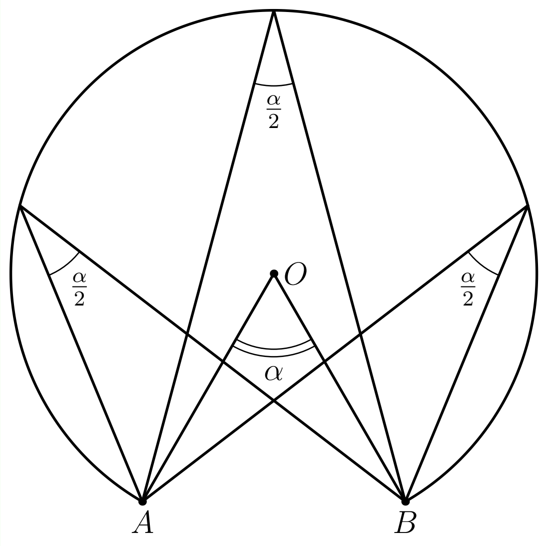
Retourne, sous forme de tableau, les points d'intersection de deux arcs. Les routines d'intersections d'un objet de type `arc` avec d'autres objets définis par l'extension *geometry.asy* sont aussi définies ; par exemple `intersectionpoints(conic co,` `intersectionpoints(arc a, conic co)`, `intersectionpoints(line l, arc a)` etc...

—
```
arc arcsubtended(point A, point B, real angle)
```

Retourne l'arc capable du segment [`AB`] vu sous un angle `angle`. Bien que le code `un_arcsubtended.C` permette de récupérer le centre de l'arc capable, il est possible de l'obtenir directement en utilisant la routine `point arcsubtendedcenter(poin`
.



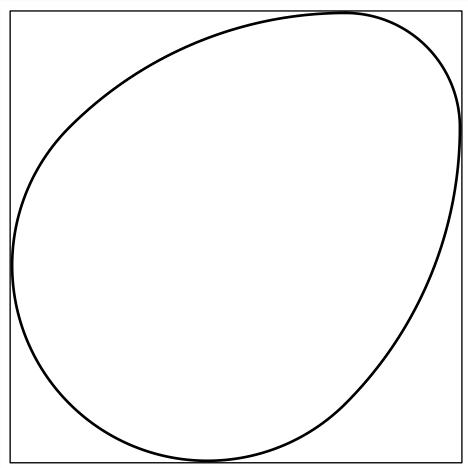

```
import geometry; size(7cm,0);
point A=(-1,0), B=(1,0);
dot("$A$", A, 2W, red); dot("$B$", B, 2E, red);

real[] angles=new real[] {60, 110, -60, -110};
pen[] p=new pen[] {red, blue+red, blue, cyan};
int i=0;

for(real a:angles) {
  arc arcsubtended=arcsubtended(A,B,a);
  draw(arcsubtended, bp+0.8*p[i]);
  for (int j=0; j < 2; ++j) {
    point M=relpoint(arcsubtended, 0.25+0.5*j);
    draw(A--M--B, 0.8*p[i]);
    real gle=degrees(B-M)-degrees(A-M);
    markangle(Label(format("%0g",-gle),UnFill),
              B, M, A, radius=sgn(-gle)*30, Arrow, 0.8*p[i])
  ++i; }
```




```
import geometry; size(7cm,0);
point A=(-1,0), B=(1,0);
dot("$A$", A, S); dot("$B$", B, S);
pen bpp=linewidth(bp);

arc Ac=arcsubtended(A,B,30); draw(Ac, bpp);
dot("$O$", Ac.el.C);
markarc("$\alpha$", Ac, n=2, radius=1cm,
        sectorpen=bpp, markpen=currentpen);

for (int i=0; i < 3; ++i) {
  point M=relpoint(Ac, 0.25+0.25*i);
  draw(M--A^^M--B, linewidth(bp));
  markangle("$\frac{\alpha}{2}$", A, M, B); }
```

— `arc arccircle(point A, point B, real angle, bool direction=CCW)`

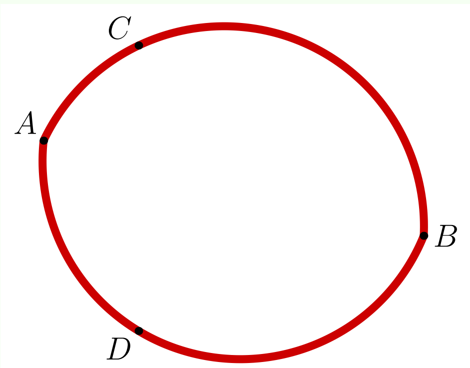
Retourne l'arc de cercle, centré en `A`, depuis `B` jusqu'à l'image de `B` par la rotation de centre `A` et d'angle `angle` parcouru dans le sens `direction`.



```
import geometry;
size(6cm);
point A=(-1,1), B=(1,-1);
point M=(A+B)/2;

point P=rotate(90,M)*B;
arc A1=arccircle(A,B,45), A2=arccircle(B,A,-45,CW),
A3=arccircle(P,relpoint(A2,1),-90,CW),
A4=arccircle(M,A,180);
draw(A1^^A2^^A3^^A4, linewidth(bp));
shipout(bbox());
```

— `arc arccircle(point A, point M, point B)`

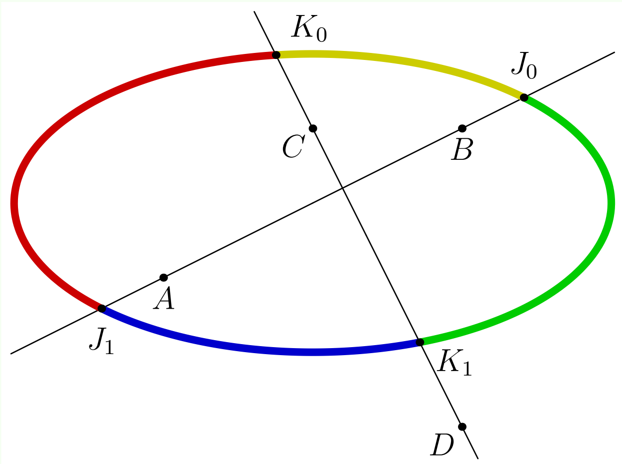
Retourne l'arc de cercle $\widehat{AB}$ passant par `M`.




```
import geometry;
size(6cm);
point A=(-1,0), B=(3,-1), C=(0,1), D=(0,-2);

draw(arccircle(A,C,B), dotsize()+0.8*red);
draw(arccircle(A,D,B), dotsize()+0.8*red);

dot("$A$", A, NW); dot("$B$", B, E);
dot("$C$", C, NW); dot("$D$", D, SW);
```

— `arc arc(ellipse el, point M, point N, bool direction=CCW)`

Retourne l'arc de `el`, parcouru dans le sens `direction`, d'extrémités `M` et `N` qui doivent être des points appartenant à `el`.



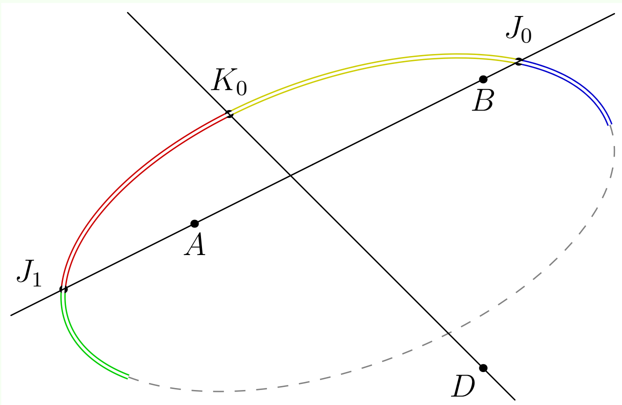

```
import geometry; size(8cm);
point A=(-1,0), B=(1,1), C=(0,1), D=(1,-1);
dot("$A$",A,S); dot("$B$",B,S); dot("$C$",C,SW); dot("$D$",D,SW);
ellipse el=ellipse((point)(0,0.5),2,1);
line l1=line(A,B), l2=line(C,D); draw(l1); draw(l2);
point[] J=intersectionpoints(l1,el), K=intersectionpoints(l2,el);
draw(arc(el, J[0],K[0]), 1mm+0.8yellow); draw(arc(el, K[0],J[1]), 1mm+0.8red);
draw(arc(el, J[1],K[1]), 1mm+0.8blue); draw(arc(el, K[1],J[0]), 1mm+0.8green);
dot("$J_0$", J[0], 2N); dot("$J_1$", J[1], 2S);
dot("$K_0$", K[0], 2NE) ; dot("$K_1$", K[1], 2dir(-35));
```

— `arc arc(ellipse el, explicit abscissa x1, explicit abscissa x2, bool direction=CCW)`

Cette routine a le même comportement que la routine précédente mais les points sont spécifiés à l'aide d'abscisses par rapport à l'ellipse.

— `arc arc(explicit arc a, point M, point N)`

Retourne la partie de l'arc a comprise entre M et N.




```
import geometry; size(8cm);
point A=(-1,0), B=(1,1), C=(0,0), D=(1,-1);
dot("$A$",A,S); dot("$B$",B,S); dot("$D$",D,SW);
arc c=arc(ellipse(C,2,1,20), 0, 270); draw(complementary(c),dashed+grey);
line l1=line(A,B), l2=line(C,D);
point[] J=intersectionpoints(l1,c), K=intersectionpoints(l2,c);
draw(arc(c,J[0],K[0]), 2bp+0.8yellow); draw(arc(c,K[0],J[1]), 2bp+0.8red);
draw(arc(c,J[1],relpoint(c,1)), 2bp+0.8green); draw(arc(c,point(c,0),J[0]), 2bp+0.8blue);
dot("$J_0$",J[0],2N); dot("$J_1$",J[1],N+2W); dot("$K_0$",K[0],2N);
draw(c, bp+white); draw(l1^^l2);
```

— `arc arc(arc el, explicit abscissa x1, explicit abscissa x2)`

Cette routine a le même comportement que la routine précédente mais les points sont spécifiés à l'aide d'abscisses par rapport à l'ellipse.

