Euclidean geometry with ASYMPTOTE

Original French Version by
Philippe Ivaldi

English Translation by
Olivier Guibé &
Philippe Ivaldi

Compiled with ASYMPTOTE version 2.14svn-r5318
on June 28, 2011

*Copyright © 2007 Philippe Ivaldi.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Lesser General Public License (see the file LICENSE in the top-level source directory).
Abstract

This document deals with the use of the package geometry.asy extension which makes the production of plane Euclidean geometry figures easier. The package geometry.asy extension defines some new types and routines for the Asymptote software.

First we will give the list with a brief description of the new types. Secondly we will study each of them and give details on the associated routines and operators.

Acknowledgements

I would like to thank

• Olivier Guié for numerous discussions on some computer algebra questions, his encouragements and his always attentive listening;
• John Bowman and Andy Hammerlindl without Asymptote does not exist;
• MB which has followed the development of this extension and has allowed to correct, improve and add some features.

Contents

1 Introduction ............................................................. 3
   1.1 Objects types list .................................................. 3
   1.2 Internal running ................................................... 4
   1.3 Index and external examples ..................................... 4
   1.4 Casting ............................................................ 4
2 Coordinate system ...................................................... 5
   2.1 The type coordsys ................................................. 5
   2.2 How to define a coordinate system ...........................................
   2.3 Changing an object pair on coordinate system ..................... 5
   2.4 Other routines .................................................... 6
3 Points and vectors ....................................................... 6
   3.1 The points ........................................................ 6
   3.1.1 Basic principles .............................................. 6
   3.1.2 Another routines ............................................. 8
   3.2 Vectors ........................................................... 10
   3.3 Basic principles ................................................ 10
   3.4 Vector/point casting ............................................ 11
   3.5 Another routines ............................................... 13
4 Mass points ............................................................. 13
   4.1 Basic Principles ................................................ 13
   4.2 Another routines ............................................... 13
5 Transformations (Part 1) ................................................. 16
   5.1 Independent coordinate system mappings ......................... 17
   5.2 Dependent coordinate system transformations .................. 19
6 Lines, half-lines and segments ........................................ 22
   6.1 The type “line” ................................................... 22
   6.1.1 Lines defined by two points, basic routines ..................... 22
   6.1.2 Lines defined by equations .................................... 23
   6.1.3 Lines and parallelism ......................................... 23
   6.1.4 Lines and angles ............................................. 24
   6.1.5 Lines and operators ......................................... 27
   6.1.6 Another routines ............................................. 28
   6.1.7 Line and marker ............................................. 29
   6.2 The type “segment” ............................................... 30
7 Affine Transformations (Part 2) ........................................ 31
8 Conics ................................................................. 33
   8.1 The type “conic” ................................................ 33
   8.1.1 Description ................................................... 33
   8.1.2 Basic routines ............................................... 34
   8.1.3 Operators ..................................................... 36
   8.1.4 Conics equations ............................................ 36
   8.1.5 Conics and casting .......................................... 37
   8.2 Circles .......................................................... 38
   8.2.1 basic routines ............................................... 38
The package `geometry.asy` defines many objects which are commonly used in plane Euclidean geometry; as Asymptote provides the types `real` and `path` to handle real number and path, the package `geometry.asy` includes many structures and functions for geometry objects. In the sequel it is important to distinguish an object and his type. For instance "Bivariate Quadratic Equation" is an object of type `bqe` and possesses a named object `a` of type `real[]`; one can accesses to it via the code `an_objet_bqe.a`.

Here all the types provided by the package `geometry.asy`:

- `coordsys` a cartesian coordinate system; this type is described in Section Coordinate system and Section Points and vectors explains his use;
- `point` and `vector` point and vector with respect to a cartesian coordinate system. The reading of Section Point and vectors which deals with this types can be omitted if the reader does not want to use another coordinate system than the default one: in this case one can consider that the types `point` and `vector` are the same than the type `pair`;
- `mass` mass point with respect to a cartesian coordinate system (cf. Mass points);
- `line` and `segment` the type `line` is devoted to line, half-line or segment. The type `segment`, for the definition of a segment, derives from the type `line` and is present to make easier the reading of the code. This types are described in Section Lines, half-lines and segments;
conic conic of any type (cf. Conics).
For optimization’s question 1 and code’s reading the derived types circle, ellipse, parabola, hyperbola, bqe (for 'Bivariate Quadratic Equation') are also defined.
Therefore according to your aims, the more restrictive type of conic should be chosen; of course a particular conic can be converted into the general conic type as shown in the following example

\begin{verbatim}
ellipse a_circle=circle ((0,0), 3);
...
conic a_conic=a_circle;
...
\end{verbatim}

arc instance of an ellipse arc (cf. Arcs);

abscissa abscissa on a line (in the general sense) or on a conic (cf. Abscissa);

triangle a triangle (cf. Triangles).
The code a_triangle.object allows to access to the objects which are connected to a triangle. Theses objects are of the types
side a triangle side (cf. Sides);
vertex a triangle vertex (cf. Vertexes).

trilinear trilinear coordinates with respect to a triangle (cf. Trilinear coordinates).

1.2. Internal running
The computations on a object given by one of the types defined by the package geometry.asy are performed with respect to the nature of the object and not with respect to his graphical representation which is finally a path or a pair.
Indeed the following code which computes and draws the intersection between two tangential circles is five times faster than an equivalent routine which uses the type path in place of the type circle.

\begin{verbatim}
import geometry;
size(3cm,0);
circle cle1=circle((point)(0,0), 1);
circle cle2=circle((point)(sqrt(2),sqrt(2)), 1);
dot(intersectionpoints(cle1, cle2));
\end{verbatim}

1.3. Index and external examples
An index and an examples gallery of all the routines, types and operators defined in the package geometry.asy allow a detailed view of this module:

- index with respect to the name of function;
- index with respect to the type of function;
- examples gallery.

1.4. Casting
The above cited objects can be processed by their own routines (the ones defined by the package geometry.asy) or by basic ASYMPOTOTE routines thanks to the casting. For example, a circle which is of type circle can be drawn directly with the standard routine draw thanks to the automatic casting circle into path while the code dot(a_circle); will return an error message because the routine dot exactly needs a type path; one must force the casting with the command dot((path)a_cercle);.

1 the method to compute the tangent to a circle is different in the case of an hyperbola
2. Coordinate system

If you do not plan to use another coordinate system than the default one, this section can be read later. In this case it is sufficient to know that the package geometry.asy uses the type point in place of the type pair and that the types vector and pair are equivalent.

The following paragraphs deal with the basic routines on cartesian coordinate systems, the real use of coordinate systems is detailed in Section Points and vectors.

2.1. The type coordsys

The package geometry.asy allows to define objects in any plane cartesian coordinate system; such a coordinate system is of the type coordsys. As shown below:

- the default coordinate system is defaultcoordsys, which is the native one used by ASYMPTOTE;
- the current coordinate system is currentcoordsys of which the default value is defaultcoordsys.

2.2. How to define a coordinate system

To define a coordinate system one can use the command cartesiansystem or one can apply a transformation to some coordinate system. For instance:

2.3. Changing an object pair on coordinate system

The examples of this section are given for information, the best method to define, modify, convert some coordinates into a coordinate system is to use the type point (see Points and vectors).

Actually the two main ways to define or to convert some coordinates of the type pair into a coordinate system are the operators:

- pair operator *(coordsys R, pair m)
  convert the coordinates of m given in the coordinate system R into coordinates in the default coordinate system. Thus, in the following example, the point M has (0.5;0.5) for coordinates in R and (2;2) in defaultcoordsys:
2.4. Other routines

- **path operator */(coordsys R, path g)**
  a code of the type `coordsys*path` returns a new path which is a reconstruction of the path `g` given in the coordinate system `R`. It is a generalization of the `coordsys*pair` for paths.

- **coordsys operator */(transform t, coordsys R)**
  allow the code `transform*coordsys`. Observe that `shiftless(t)` is applied to `R.i` and `R.j`.

3. Points and vectors

3.1. The points

3.1.1. Basic principles
While the type `pair` only allows to locate a point in the default coordinate system, the type `point` allows to locate a point in any cartesian coordinate system (see `coordsys`); an object of the type `point` refers to the coordinate system in which it is defined.

Thanks to the casting, a `point` is on the whole equivalent to a `pair` if one uses only the default coordinate system. Thus in the following example the `pair` `M` and the `point` `P` show the same point:
The following example shows the consequence of a modification of the current coordinate system on the casting pair in point for the point $A$ and how to define a point into a specific coordinate system through to the routine `point point(coordsys R, p)` for the point $B$.

```plaintext
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);
```

The routine `point locate(pair m)` also allows to convert directly a `pair` into `point` without point’s declaration:

```plaintext
import geometry;
size(4cm,0);
currentcoordsys=cartesiansystem((3,0), (1,0), (1,1));
show("", currentcoordsys);
point A=(1,1);
dot("$A$", A, red);
point B=point((0,1), A);
dot("$B$", B, W, blue);
```

The example below shows how to convert a type `pair` into a type `point` so that the result represents the same point and then how to obtain his coordinates into two distinct coordinate systems.

```plaintext
import geometry;
size(4cm,0);
coordsys R=cartesiansystem((2,3), i=(1,0), j=(0.5,1));
show(coordsys);
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); dot("", B, W, blue);
```

The routine `changecoordsys` allows to change easily the coordinate system of a `point`:
As for the type \texttt{pair} the operators $+$, $-$, $\ast$, $/$ are available for the type \texttt{point}. Observe that such an operation on two points defined into two different coordinate systems gives a \texttt{point} defined into the default coordinate system \texttt{defaultcoordsys}; a warning is given to the user about this automatic conversion.