— `arc inverse(real k, point A, segment s)`

Retourne l'image de `s` par l'inversion de pôle `A` et de puissance `k`; voir l'illustration de `inversion*segment` dans la section Inversions.

— `line tangent(explicit arc a, point M)`

Retourne la tangente à `a` au point `M` de `a`.

— `line tangent(explicit arc a, abscissa x)`

Retourne la tangente à `a` au point d'abscisse `x` donné par rapport à `a`.

10. Abscisses

Le type `abscissa` permet d'instancier une abscisse sur un objet de type `line`, `segment`, `conic` et `arc`. La structure d'un objet de type `abscissa` est la suivante :

```
struct abscissa {
  real x; int system; polarconicroutine
  polarconicroutine;
  abscissa copy() {...}
}
```

`x` est la valeur de l'abscisse.

`system` représente le type d'abscisse :

0 pour une abscisse comme fraction de la longueur d'un chemin ;

1 pour une abscisse curviligne ;

2 pour une abscisse angulaire ;

3 pour une abscisse relative aux nœds du chemin.

Pour une meilleure lisibilité du code, les constantes suivantes sont prédéfinies : `int relativesystem=0, curvilinearsys`

`polarconicroutine` permet de spécifier le centre de référence dans le cas d'une abscisse angulaire ; les valeurs possibles sont `fromCenter` et `fromFocus` ;

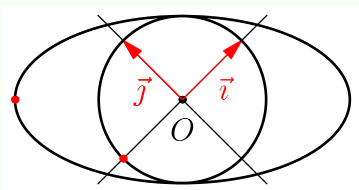
`abscissa copy()` retourne la copie de l'abscisse.

10.1. Définir une abscisse

Il y a autant de routines pour définir une abscisse que de types d'abscisses. Une fois une abscisse définie, on peut récupérer le point d'un objet à cette abscisse par la routine `point(objet,abscisse)`.

— `abscissa relabscissa(real x)`

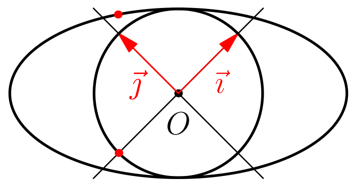
Retourne l'abscisse `x` comme fraction de la longueur d'un chemin. On notera que le code `point(objet,relabscissa(x))` est équivalent à `relpoint(objet,x)`.




```
import geometry; size(4.5cm);
currentcoordsys=rotate(45)*defaultcoordsys;
show(currentcoordsys);
abscissa rel=relabscissa(0.5);
ellipse el=ellipse(origin(),2,1,-45); draw(el,linewidth(bp));
circle c=circle(origin(),1);          draw(c,linewidth(bp));
dot(point(el,rel), red); dot(point(c,rel), red);
```

— `abscissa curabscissa(real x)`

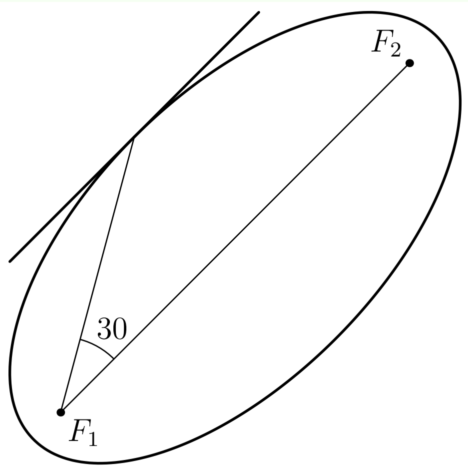
Retourne l'abscisse curviligne `x`. On notera que le code `point(objet,curabscissa(x))` est équivalent à `curpoint(objet,x)`.




```
import geometry; size(4.5cm);
currentcoordsys=rotate(45)*defaultcoordsys;
show(currentcoordsys);
abscissa cur=curabscissa(pi);
ellipse el=ellipse(origin(),2,1,-45); draw(el,linewidth(bp));
circle c=circle(origin(),1);          draw(c,linewidth(bp));
dot(point(el,cur), red); dot(point(c,cur), red);
```

— `abscissa angabscissa(real x, polarconicroutine polarconicroutine=currentpolarconicroutine)`

Retourne l'abscisse angulaire `x`. On notera que le code `point(objet,angabscissa(x))` est équivalent à `angpoint(objet,x)`.




```
import geometry;
size(6cm);
abscissa x=angabscissa(30);

ellipse el=ellipse(origin(),2,1,45);
draw(el,linewidth(bp));

point M=point(el,x);
draw(M--el.F1--el.F2);
dot("$F_1$", el.F1, SE); dot("$F_2$", el.F2, NW);
markangle((string)x.x, el.F2, el.F1, M);
draw(tangent(el,x), linewidth(bp));
```

— `abscissa nodabscissa(real x)`

Retourne l'abscisse `x` relative aux nœds d'un chemin. On notera que le code `point(objet,nodabscissa(x))` est équivalent à `point(objet,x)`.

10.2. Récupérer une abscisse d'un point

Les routines permettant de récupérer une abscisse d'un point appartenant à un objet donné portent le même nom que celles décrites dans le section précédente. Les routines suivantes renvoient respectivement l'abscisse relative, l'abscisse curviligne, l'abscisse angulaire et « l'abscisse par nœud » du point `M` appartenant à l'objet spécifié.

Abscisse relative

— `abscissa relabscissa(line l, point M)`

— `abscissa relabscissa(ellipse el, point M)`

— `abscissa relabscissa(arc a, point M)`

Abscisse curviligne

— `abscissa curabscissa(line l, point M)`

— `abscissa curabscissa(ellipse el, point M)`

— `abscissa curabscissa(parabola p, point M)`

Abscisse angulaire

— `abscissa angabscissa(circle c, point M)`

— `abscissa angabscissa(ellipse el, point M, polarconicroutine polarconicroutine=currentpolarconicroutine)`

— `abscissa angabscissa(hyperbola h, point M, polarconicroutine polarconicroutine=currentpolarconicroutine)`

— `abscissa angabscissa(parabola p, point M)`

« Abscisse par nœud »

— `abscissa nodabscissa(line l, point M)`

— `abscissa nodabscissa(ellipse el, point M)`

— `abscissa nodabscissa(parabola p, point M)`

10.3. Opérateurs

`abscissa operator +(real x, explicit abscissa a)`

Retourne la copie de `a` d'abscisse `x+a.x`.

Les opérateurs suivants sont aussi définis :

```
operator +(explicit abscissa,real)
operator -(real,explicit abscissa)
operator -(explicit abscissa,real)
operator -(explicit abscissa a)
operator *(real x, explicit abscissa a)
operator *(explicit abscissa a, real x)
operator /(real x, explicit abscissa a)
operator /(explicit abscissa a, real x).
```

11. Triangles

11.1. La structure

La structure du type `triangle` est un peu plus complexe que celles déjà rencontrées dans ce document car elle définit de nouveaux types instanciant des objets indissociables d'un triangle et qui possèdent eux-mêmes la référence du triangle auquel ils sont associés. Autrement dit, pour fixer les idées, un objet `TR` de type triangle possède l'objet `VA` de type `vertex`, accessible par `TR.VA`, qui contient à son tour l'objet `t` de type `triangle` dont la valeur est justement `TR` ; ainsi `TR.VA.t` vaut `TR`.

Sachant qu'un objet de type `vertex` représente un sommet d'un triangle, il est alors aisé de définir une routine renvoyant, par exemple, la première bissectrice d'un triangle passant par un sommet donné avec comme seul paramètre un objet de type `vertex`, puisque celui-ci contient la référence du triangle dont il est le sommet.

Voici une version simplifiée de la structure du type `triangle` ; la structure complète est détaillée séparément.

```
struct triangle {
  restricted point A, B, C;

  struct vertex {
    int n;
    triangle t; }

  restricted vertex VA, VB, VC;

  struct side {
    int n;
    triangle t; }

  side AB, BC, CA, BA, AC, CB; }
```

`A`, `B` et `C` représentent les points marquant les sommets du triangle ;

`struct vertex` définit la structure `vertex` qui permet d'instancier un objet représentant le sommet d'un triangle. Bien que cette structure ne soit pas destinée à une utilisation classique de l'extension *geometry.asy* il peut être utile d'en connaître les propriétés.

La propriété `n` permet d'associer un sommet au point, de type `point`, marquant ce sommet :
si `n` = 1, le sommet est associé au point `A` ;

si `n` $= 2$, le sommet est associé au point `B` ;
si `n` $= 3$, le sommet est associé au point `C` ;
si `n` $= 4$, le sommet est associé au point `A` ;
etc...
La propriété `t` a pour valeur « l'objet, de type triangle, auquel appartient le sommet ».
L'utilisation de cette structure est détaillée dans la section Sommets de triangles.

`VA, VB et VC` représentent abstraitement les sommets du triangle, par opposition aux objets `point A`, `B` et `C` qui sont concrètement les points marquant le sommet ; voir la section Sommets de triangles.