Locating a point with polar coordinate is possible via the code \texttt{pair polar(real r, real angle)} of an object of the type \texttt{coordsys} as shown in the following example:

\begin{verbatim}
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);
\end{verbatim}

### 3.1.2. Another routines

Now that the basic routines concerning the type \texttt{point} are defined, here we give another routines:

- \texttt{point origin(coordsys R=currentcoordsys)}
  
  Return the origin of the coordinate system \texttt{R} as a point.
  The constant \texttt{point origin} is the origin of the default coordinate system.

- \texttt{point point(coordsys R, explicit point M, real m=M.m)}
  
  Return the point of mass \texttt{m} of which the relative coordinates to \texttt{R} are equal to the values of \texttt{M}. Do not confuse this routine with \texttt{changecoordsys}.

- \texttt{pair coordinates(point M)}
  
  Return the coordinates of \texttt{M} with respect to its coordinate system.

- \texttt{bool samecoordsys(bool warn=true ... point[] M)}
  
  Return \texttt{true} if and only if all the points are relative to the same coordinate system. If the value of \texttt{warn} is true and if the coordinate systems are different a warning is given.
Generalization of the routine `changecoordsys` to a array of points. Return, in a array form, the points into the same coordinate system \( R \). If the value of `warn` is true and if the coordinate systems of the points are different then a warning is given.

Casting of `point[]` into `pair[]`.

Return the coordinates of \( P \) in the default coordinate system.

Define `transform*point`. Remark that the transformations `scale`, `xscale`, `yscale` and `rotate` are defined with respect to the default coordinate system which is not wished in general when the current coordinate system has been modified.

To make up for this inconvenient one can use the routines `scale(real,point)`, `xscale(real,point)`, `yscale(real,point)`, `rotate(real,point)`, `scaleO(real)`, `xscaleO(real)`, `yscaleO(real)` and `rotateO(real)` which are described in Section Transformations (Part 1).

We assume that the sense of \( p2 \) is a point of \( p2 \) coordinates in the coordinate system where \( P1 \) is defined (point(coordinates(P1),p2)).

The same behavior than `finite(pair m)` in order to avoid computation with infinite coordinates.

Return the dot product \( A \cdot B \) with respect to the coordinate system of \( A \).

Return the dot product \( A \cdot B \) where the coordinates of \( A \) are first converted into the default coordinate system. `dot(explicit pair,point)` is also defined.
3.2. Vectors
3.3. Basic principles

In the following example the points $M$ and $P$ are defined relatively to two different coordinate systems. The point $Q$, namely the sum of $M$ and $P$, is obtained through the addition of the coordinates of $M$ and $P$ after their conversion into the default coordinate system: thus $\overrightarrow{OQ} = \overrightarrow{OM} + \overrightarrow{OP}$.

An object $u$ of the type $\text{vector}$ has the same behavior than an object of the type $\text{point}$ but his conversion into a $\text{pair}$ or a $\text{point}$ $M$ relatively to the default coordinate system is such that $\overrightarrow{u} = \overrightarrow{OM}$. It is shown in the example below which also uses the routine $\text{pair locate(vector v)}$:

The routines that are described for the type $\text{point}$ are also available for the type $\text{point}$. See below some simple examples:
3.4. Vector/point casting

With casting process an object point can be converted into an object vector and mutually an object vector can be converted into an object point. If I may repeat myself it is important to understand that the difference between point and vector does not exist if you do not use another coordinate system than the default one.

Observe in the following example the difference between dot(w) and dot(point(w)); in the second case the vector is converted into a point, the one “pointed out by the vector in the coordinate system R” while in the first case the vector is converted into a pair as explained above.