`struct side` définit la structure `side` qui permet d'instancier un objet représentant le côté d'un triangle. Bien que cette structure ne soit pas destinée à une utilisation classique de l'extension *geometry.asy* il peut être utile d'en connaître les propriétés.

La propriété `n` permet d'associer l'objet de type `side` au côté **orienté** du triangle :
si `n` $= 1$, le côté représente `AB` orienté de `A` vers `B` ;
si `n` $= 2$, le côté représente `BC` orienté de `B` vers `C` ;
si `n` $= 3$, le côté représente `CA` orienté de `C` vers `A` ;
si `n` $= 4$, le côté représente `AB` ;
etc...
Si `n` est négatif l'orientation est inversée.
La propriété `t` a pour valeur « l'objet, de type triangle, auquel appartient le sommet ».
L'utilisation de cette structure est détaillée dans la section Côtés de triangles.

`AB, BC, CA, BA, AC et CB` représente abstraitement les côtés du triangle, par opposition aux objets `line line(TR.AB)`, `line(TR.BC)`, `line(TR.CA)`, etc.. qui sont concrètement les droites marquant les côtés du triangle `TR` ; voir la section Côtés de triangles.

11.2. Définir et tracer un triangle

Dans cette section ne seront décrites que les routines de base pour définir et tracer un triangle. D'autres routines retournant un triangle seront introduites au fur et à mesure.

—
```
void label(picture pic=currentpicture, Label LA="$A$",
Label LB="$B$", Label LC="$C$",
triangle t,
real alignAngle=0,
real alignFactor=1,
pen p=nullpen, filltype filltype=NoFill)
```

Place les labels `LA`, `LB` et `LC` aux sommets du triangle `t`, alignés suivant la première bissectrice du sommet correspondant. Les paramètres `alignAngle` et `alignFactor` permettent de modifier la direction et la longueur de l'alignement.

—
```
void show(picture pic=currentpicture,
Label LA="$A$", Label LB="$B$", Label LC="$C$",
Label La="$a$", Label Lb="$b$", Label Lc="$c$",
triangle t, pen p=currentpen, filltype filltype=NoFill)
```

Trace le triangle `t` et affiche les labels aux sommets du triangle ainsi que les longueurs de ses côtés. Cette routine est surtout utile pour localiser les sommets `t.A`, `t.B` et `t.C` en cours de codage.

—
```
void draw(picture pic=currentpicture, triangle t,
pen p=currentpen, marker marker=nomarker)
```

Trace le triangle `t` ; les côtés sont tracés comme des segments.

—
```
void drawline(picture pic=currentpicture, triangle t, pen p=currentpen)
```

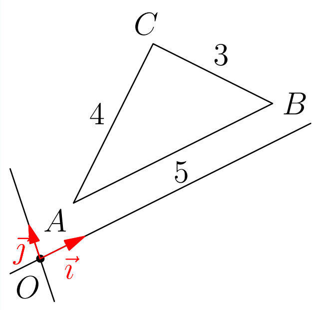
Trace le triangle `t` ; les côtés sont tracés comme des droites.

—
```
triangle triangle(point A, point B, point C)
```

Renvoie le triangle dont les sommets sont `A`, `B` et `C`.

—
```
triangle triangleabc(real a, real b, real c, real angle=0, point A=(0,0))
```

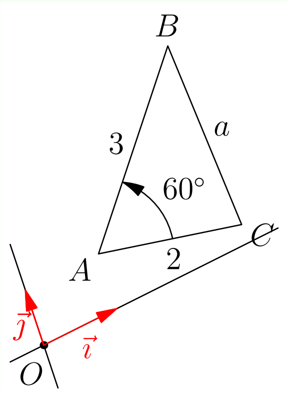
Retourne le triangle `ABC` tel que $\mathtt{BC} = \mathtt{a}$, $\mathtt{AC} = \mathtt{b}$, $\mathtt{AB} = \mathtt{c}$ et $\left(\overrightarrow{\imath} ; \overrightarrow{\mathtt{AB}}\right) = \mathtt{angle}$.




```
import geometry;
size(4cm);

currentcoordsys=cartesiansystem(i=(1,0.5), j=(-0.25,.75));
show(currentcoordsys);

triangle t=triangleabc(3,4,5, (1,1));
show(La="3", Lb="4", Lc="5", t);
```

— `triangle triangleAbc(real alpha, real b, real c, real angle=0, point A=(0,0))`

Retourne le triangle `ABC` tel que $\left(\overrightarrow{\mathtt{AB}}\,;\,\overrightarrow{\mathtt{AC}}\right)=\mathtt{alpha}$, $\mathtt{AC}=\mathtt{b}$, $\mathtt{AB}=\mathtt{c}$ et $\left(\vec{\imath}\,;\,\overrightarrow{\mathtt{AC}}\right)=\mathtt{angle}$.




```
import geometry;

size(5cm);

currentcoordsys=cartesiansystem(i=(1,0.5),j=(-0.25,.75));
show(currentcoordsys);

triangle t=triangleAbc(-60,2,3,angle=45,(1,1));
show(Lb="2", Lc="3",t);
markangle("$60^\circ$",t.C,t.A,t.B, Arrow);
```

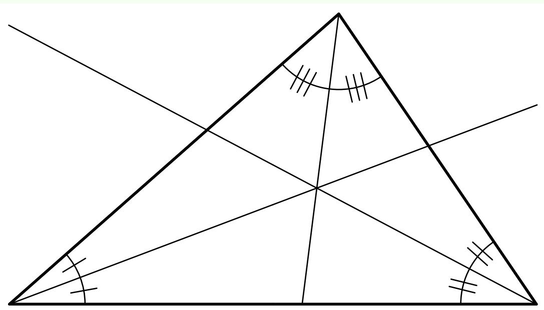
— `triangle triangle(line l1, line l2, line l3)`

Renvoie le triangle dont les côtés sont `l1`, `l2` et `l3`.

11.3. Sommets de triangles

Étant donné un objet `t` de type `triangle`, ses propriétés `t.VA`, `t.VB` et `t.VC`, de type `vertex`, représentent les sommets du triangle `t`. L'extension *geometry.asy* implémente ainsi des routines admettant comme paramètre un sommet de triangle sans avoir à spécifier explicitement le triangle auquel il se réfère.

Par exemple, dans le code suivant, la routine `line bisector(vertex V, real angle=0)` retourne l'image par la rotation d'angle `angle` et de centre `V` de la première bissectrice passant par `V`.



```
size(7cm); import geometry;
triangle t=triangleabc(4,5,6);
drawline(t, linewidth(bp));
line ba=bisector(t.VA), bb=bisector(t.VB);
line bc=bisector(t.VC); draw(ba^^bb^^bc);
markangle((line) t.AB, (line) t.AC, StickIntervalMarker(2,1));
markangle((line) t.BC, (line) t.BA, StickIntervalMarker(2,2));
markangle((line) t.CA, (line) t.CB, StickIntervalMarker(2,3));
```

Voici quelques routines et opérateurs élémentaires relatifs aux objets de type `vertex` :

— `point operator cast(vertex V)`

Permet le « casting » d'un objet de type `vertex` en objet de type `point`.

— `point point(explicit vertex V)`

Renvoie l'objet de type `point` représenté par l'objet `V` de type `vertex`. Le code `point(V)` est équivalent au code `(point)V` qui force le « casting » de `vertex` vers `point`.

— `vector dir(vertex V)`

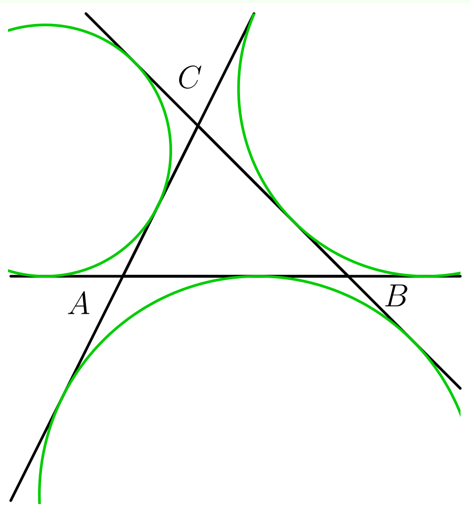
Renvoie le vecteur unitaire porté par la première bissectrice de l'angle en `V` et orienté vers l'extérieur du triangle auquel se réfère `V`. Cette routine est particulièrement utile pour placer des labels aux sommets d'un triangle.

D'autres routines admettant comme paramètre un objet de type `vertex` sont décrites dans la section suivante et conjointement aux routines concernant les triangles.

11.4. Côtés de triangles

Étant donné un objet `t` de type `triangle`, ses propriétés `t.AB`, `t.BC`, `t.CA`, `t.BA`, `t.AC` et `t.CB`, de type `side`, représentent les côtés du triangle `t`. L'extension *geometry.asy* implémente ainsi des routines admettant comme paramètre un côté de triangle sans avoir à spécifier explicitement le triangle auquel le côté se réfère.

Par exemple, dans le code suivant, la routine `circle excircle(side s)` retourne le cercle exinscrit du triangle auquel se réfère `s` et tangent à `s`.




```
import geometry;
size(6cm,0);

triangle t=triangle((-1,0), (2,0), (0,2));

drawline(t, linewidth(bp));
label(t,alignFactor=4);

clipdraw(excircle(t.AB), bp+0.8green);
clipdraw(excircle(t.BC), bp+0.8green);
clipdraw(excircle(t.AC), bp+0.8green);

draw(box((-2.5,-3), (3.5,3.5)), invisible);
```

Voici quelques routines et opérateurs élémentaires relatifs aux objets de type `side` :

— `line operator cast(side side)`

Permet le « casting » d'un objet de type `side` en objet de type `line`.