Notice that it is possible to write point M=w; in place of point M=point(w);
The attentive reader will understand the following example:

The counterpart of the routine `point(point(explicit vector))` is `vector(vector(point))`:

Finally the reader must pay attention to the routines `vector unit(point)` and `vector unit(vector)` which always return an object `vector`. Thus, in the following example the behavior of `point P=unit(B-A)` is not amazing while the one of `dot(unit(B-A))` can be dubious.
3.5. Another routines

As already mentioned all the routines devoted to the type type point can also be used with the type vector. Let us mention the easily understanding routine:

```cpp
bool collinear(vector u, vector v)
```

4. Mass points

4.1. Basic Principles

A point object possesses a mass which is attained with a_point.m and a routine allows to compute the center of a points set.

“Perfect!”

No... not quite:

- if one defines the point M with M=P1+P2; the mass of the point M is the sum of the masses of P1 and P2, which is fine but if the point M is defined by M=P/2 the coordinates of M are well half the ones of P but the mass is unchanged. Thus to define a point with a half mass of a point we should write:

```cpp
point P=point((1,1), 3);// point of mass 3
point Q=P;
Q.m=Q.m/2;
```

which can become quickly painful;

- to show the mass with an homogeneous way in all the figures at the time of using the routines dot and label can also be a painful task.

Mainly for this two reasons, the package geometry.asy defines a new type, the type mass which has at best an expected “mass point” behavior. For instance mass M=object_mass/2 defines the mass point M with the same coordinates than M but a half mass and the code point M=object_mass/2 has the same behavior but the result is directly converted into point.

The routines mass mass(explicit point) and point point(explicit mass) allow to overbalance easily between the two types mass and point; an elegant way to divide the mass of a point object is then the following:

```cpp
point P=point((1,1), 3);// Point of mass 3
point Q=mass(P)/2;// Division of the mass, the coordinates remain unchanged
```

The division of the coordinates of an object of the type mass is similarly:

```cpp
mass P=mass((1,1), 3);// Mass weight equal to 3
mass Q=point(P)/2;// Division of the coordinates, the weight remains unchanged
```

4.2. Another routines

Thanks to the casting all the features of the type point are available for the type mass with the already mentioned nuances. Here the list of another routines in connections with the mass points:

- **mass mass(coordsys R, explicit pair p, real m)**
  Return an object of the type mass with a weight m from which the coordinates into R are p.

- **mass mass(point M, real m)**
  Convert the point M into a mass of weight m.

- **point(explicit mass m)**
  Convert a mass of the type mass into a point of the type point.

- **mass mass(explicit point P)**
  Convert a point of the type point into a mass of the type mass.

- **mass masscenter(... mass[] M)**
  Compute the center of the masses array M. Thanks to the casting of point[] into mass[], notice that this routine also works with a parameter of the type point[].
• **string defaultmassformat;**

Default format used to construct the mass label. Its default value is "$\left(%L;%.4g\right)$" in which %L will be replaced by the label of the mass. The following example sheds a light on this routine:

```latex
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)**

Return a string which is formated by the command *format* with *format* as parameter, in which %L is replaced by *s* and the weight of *M*.

```latex
import geometry;
write(massformat(s="foo", mass((0,0),1000)));  // Return $\left(foo;1000\right)$
write(massformat("%L\_\%e", "foo", mass((0,0),1000)));  // Return foo\_1\!	imes\!10^{3\phantom{+}}
```

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

Draw the label returned by *massformat(format,L,M)* at the coordinates of *M*.

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

Draw a dot into the point *M* and the label returned by *massformat(format,L,M)*.

To conclude this section here are three examples using some of the above described routines.

```latex
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);
```
The following example shows how one can construct all the partial centers of n points, each center being connecting to the points from which it is emerging.
5. Transformations (Part 1)

In addition to the native affine mappings the package `geometry.asy` defines another plane transformations. Some of them have a specific behavior with respect the current coordinate system which is used. Thus, this section is divided into two subsections. In the first one we study the mappings which are independent of the current coordinate system. Secondly we detail the coordinate system dependent transformations.
5.1. Independent coordinate system mappings

- \textbf{transform scale}(\texttt{real} \ k, \ \texttt{point} \ M)

Central similarity of center M and coefficient k.

- \textbf{transform projection}(\texttt{point} \ A, \ \texttt{point} \ B)

Orthogonal projection onto the line \((AB)\).
• **transform projection**(point A, point B, point C, point D, bool safe=false)

Return the projection along (CD) onto (AB).
If the value of safe is true and if the line (AB) is parallel to the line (CD) then the identity is returned.
If the value of safe is false and if the line (AB) is parallel to the line (CD) then the central similarity of center O and infinite coefficient is returned.

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

Return the affine transformation of coefficient k, of axis (AB) and of direction (CD): if M is point of the Euclidean plane the image $M'$ is defined by $P + k \overrightarrow{PM}$ where P is the projection along (CD) onto (AB) of the point M.
If the value of safe is true and if the line (AB) is parallel to the line (CD), the identity is returned.
If the value of safe is false and if the line (AB) is parallel to the line (CD), the central similarity of center O and infinite coefficient is returned.
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);

code:

5.2. Dependent coordinate system transformations

1. \textbf{transform xscale(real } k \text{, point } M)\textbf{}

Affine transformation of coefficient \(k\), of axis “the line passing through \(M\) and parallel to (Oy)” and of direction (Ox).

The same example in any coordinate system:
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)
  Affine transformation of coefficient k, of axis “the line passing through M and parallel to (0x)” and of direction (0y).

M
A'
A

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);

The same example in any coordinate system:
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 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);

• \textbf{transform \texttt{scale0(real x)}}
Return the central similarity of coefficient x and center “the origin of the current coordinate system”. This transformation is equivalent to \texttt{scale(x, origin())}.
In the following example, one can see the difference between \texttt{scale(k)*P} and \texttt{scale0(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("scale0(k)*P", P2, W); draw((0,0)--locate(P1)); draw(origin()--P2);

• \textbf{transform \texttt{xscale0(real x)}}
Equivalent to \texttt{xscale(x, origin())} (see \texttt{xscale(real,point))}.

• \textbf{transform \texttt{yscale0(real x)}}
Equivalent to \texttt{yscale(x, origin())} (see \texttt{yscale(real,point))}.

• \textbf{transform \texttt{rotate0(real angle)}}
Equivalent to \texttt{rotate(angle, origin())}. 

21
6. Lines, half-lines and segments

6.1. The type “line"

An object of type line is devoted to describe line, half-line or segment depending on the properties value bool extendA, extendB; which is attained with the codes line.extendA and line.extendB. The full description about the methods and properties of the type line is accessible here.

6.1.1. Lines defined by two points, basic routines

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

Define an object of type line passing through the two points A and B and directed from A to B. If the value of extendA is true then the “line” goes on the side of A.

An object of type line belongs to the coordinate system in which the two points A and B are defined. If it is not the case the two points are automatically redefined in the current coordinate system and a warning message is given.

• line Ox(coordsys R=currentcoordsys)

Return the x-axis of the coordinate system R.

The routine line Oy(coordsys R=currentcoordsys) is also defined. The constants Ox and Oy are the axis of the default coordinate system.

• 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=ArrowBar, Label legend="", marker marker=NoMarker)

Draw into pic the “line” l without changing the image size provide that the position of the Label is correct and that the variable linemargin is non negative.

The boolean parameters dirA and dirB check the infinite part of the line to be drawn.

Remark that the real variable linemargin allows to control the margin between the border of the image and the line drawing. The default value is 0 and a negative value will change the size of the image.

A
B
CD
(AB)
[BC)
[CD]
[AD)

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)

Draw into pic the points which are used to define the line l as well as the direction of the line and the orthogonal direction.

• point intersectionpoint(line l1, line l2)

Return the intersection point between l1 and l2.

If the two lines do not intersect or if there is an infinite number of intersection points then this routine returns the point of coordinate (infinity, infinity). Remark that if the two lines are defined into two different coordinate systems, the intersection point is defined into the default coordinate system and a warning message is given.

The following example is an illustration of the famous Pascal theorem which claims that “if a hexagon is inscribed in a conic, then the three points at which the pairs of opposite sides meet, lie on a straight line” (here the conic is a circle).
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)
  Return an array of all the intersection points of the “line” l and the path g.

6.1.2. Lines defined by equations

• line line(coordsys R=currentcoordsys, real a, real b, real c)
  Return the line described by the linear equation $ax + by + c = 0$ in the coordinate system R.

• line line(coordsys R=currentcoordsys, real slope, real origin)
  Return the line of direction slope which the y-intersect is origin given in the coordinate system R.

6.1.3. Lines and parallelism

• line parallel(point M, line l)
  Return the line passing through M which is parallel to l.

• line parallel(point M, explicit vector dir)
  Return the line of direction dir and passing through M.

• line parallel(point M, explicit pair dir)
  Return the line of direction dir given in the current coordinate system and passing through M.

• bool parallel(line l1, line l2, bool strictly=false)
  Return true if l1 is parallel to l2 (strictly if the value of strictly is true).
6.1.4. Lines and angles

- **line line(real a, point A=point(currentcoordsys,(0,0)))**

  Return the line passing through A and making an angle a with the x-axis of the coordinate system in which A is defined. The routine line(point,real) is also defined.