— `line line(explicit side side)`

Renvoie l'objet de type `line` représenté par l'objet `side` de type `side`. Le code `line(S)` est équivalent au code `(line)S` qui force le « casting » de `side` vers `line`.

— `segment segment(explicit side side)`

Renvoie l'objet de type `segment` représenté par l'objet `side` de type `side`. Le code `segment(S)` est équivalent au code `(segment)S` qui force le « casting » de `side` vers `segment`.

— `side opposite(vertex V)`

Renvoie le côté opposé à `V` dans le triangle auquel se réfère `V`.

— `vertex opposite(side side)`

Renvoie le sommet opposé à `side` dans le triangle auquel se réfère `side`.

Les autres routines admettant comme paramètre un objet de type `side` sont décrites conjointement aux routines concernant les triangles.

11.5. Opérateurs

Le seul opérateur s'appliquant aux objets de type `triangle` est `triangle operator *(transform T, triangle t)` qui autorise le code `transform*triangle`.

11.6. Autres routines

— `point orthocentercenter(triangle t)`

Retourne l'orthocentre du triangle `t`.

— `point foot(vertex V)`

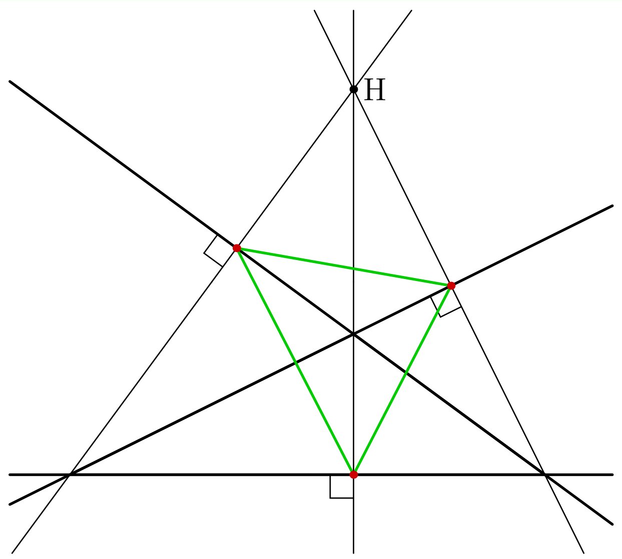
Retourne le pied de la hauteur issue de `V`. La routine `point foot(side side)` est aussi disponible.

— `line altitude(vertex V)`

Retourne la hauteur issue de `V`. La routine `line altitude(side side)` est aussi disponible.

— `triangle orthic(triangle t)`

Retourne le triangle orthique de `t` ; les sommets sont les pieds des hauteurs de `t`.




```
size(8cm);
import geometry;

triangle t=triangleabc(3,4,6);
drawline(t, linewidth(bp));
line hc=altitude(t.AB), hb=altitude(t.AC);
line ha=altitude(t.BC); draw(hc^^hb^^ha);
dot("H", orthocentercenter(t));

perpendicularmark(t.AB,hc,quarter=-1);
perpendicularmark(t.AC,hb,quarter=-1);
perpendicularmark(t.BC,ha);

triangle ort=orthic(t);
draw(ort,bp+0.8*green); dot(ort, 0.8*red);
addMargins(1cm,1cm);
```

— `point midpoint(side side)`

Retourne le milieu de `side`.

— `point centroid(triangle t)`

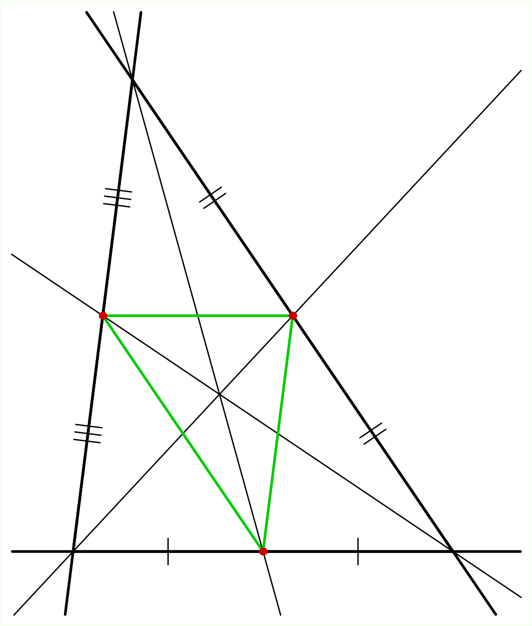
Retourne le centre de gravité du triangle `t`.

— `line median(vertex V)`

Retourne la médiane issue de `V`. La routine `line median(side side)` est aussi disponible.

— `triangle medial(triangle t)`

Retourne le triangle dont les sommets sont les milieux de `t`.



```
size(8cm);
import geometry;

triangle t=triangleabc(6,5,4);
drawline(t, linewidth(bp));
line ma=median(t.VA), mb=median(t.VB);
line mc=median(t.VC); draw(ma^^mb^^mc);

draw(segment(t.AB), StickIntervalMarker(2,1));
draw(segment(t.BC), StickIntervalMarker(2,2));
draw(segment(t.CA), StickIntervalMarker(2,3));

triangle med=medial(t);
draw(med,bp+0.8*green); dot(med, 0.8*red);
addMargins(1cm,1cm);
```

— `triangle anticomplementary(triangle t)`

Retourne le triangle dont les milieux des côtés sont les sommets de `t`.

— `line bisector(vertex V, real angle=0)`

Retourne l'image par la rotation d'angle `angle` et de centre `V` de la première bissectrice passant par `V`. Un exemple est donné ci-avant.

— `point bisectorpoint(side side)`

Retourne le point d'intersection de la première bissectrice de l'angle opposé à `side` avec `side`.

— `line bisector(side side)`

Retourne la médiatrice du segment représenté par `side`.

— `point circumcenter(triangle t)`

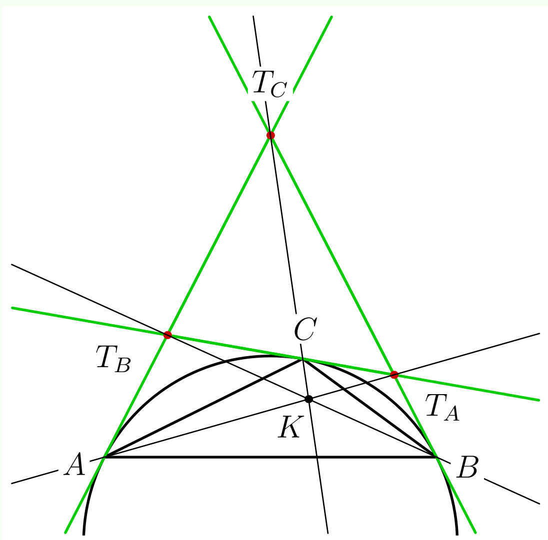
Retourne le centre du cercle circonscrit au triangle `t`.

— `circle circle(triangle t)`

Retourne le cercle circonscrit au triangle `t`. La routine `circumcircle(triangle t)` en est un alias.

— `triangle tangential(triangle t)`

Retourne le triangle dont les côtés sont tangents au cercle circonscrit à `t` et passent par ses sommets.




```
size(7cm); import geometry;
triangle t=triangleabc(3,4,6);
draw(t, linewidth(bp));
clipdraw(circle(t), linewidth(bp));
triangle itr=tangential(t);
drawline(itr, bp+0.8*green); dot(itr, 0.8*red);
line syma=line(itr.A,t.A), symb=line(itr.B,t.B);
line symc=line(itr.C,t.C); draw(syma^^symb^^symc);
dot("$K$", intersectionpoint(syma,symb),
    2*dir(-120));
label(t,alignFactor=2,UnFill);
label("$T_A$","$T_B$","$T_C$", itr, alignFactor=4,
      UnFill);
addMargins(1cm,1cm);
```

— `point incenter(triangle t)`

Retourne le centre du cercle inscrit dans le triangle `t`.

— `real inradius(triangle t)`

Retourne le rayon du cercle inscrit dans le triangle `t`.

— `circle incircle(triangle t)`

Retourne le cercle inscrit dans le triangle `t`.

— `triangle intouch(triangle t)`

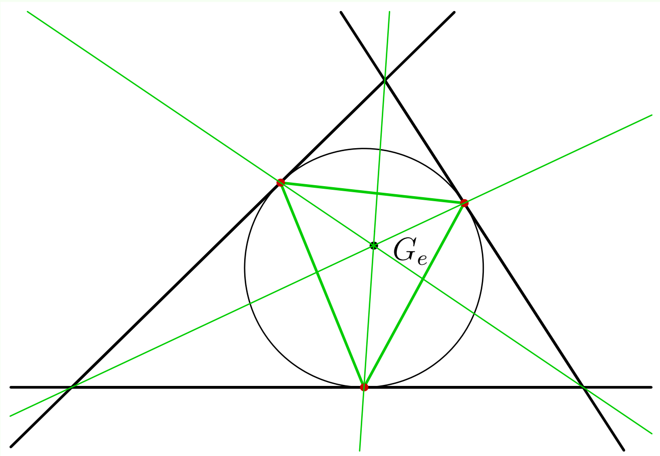
Retourne le triangle dont les sommets sont les points de contact du cercle inscrit à `t` avec `t`.

— `point intouch(side side)`

Retourne le point de contact du côté `side` avec le cercle inscrit au triangle auquel se réfère `side`.

— `point gergonne(triangle t)`

Renvoie le point de GERGONNE du triangle `t`.




```
size(8.5cm,0);import geometry;
triangle t=triangleabc(5,6,7);
drawline(t, linewidth(bp));
draw(incircle(t));
triangle itr=intouch(t);
draw(itr,bp+0.8*green); dot(itr, 0.8*red);
point Ge=gergonne(t);
dot("$G_e$", Ge, 2*dir(-10));
draw(line(Ge,t.A), 0.8*green);
draw(line(Ge,t.B), 0.8*green);
draw(line(Ge,t.C), 0.8*green);
addMargins(1cm,1cm);
```

— `point excenter(side side)`

Retourne le centre du cercle exinscrit du triangle auquel se réfère `side` et tangent à `side`.

— `real exradius(side side)`

Retourne le rayon du cercle exinscrit du triangle auquel se réfère `side` et tangent à `side`.

— `circle excircle(side side)`

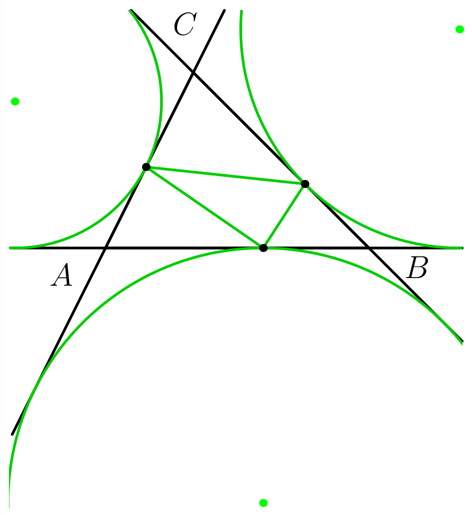
Retourne le cercle exinscrit du triangle auquel se réfère `side` et tangent à `side`.

— `triangle extouch(triangle t)`

Retourne le triangle dont les sommets sont les points de contact des cercles exinscrit à `t` avec ses côtés.

— `point extouch(side side)`

Retourne le point de contact du côté `side` avec le cercle exinscrit renvoyé par `excircle(side)`.




```
import geometry; size(6cm,0);

triangle t=triangle((-1,0), (2,0), (0,2));
drawline(t, linewidth(bp));
label(t,alignFactor=4);

circle c1=excircle(t.AB), c2=excircle(t.BC);
circle c3=excircle(t.AC);
clipdraw(c1, bp+0.8green);
clipdraw(c2, bp+0.8green);
clipdraw(c3, bp+0.8green);
dot(c1.C^^c2.C^^c3.C, green);
draw(extouch(t), bp+0.8green, dot);
```

— `point symmedian(triangle t)`

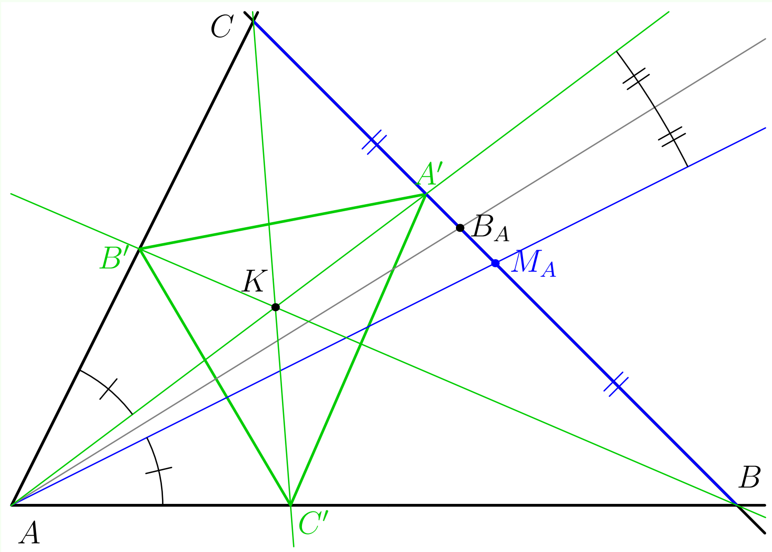
Retourne le point symédian (ou point de Lemoine) du triangle `t`.

— `point symmedian(side side)`

Retourne le point symédian du côté `side`.

— `line symmedian(vertex V)`

Retourne la droite symédiane passant par `V` du triangle auquel se réfère `V`.




```
import geometry; size(10cm,0);
triangle t=triangle((-1,0), (2,0), (0,2)); drawline(t, linewidth(bp));
label(t,alignFactor=2, alignAngle=90);
triangle st=symmedial(t); draw(st, bp+0.8green);
label("$A'$", "$B'$", "$C'$", st, alignAngle=45, 0.8green);
line mA=median(t.VA); draw(mA, blue); dot("$M_A$",midpoint(t.BC), 1.5E, blue);
draw(segment(t.BC), bp+blue, StickIntervalMarker(2,2,blue));
line bA=bisector(t.VA); draw(bA, grey); dot("$B_A$", bisectorpoint(t.BC));
line sA=symmedian(t.VA); draw(sA, 0.8*green);
draw(symmedian(t.VB), 0.8*green); draw(symmedian(t.VC), 0.8*green);
point sP=symmedian(t); dot("$K$", sP, 2*dir(125));
markangle(sA, (line) t.AC, radius=2cm, StickIntervalMarker(1,1));
markangle((line) t.AB, mA, radius=2cm, StickIntervalMarker(1,1));
markangle(mA, sA, radius=10cm, StickIntervalMarker(2,2));
```

— `point cevian(side side, point P)`

Renvoie le point de Cevian de P appartenant au côté side.

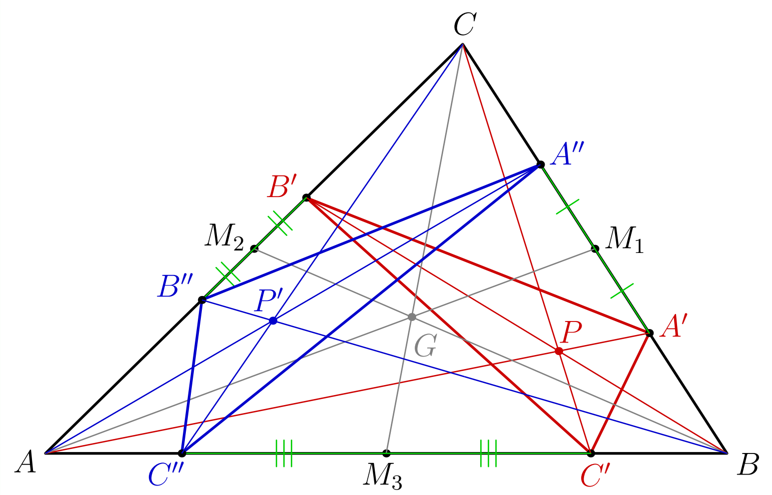
— `triangle cevian(triangle t, point P)`

Renvoie le triangle de Cevian relatif à P.

— `line cevian(vertex V, point P)`

Renvoie la droite de Cevian relative à P, passant par V dans le triangle auquel se réfère V.

L'exemple suivant illustre la propriété « *si un triangle $A'B'C'$ est un triangle de Cevian d'un triangle ABC alors le triangle $A''B''C''$, dont les sommets sont les symétriques de A', B' et C' par rapport aux milieux des côtés respectifs, est aussi un triangle de Cevian* »




```
import geometry; size(10cm,0);
triangle t=triangleabc(5,6,7); label(t); draw(t, linewidth(bp));
point P=0.6*t.B+0.25*t.C; dot("$P$", P, dir(60), 0.8*red);
triangle C1=cevian(t, P);
label("$A'$","$B'$","$C'$", C1, 0.8*red); draw(C1, bp+0.8*red, dot);
draw(t.A--C1.A, 0.8*red); draw(t.B--C1.B, 0.8*red); draw(t.C--C1.C, 0.8*red);

point Ma=midpoint(t.BC), Mb=midpoint(t.AC), Mc=midpoint(t.BA);
dot("$M_1$",Ma,-dir(t.VA)); dot("$M_2$",Mb,-dir(t.VB)); dot("$M_3$",Mc,-dir(t.VC));
draw(t.A--Ma^^t.B--Mb^^t.C--Mc, grey); dot("$G$", centroid(t), 2*dir(-65), grey);

point App=rotate(180,Ma)*C1.A, Bpp=rotate(180,Mb)*C1.B, Cpp=rotate(180,Mc)*C1.C;
draw(C1.A--App, 0.8*green, StickIntervalMarker(2,1,0.8*green));
draw(C1.B--Bpp, 0.8*green, StickIntervalMarker(2,2,0.8*green));
draw(C1.C--Cpp, 0.8*green, StickIntervalMarker(2,3,0.8*green));

triangle C2=triangle(App,Bpp,Cpp);
label("$A''$","$B''$","$C''$", C2, 0.8*blue); draw(C2, bp+0.8*blue, dot);
segment sA=segment(t.A,C2.A), sB=segment(t.B,C2.B);
point PP=intersectionpoint(sA,sB);
dot("$P'$", PP, dir(100), 0.8*blue);
draw(sA, 0.8*blue); draw(sB, 0.8*blue); draw(segment(t.C,C2.C), 0.8*blue);
```

— `line isotomic(vertex V, point M)`

Renvoie la droite isotomique passant par V et relative à M dans le triangle auquel se réfère V.