- **line bisector(line 11, line 12, real angle=0, bool sharp=true)**

  Return the image of the bisector of the angle between the two oriented lines 11 and 12 by the rotation of center “the intersection between the lines 11 and 12” and angle angle.

  If the value of sharp is true, this routine returns the internal bisector.

  Remark that the returned line inherits of the coordinate system in which the line 11 is defined.
import geometry;
size(5cm,0);
point A=(0,0), B=(2*Cos(40),2*Sin(40)); line l1=line(A,B);
draw(l1, linewidth(bp));
line l2=rotate(100,A)*l1; draw(l2, linewidth(bp));
line bis=bisector(l1,l2); draw(bis, bp+blue);
line Bis=bisector(l1,l2,false); draw(Bis, bp+0.8*red);
markangleradiusfactor *= 4;
marker mark2=StickIntervalMarker(2, 1, red, true);
markangle(2, reverse(l2), reverse(l1), radius=-markangleradius(), red, mark2);
markangle(2, l1, reverse(l2), radius=-markangleradius(), blue, mark1);
draw(box((-1,-1),(1,1)), invisible);

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

Return the image of the $p$-th line which decompose the angle between the oriented lines $l1$ and $l2$ into $n$ equal parts by the rotation of center “the intersection between the lines $l1$ and $l2$” and angle $angle$.
If the value of $sharp$ is $true$, this routine is performed on the acute angle.
Remark that the returned line inherits of the coordinate system in which the line $l1$ is defined. See below an example to decompose an angle into three equal measure parts.
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)**
  Return the line passing through M and perpendicular to the line l.

- **line perpendicular(point M, explicit vector normal)**
  Return the line passing through M and of orthogonal direction (or normal vector) normal.

- **line perpendicular(point M, explicit pair normal)**
  Return the line passing through M and of orthogonal direction normal given in the current coordinate system 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 1, coordsys R=coordsys(1))
  Return the radian measure of the angle, into \((-\pi; \pi]\), with respect to the coordinate system R of the oriented line 1, i.e. the angle between the oriented line and the x-axis.

• real degrees(line 1, coordsys R=coordsys(1))
  Return the degree measure of the angle, into \([0; 360)\), with respect to the coordinate system R of the oriented line 1, i.e. the angle between the oriented line and the x-axis.

• real sharpangle(line 11, line 12)
  Return the radian measure of the acute oriented angle, into \(]-\frac{\pi}{2}; \frac{\pi}{2}\] , between 11 and 12.

• real sharpdegrees(line 11, line 12)
  Return the degree measure of the acute oriented angle, into \(]-90; 90]\), between 11 and 12.

• real angle(line 11, line 12)
  Return the radian measure of the oriented angle, into \((-\pi; \pi]\), between the two oriented lines 11 and 12.

• real degrees(line 11, line 12)
  Return the degree measure of the oriented angle, into \((-180; 180]\), between the two oriented lines 11 and 12.

6.1.5. Lines and operators

• line operator *(transform t, line 1)
  Allow the code transform*line.

• line operator /(line 1, real x)
  Allow the code line/real.
  Return the “line” passing through 1.A/x and 1.B/x.
  The code line operator *(real x, line 1) is also defined.

• line operator *(point M, line 1)
  Allow the code point*line.
  Return the “line” passing through unit(M)*1.A and unit(M)*1.B.

• line operator +(line 1, vector u)
  Allow the code line+vector.
  Return the image of 1 by the translation of vector u.
  The code line operator -(line 1, vector u) is also defined.
• \texttt{line[]} \ operator \ ^{\wedge}\text{(line} \ 11, \ \text{line} \ 12)\)

Allow the code \texttt{line}^{\wedge}\text{line}.
Return the array \texttt{new \ line[]} \ 11,12.

• \texttt{bool \ operator \ ==\text{(line} \ 11, \ \text{line} \ 12)\)

Allow the test \texttt{line} == \texttt{line}.
Return \texttt{true} if and only if the lines 11 and 12 are equal.

• \texttt{bool \ operator \ !=\text{(line} \ 11, \ \text{line} \ 12)\)

Allow the test \texttt{line} != \texttt{line}.
Return \texttt{false} if and only if the lines 11 and 12 are equal.

• \texttt{bool \ operator \ []\text{(point} \ m, \ \text{line} \ 1)\)

Allow the code \texttt{point} \ @ \ 	exttt{line}.
Return \texttt{true} if and only if the point \texttt{M} belongs to the object \texttt{1}.

6.1.6. Another routines

In this section we describe routines which concern lines. Some of them allow to get the \texttt{abscissa} of a point belonging to an object of type \texttt{line}.

• \texttt{void \ draw\text{(picture} \ pic=currentpicture, \texttt{Label[]} \ L=new \ 	exttt{Label}[], \ \texttt{line[]} \ 1, \ \texttt{align} \ align=\texttt{NoAlign, \ pen[]} \ p=new \ 	exttt{pen}[], \ \texttt{arrowbar} \ arrow=\texttt{None, \ 	exttt{Label[]} \ legend=new \ 	exttt{Label}[], \ 	exttt{marker} \ marker=\texttt{nomarker})\)

Draw every line defined in the array \texttt{line[]} \ 1 with the pen corresponding to the array \texttt{pen[]} \ p.
If \texttt{p} is not specified then the current pen is used.

• \texttt{void \ draw\text{(picture} \ pic=currentpicture, \texttt{Label[]} \ L=new \ 	exttt{Label}[], \ \texttt{line[]} \ 1, \ \texttt{align} \ align=\texttt{NoAlign, \ pen} \ p, \ \texttt{arrowbar} \ arrow=\texttt{None, \ 	exttt{Label[]} \ legend=new \ 	exttt{Label}[], \ 	exttt{marker} \ marker=\texttt{nomarker})\)

Draw every line defined in the array \texttt{line[]} \ 1 with the same pen \texttt{p}.

• \texttt{real \ distance\text{(point} \ M, \ \text{line} \ 1)}

Return the distance from \texttt{M} to 1.
\texttt{real \ distance\text{(line} \ 1, \texttt{point} \ M)} is also defined.

• \texttt{bool \ sameside\text{(point} \ M, \ \texttt{point} \ P, \ \text{line} \ 1)}

Return \texttt{true} if and only if \texttt{M} and \texttt{P} are on the same side of \texttt{1}.

• \texttt{point[]} \ sameside\text{(point} \ M, \ \texttt{line} \ 11, \ \texttt{line} \ 12)

Return an array of two points: the first point is the projection of \texttt{M} onto 11 along 12 and the second one is the projection of \texttt{M} onto 12 along 11.

• \texttt{coordsys \ coordsys\text{(line} \ 1)}

Return the coordinate system in which \texttt{1} is defined.

• \texttt{line \ changecoordsys\text{(coordsys} \ R, \ \text{line} \ 1)}

Return the “line” described by \texttt{1} in the coordinate system \texttt{R}.

• \texttt{line \ reverse\text{(line} \ 1)}

Return the “line” described by \texttt{1} with a contrary orientation.

• \texttt{line \ extend\text{(line} \ 1)}

Return the line which contains \texttt{1} which can be an half line or a segment.

• \texttt{line \ complementary\text{(explicit} \ 	ext{line} \ 1)}

If \texttt{1} is an half line, it returns the complementary half line of \texttt{1}.
• **bool** concurrent(... line[] l)
  Return true if and only if the lines of the array line[] l are concurrent.

• **bool** perpendicular(line 11, line 12)
  Return true if and only if the lines 11 and 12 are perpendicular.

• **point** point(line l, real x)
  Return the point between l.A and l.B as the code point(l.A−−l.B,x) does.

• **point** relpoint(line l, real x)
  Return the point of relative abscissa x with respect to the oriented segment [AB]. In other words the codes relpoint(l,x) and l.A+x*vector(l.B-l.A) are equivalent.

```latex
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)
  Return the point of abscissa x with respect to the coordinate system (l.A; 1−−u). In other words the code curpoint(l,x) and l.A+x*unit(l.B-l.A) are equivalent.

```latex
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. Line and marker

• **void** markangle(picture pic=currentpicture, Label L="", int n=1, real radius=0, real space=0, line 11, line 12, arrowbar arrow=None, pen p=currentpen, margin margin=NoMargin, marker marker=nomarker)
  This routine marks by n circle arc(s) the oriented angle between the lines 11 and 12. The arc(s) are drawn counterclockwise if radius is positive, clockwise otherwise. See this figure for an example.
void perpendicularmark(picture pic=currentpicture, line l1, line l2, real size=0, pen p=currentpen, int quarter=1, margin margin=NoMargin, filltype filltype=NoFill)

Mark a right angle to the intersection point of 11 and 12 in the quarter-th quarter of the plane counted in the counterclockwise, the first being the one delimited by the vectors 11.u and 12.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("", -l1.u, invisible);

6.2. The type “segment”

As announced in Introduction, the type segment, which deals with segment of line, derives from the type line. With casting, almost all the routines concerning the objects of type line are performed on a object of type segment and conversely.

However observe that when a segment is drawn the value of the variable addpenline is added to the used pen. The default value of this variable is squarecap in order to have right ends. It follows that the code draw(a_segment, dotted); do not produce a dotted segment.

There are three solutions to bypass this problem:

1. to write draw(a_segment,roundcap+dotted); instead of draw(a_segment, dotted);
2. to assign the variable addpenline the value nullpen
3. say to the author of the package geometry.asy that you do not agree about the default value of addpenline;

At last, like we can switch the types point and mass, the objects of types line and segment can be converted from one to another by writing for example segment s=an_obj_line; draw(segment(an_obj_line)); or draw(line(an_obj_segment));. The following example is an illustration

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);

Apart from the routines defined for the line objects, we also have some specific routines which deals with the segment objects:
- **segment segment(point A, point B)**
  Return the line segment with the end points A and B.

- **point midpoint(segment s)**
  Return the midpoint of the segment s.

- **line bisector(segment s, real angle=0)**
  Return the image to the perpendicular bisector of s by the rotation of center “the midpoint of s” and angle angle.

- **line[] complementary(explicit segment s)**
  Return the two half-lines which contain the segment s and which have the end point s.A and s.B respectively.

## 7. Affine Transformations (Part 2)

Some transformations which are defined from points in Section Affine transformations (Part 1) can also be defined from lines.

- **transform reflect(line 1)**
  Return the reflection with respect to 1.