— `point isotomicconjugate(triangle t, point M)`

Renvoie le conjugué isotomique de M relativement à t.

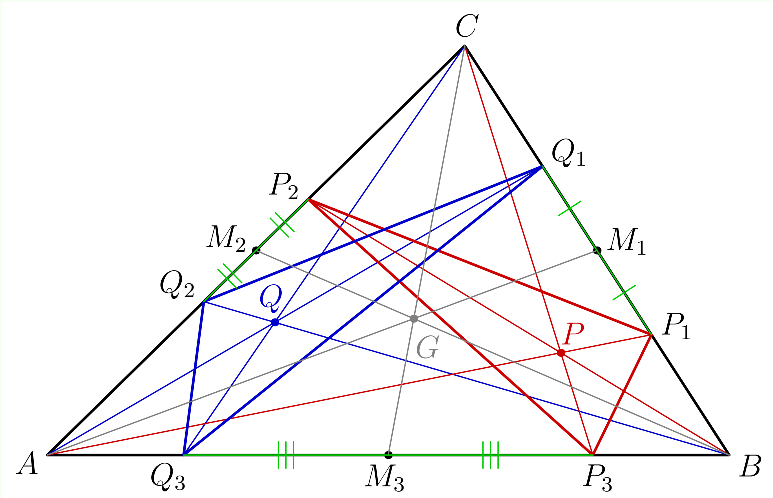
— `point isotomic(side side, point M)`

Renvoie le point d'intersection de la droite isotomique de M, relativement au triangle auquel se réfère side, avec le côté side.

— `triangle isotomic(triangle t, point M)`

Renvoie le triangle dont les sommets sont les points d'intersection des droites isotomiques relatives à `M` dans `t` avec les côtés de `t`. Ainsi, dans la figure précédente, le triangle $A''B''C''$ est le triangle isotomique relatif à `P`.

Ci-dessous, la même figure obtenue à l'aide des routines `isotomic` gagne en concision.




```
import geometry; size(10cm,0);
triangle t=triangleabc(5,6,7); label(t); draw(t, linewidth(bp));
point P=0.6*t.B+0.25*t.C; dot("$P$", P, dir(60), 0.8*red);
draw(segment(isotomic(t.VA,P))^^segment(isotomic(t.VB,P))^^segment(isotomic(t.VC,P)),
     0.8*blue);
draw(segment(cevian(t.VA,P))^^segment(cevian(t.VB,P))^^segment(cevian(t.VC,P)),
     0.8*red);
triangle t1=cevian(t,P); label("$P_1$", "$P_2$", "$P_3$", t1); draw(t1, bp+0.8*red);
triangle t2=isotomic(t,P); label("$Q_1$", "$Q_2$", "$Q_3$", t2); draw(t2, bp+0.8*blue);
dot("$Q$", isotomicconjugate(t,P), dir(100), 0.8*blue);

point Ma=midpoint(t.BC), Mb=midpoint(t.AC), Mc=midpoint(t.BA);
dot("$M_1$",Ma,-dir(t.VA)); dot("$M_2$",Mb,-dir(t.VB)); dot("$M_3$",Mc,-dir(t.VC));
draw(t.A--Ma^^t.B--Mb^^t.C--Mc, grey); dot("$G$", centroid(t), 2*dir(-65), grey);
draw(t1.A--t2.A, 0.8*green, StickIntervalMarker(2,1,0.8*green));
draw(t1.B--t2.B, 0.8*green, StickIntervalMarker(2,2,0.8*green));
draw(t1.C--t2.C, 0.8*green, StickIntervalMarker(2,3,0.8*green));
```

— `point isogonalconjugate(triangle t, point M)`

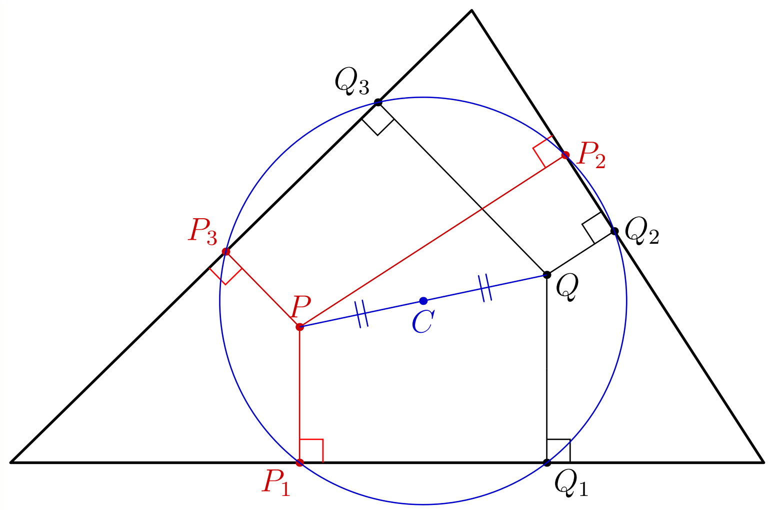
Renvoie le conjugué isogonal de `M` relativement à `t`.

— `point isogonal(side side, point M)`

Renvoie le point d'intersection de la droite isogonale de `M`, relativement au triangle auquel se réfère `side`, avec le côté `side`.

— `triangle isogonal(triangle t, point M)`

Renvoie le triangle dont les sommets sont les points d'intersection avec les côtés de `t` des droites isogonales relatives à `M` dans `t`.
L'exemple suivant illustre la propriété « les triangles podaires de deux points isogonaux P et Q sont inscrits dans un même cercle de centre le milieu de $[PQ]$ ».




```
import geometry; size(10cm,0);
triangle t=triangleabc(5,6,7); draw(t, linewidth(bp));
point P=0.5*t.B+0.3*(t.C-t.B); dot("$P$", P, N, 0.8*red);

point Q=isogonalconjugate(t,P); dot("$Q$", Q, dir(-30));
point Q1=projection(t.AB)*Q; segment sq1=segment(Q,Q1);
point Q2=projection(t.BC)*Q; segment sq2=segment(Q,Q2);
point Q3=projection(t.AC)*Q; segment sq3=segment(Q,Q3);
draw(sq1); draw(sq2); draw(sq3);
dot("$Q_1$", Q1, SE); dot("$Q_2$", Q2); dot("$Q_3$", Q3, NW);

point P1=projection(t.AB)*P; segment sp1=segment(P,P1);
point P2=projection(t.BC)*P; segment sp2=segment(P,P2);
point P3=projection(t.AC)*P; segment sp3=segment(P,P3);
draw(sp1, 0.8*red); draw(sp2, 0.8*red); draw(sp3, 0.8*red);
dot("$P_1$",P1,SW,0.8*red); dot("$P_2$",P2,0.8*red); dot("$P_3$",P3,NW,0.8*red);

perpendicularmark(t.AB,sq1); perpendicularmark(t.BC,sq2);
perpendicularmark(reverse(t.AC),sq3); perpendicularmark(t.AB,sp1, red);
perpendicularmark(t.BC,sp2, red); perpendicularmark(reverse(t.AC),sp3, red);

circle C=circle(Q1,Q2,Q3); draw(C, 0.8*blue);
draw(segment(Q,P), 0.8*blue, StickIntervalMarker(2,2, 0.8*blue));
dot("$C$", C.C, S, 0.8*blue);
```

— `point[] fermat(triangle t)`

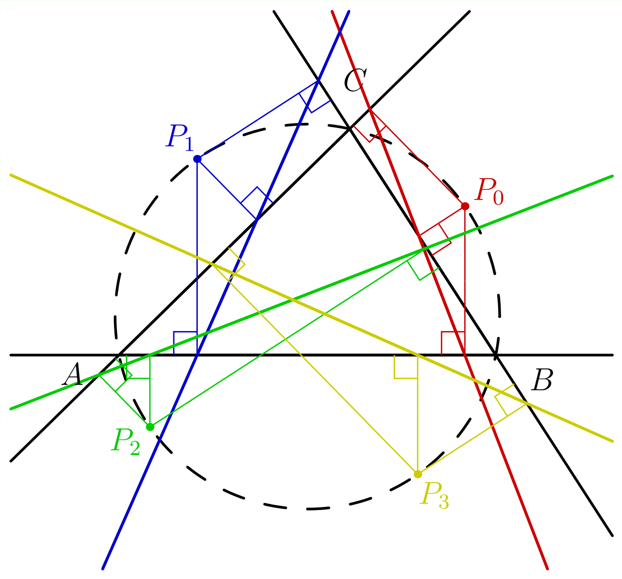
Renvoie les points de Fermat du triangle `t`.

— `triangle pedal(triangle t, point M)`

Renvoie le triangle podaire par rapport à `M` dans `t`.

— `line pedal(side side, point M)`

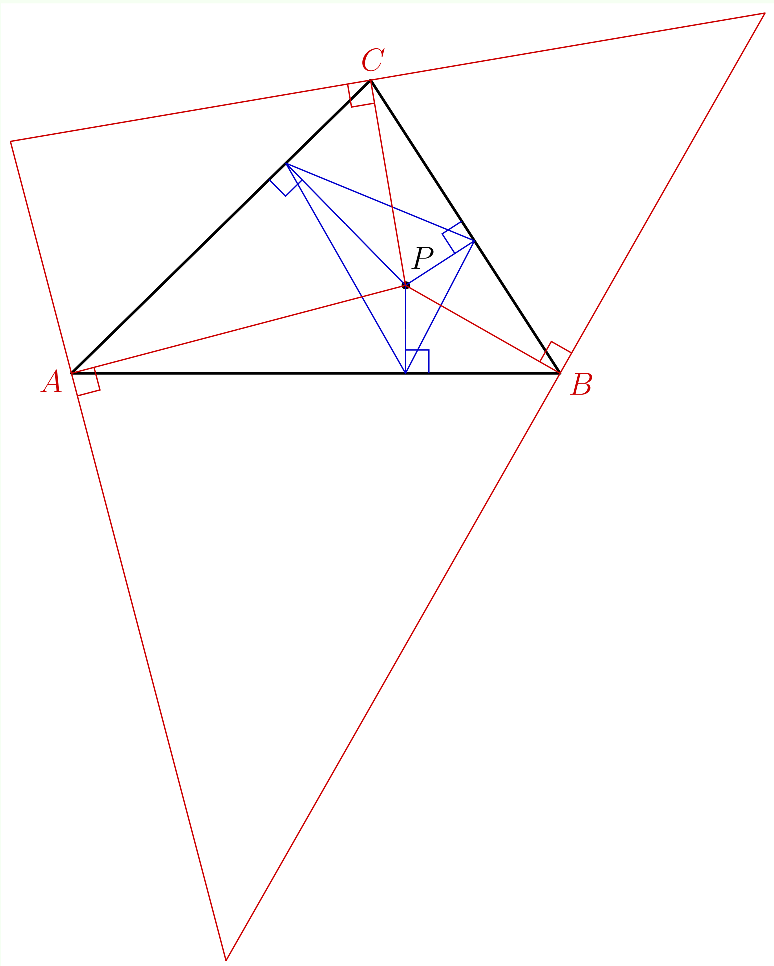
Renvoie la droite passant pas `M` et par le projeté orthogonal de `M` sur le côté `side`.
L'exemple suivant montre quelques droites de Simson ; on remarquera l'utilisation des méthodes `t.side(int)` et `t.vertex(int)` qui permettent de récupérer par leurs numéros les côtés et les sommets du triangle `t`.




```
import geometry; size(8cm,0);
triangle t=triangleabc(5,6,7);
label(t, alignFactor=4);
drawline(t, linewidth(bp));
circle C=circle(t); draw(C, bp+dashed);
pen[] p=new pen[] {0.8*red,0.8*blue,
                   0.8*green, 0.8*yellow};
for (int i=0; i < 4; ++i) {
  real x=35+i*90; point P=angpoint(C,x);
  dot("$P_"+(string)i+"$",P,dir(x),p[i]);
  for (int j=1; j < 4; ++j) {
    segment Sg=segment(pedal(t.side(j),P));
    draw(Sg,p[i]);
    markrightangle(P,Sg.B,t.vertex(j),p[i]);
  }
  drawline(pedal(t,P), bp+p[i]);
}
addMargins(1cm,1cm);
```

— `triangle antipedal(triangle t, point M)`

Renvoie le triangle dont le triangle podaire par rapport à `M` est `t`.




```
import geometry; size(10cm,0);
triangle t=triangleabc(5,6,7);
label(t); draw(t, linewidth(bp));
point P=0.5*t.B+0.3*t.C;
dot("$P$", P, 2*dir(60));

triangle Pt=pedal(t,P);
currentpen=0.8*blue; draw(Pt);
segment psA=segment(P,Pt.A);
segment psB=segment(P,Pt.B);
segment psC=segment(P,Pt.C);
draw(psA); draw(psB); draw(psC);
perpendicularmark(t.BC,psA);
perpendicularmark(t.CA,psB);
perpendicularmark(t.AB,psC);

triangle APt=antipedal(t, P);
currentpen=0.8*red; draw(APt);
segment apsA=segment(P,t.A);
segment apsB=segment(P,t.B);
segment apsC=segment(P,t.C);
draw(apsA); draw(apsB); draw(apsC);
perpendicularmark(APt.BC,apsA);
perpendicularmark(APt.CA,apsB);
perpendicularmark(APt.AB,apsC);
```

11.7. Coordonnées trilinéaires

Le type `trilinear`, dont la structure est donnée ci-après, permet d'instancier un objet représentant les coordonnées trilinéaires `a:b:c` par rapport au triangle `t`.

```
struct trilinear
{
  real a,b,c;
  triangle t;
}
```

Pour définir les coordonnées trilinéaires `a:b:c` par rapport à un triangle `t` on peut utiliser la routine `trilinear trilinear(tr`

Il est aussi possible de récupérer les coordonnées trilinéaires d'un point grâce à la routine `trilinear trilinear(triangle t,`

Il est enfin possible de définir des coordonnées trilinéaires grâce à une fonction de centre de triangle f et trois paramètres `a`, `b` et `c` en utilisant à la routine suivante :

```
trilinear trilinear(triangle t, centerfunction f,
real a=t.a(), real b=t.b(), real c=t.c())
```

où le type `centerfunction` représente une fonction réelle à trois variables réelles.

La conversion d'un objet de type `trilinear` en type `point` peut s'effectuer, comme d'habitude, de deux façon : avec la routine `point(trilinear)` ou par la syntaxe de « casting » `(point) trilinear`.

Par exemple, en utilisant les coordonnées trilinéaires du conjugués isotomique d'un point, voici comment est définie la routine `isotomicconjugate` :

```
point isotomicconjugate(triangle t, point M)
{
  trilinear tr=trilinear(t,M);
  return point(trilinear(t,1/(t.a()^2*tr.a),1/(t.b()^2*tr.b),1/(t.c()^2*tr.c)));
}
```

12. Inversions

Le type `inversion`, dont la structure est donnée ci-après, permet d'instancier l'inversion de pôle `C` et de puissance `k`

```
struct inversion
{
  point C;
  real k;
}
```

12.1. Définir une inversion

Les routines et opérateurs suivants permettent de définir une inversion.

— `inversion inversion(real k, point C)`

Renvoie l'inversion de pôle `C` et de puissance `k`. La routine `inversion(point C, real k)` est aussi disponible.

— `inversion inversion(circle c1, circle c2, real sgn=1)`

  — si `sgn` est non nul, cette routine renvoie l'inversion dont la puissance est du signe de `sgn` et transformant `c1` en `c2` ;

  — si `sgn` est nul, cette routine renvoie l'inversion centrée au pied de l'axe radical et laissant globalement invariants chacun des deux cercles `c1` et `c2`.

Un exemple utilisant cette routine a déjà été donné.

— `inversion inversion(circle c1, circle c2, circle c3)`

Renvoie l'inversion laissant globalement invariants les trois cercles `c1`, `c2` et `c3`.

— `circle operator cast(inversion i)`

Permet le « casting » d'un objet de type `inversion` en `circle`. Le cercle renvoyé est le cercle directeur (ou principal) de `i`.
On peut aussi forcer le « casting » grâce à la routine `circle circle(inversion i)`.

— `inversion operator cast(circle c)`

Permet le « casting » d'un objet de type `circle` en `inversion`. L'inversion renvoyée laisse globalement invariant `c`, a pour pôle le centre de `c` et le signe de la puissance est celui du rayon de `c`.
On peut aussi forcer le « casting » grâce à la routine `inversion inversion(circle c)`.

12.2. Appliquer une inversion

Les opérateurs suivants autorisent les codes du type `inversion*objet` qui renvoient l'image par `inversion` de l'objet `objet`.

— `point operator *(inversion i, point P)`

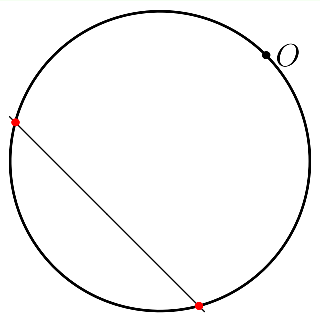
— `circle operator *(inversion i, line l)`

— `circle operator *(inversion i, circle c)`

— `arc operator *(inversion i, segment s)`

— `path operator *(inversion i, triangle t)`

On notera que l'inverse d'un cercle ou d'une droite peut être une droite. Dans ce cas le cercle `C` renvoyé a un rayon infini et la propriété `C.l` de type `line` est initialisée à la valeur adéquate ; les routines admettant ce cercle comme paramètre utiliseront `C.l` à la place comme le montre l'exemple suivant.

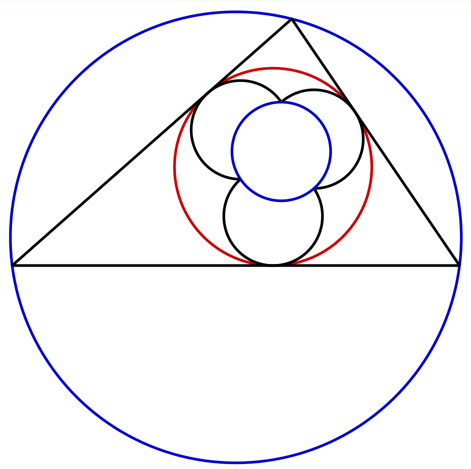



```
import geometry;
size(4cm);

circle C=circle((point)(0,0),1);
draw(C, linewidth(bp));

point O=dir(45);
dot("$O$",O);

inversion inv=inversion(3,O);
circle Cp=inv*C;
draw(Cp);

dot(intersectionpoints(C,Cp), red);
```

Cette fonctionnalité, rajoutée récemment, a été testée sommairement. Merci d'envoyer un rapport de bogue en cas de problème.

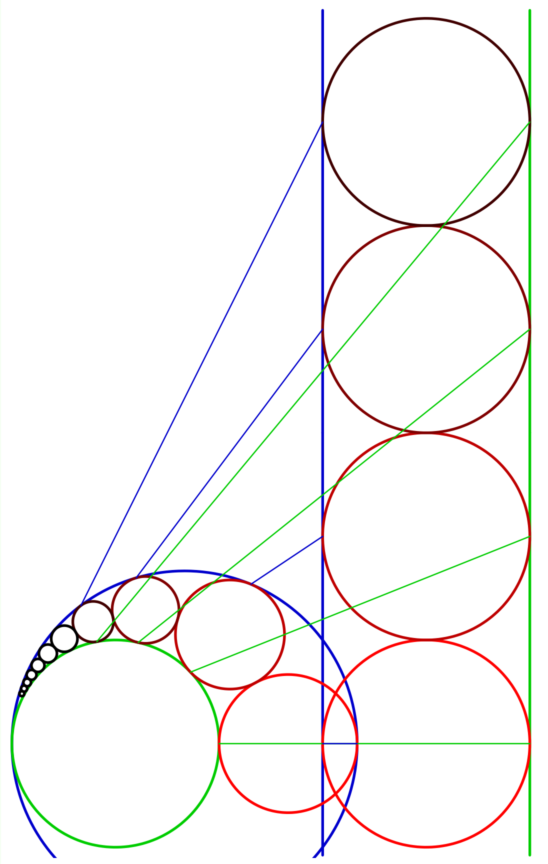
12.3. Exemples

Des exemples qui utilisent les inversions ont déjà été donnés, en voici d'autres.
Commençons par illustrer l'utilisation d'un cercle, ici le cercle inscrit à un triangle, en tant qu'inversion :



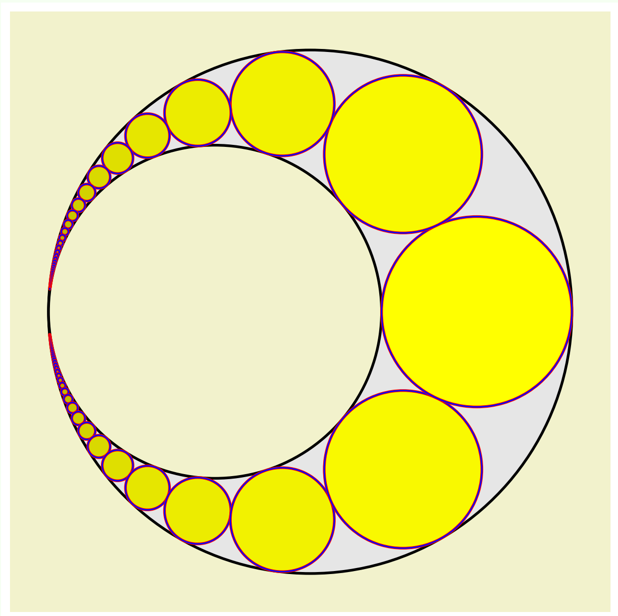
```
import geometry;
size(6cm,0);

triangle t=triangleabc(4,5,6);
circle C=circumcircle(t), inC=incircle(t);

draw(inC, bp+0.8*red);
draw(C, bp+0.8*blue);
draw(t, linewidth(bp));

draw(inC*t, linewidth(bp));
draw(inC*C, bp+0.8*blue);
```

Ci-dessous la construction du collier de Pappus.



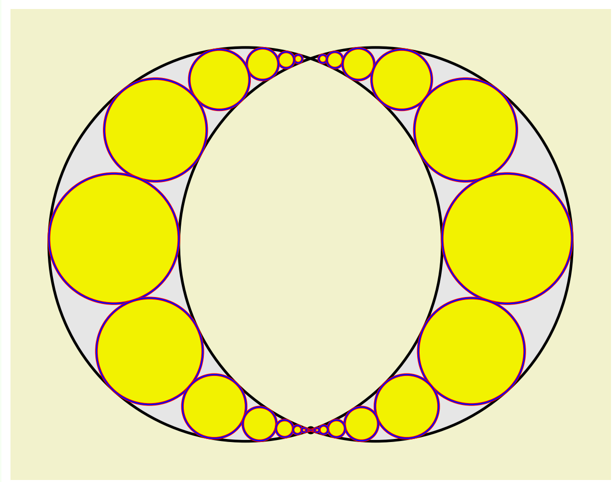
```
import geometry; size(7cm,0);
inversion inv=inversion(10,(-4,0));
line l1=line((-1,0),(-1,1)), l2=line((1,0),(1,1));
draw(l1, bp+0.8*blue); draw(l2, bp+0.8*green);
clipdraw(inv*l1,bp+0.8*blue);
clipdraw(inv*l2,bp+0.8*green);
int n=10;
for (int i=0; i <= n; ++i) {
  circle C=circle((point)(0,2*i),1);
  circle Cp=inv*C;
  draw(Cp,bp+(1-abs(i/4))*red);
  if(abs(i) < 4){
    draw(C,bp+(1-abs(i/4))*red);
    draw((1,2*i)--inv*(1,2*i),0.8*green);
    draw((-1,2*i)--inv*(-1,2*i),0.8*blue);
  }
}
addMargins(1mm,1mm);
```

On peut ainsi facilement en obtenir une belle représentation :



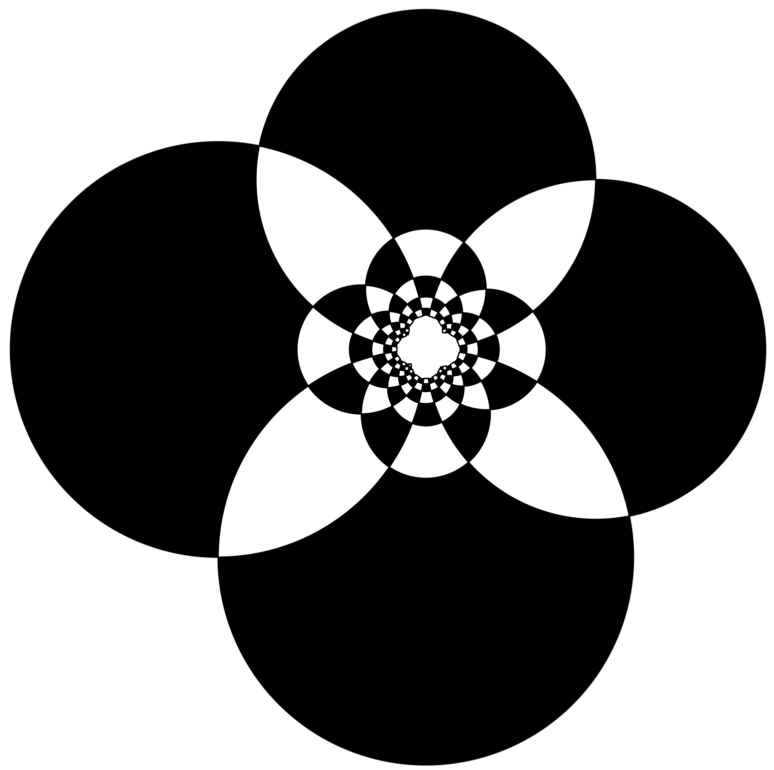
```
import geometry; size(8cm);

inversion inv=inversion(1,(-4.5,0));
path g1=inv*line((-1,0),(-1,1)),
g2=inv*line((1,0),(1,1));
fill(g1,lightgrey); draw(g1,linewidth(bp));
unfill(g2); draw(g2,linewidth(bp));

int n=40;
for (int i=-n; i <= n; ++i) {
  path g=inv*circle((point)(0,2*i),1);
  fill(g,(1-abs(i)/n)*yellow);
  draw(g,bp+red); draw(g,blue);
}
shipout(bbox(5mm,Fill(rgb(0.95,0.95,0.8))));
```

La figure suivante, où les droites ne sont pas parallèles, en est une variante :



```
import geometry; size(8cm,0);
point P=(0,-4.5); dot(P); inversion inv=inversion(1,P);
line l1=line((0,0),(1,0.35)), l2=line((0,0),(-1,0.35));
path g1=inv*l1, g2=inv*l2;
fill(g1^^g2,evenodd+lightgrey); draw(g1,linewidth(bp)); draw(g2,linewidth(bp));

for (int i:new int[]{-1,1}) {
  point P=(i*0.1,0); triangle t=triangle(shift(P)*vline,l1,l2); int n=15;
  for (int j=0; j <= n; ++j) {
    circle C=excircle(t.AB);
    t=triangle(shift(angpoint(C,(i-1)*90))*vline,l1,l2);
    circle Cp=inv*C; path g=Cp; fill(g,0.95*yellow); draw(g,bp+red); draw(g,blue); }}
shipout(bbox(5mm,Fill(rgb(0.95,0.95,0.8))));
```

La figure suivante est l'image d'un damier 12×12 par l'inversion dont le centre est proche « du centre du damier » et de rapport 1.



```
import geometry;
size(10cm,0);
int n=12; segment[] S;
inversion inv=inversion(1,(n/2+0.45,n/2+0.45));
transform tv=shift(0,1), th=shift(1,0);
for (int i=0; i < n; ++i)
  for (int j=0; j < n; ++j) {
    for (int l=0; l < 4 ; ++l)
      S[l]=segment(point(tv^i*th^j*unitsquare,l), point(tv^i*th^j*unitsquare,(l+1)%4));
    path g;
    for (int l=0; l < 4; ++l) g=g--(path)(inv*S[l]);
    g=g--cycle;
    if((i+j)%2 == 0)  draw(g); else fill(g);
  }
```

13. Index