```plaintext
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 11, line 12, bool safe=false)**
  Return the symmetry about 11 in the direction 12.
  If the value safe is true and if the lines 11 and 12 are parallel the routine returns the identity mapping.

```plaintext
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 11, line 12, bool safe=false)**
  Return the affine transformation of coefficient k of axis 11 and direction 12 (see the definition in Section 5.1, such an affine transformation is called “affinity”).
  If the value safe is true and if the lines 11 and 12 are parallel the routine returns the identity mapping.
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)**
  Return the orthogonal projection onto the line l.

- **transform projection(line l1, line l2, bool safe=false)**
  Return the projection along l2 onto l1.
  If the value of safe is true and if the lines l1 and l2 are parallel the routine returns the identity mapping.

- **transform vprojection(line l, bool safe=false)**
  Return the projection onto l along the y-axis.
  It is equivalent to projection(l,line(origin,point(defaultcoordsys,S)),safe).
  If the value of safe is true and if l is parallel to the y-axis the identity mapping is returned.

- **transform hprojection(line l, bool safe=false)**
  Return the projection onto l along the x-axis.
  It is equivalent to projection(l,line(origin,point(defaultcoordsys,E)),safe).
  If the value of safe is true and if l is parallel to the x-axis the identity mapping is returned.
8. Conics

8.1. The type “conic”

8.1.1. Description

The package `geometry.asy` defines the type `conic` to instantiate a conic section. While it is quite possible to use an instance of this type, its existence is rather intended for the internal workings of the extension. We will prefer to use directly derived types `circle`, `ellipse`, `parabola` and `hyperbola` described later.

A look to its structure in order to define the components:

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

- `e` is the eccentricity;
- `F` is a focus and `D` the associated directrix;
- `h` is the distance from `F` to `D`;
- `p` is the focal parameter ensuring the equality `p=he`.

The two main routines to define any conic are:

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

Return the conic specified by the focus `F`, which is associated to the directrix `l`, and eccentricity `e`; here is an example of use:
2. conic conic(point M1, point M2, point M3, point M4, point M5)

Return the non-degenerated conic passing through the points M1, M2, M3, M4 et M5.

It should be noticed that it is also possible to define a conic by its equation in a specified coordinate system, look at the section Conics equations, and that other ways to define a conic are implemented by routines referring to a specific conic type which may be converted to type conic; look at Conics and casting.

8.1.2. Basic routines

The following routines may be used replacing an object of type conic by one of types circle, ellipse, parabola or hyperbola except when the keyword explicit precedes the type conic in the definition of the routine.

It should be noted that, in addition to the routines described in this section, exist routines returning an abscissa of a point onto an object of type conic.
• **conic changecordsys(coordsys R, conic co)**
  Return the same conic as co relatively to the coordinate system R.

• **coordsys coordsys(conic co)**
  Return the coordinate system in which the conic co is defined.

• **coordsys canonicalcartesiansystem(explicit conic co)**
  Return the canonical coordinate system of the conic co.
  The routines `canonicalcartesiansystem(ellipse)`, `canonicalcartesiansystem(parabola)` and `canonicalcartesiansystem(hyperbola)` are also available.
  The following example is an illustration in the case of an 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)**
  Return the nodes number used to convert the conic co to path between the angles angle1 and angle2 given in the direction of course dir.

• **point[] intersectionpoints(conic co1, conic co2)**
  Return, in an array form, the intersection points of the two conics co1 and 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)

Return, in a array form, the intersection points of the line l with the conic co. The routine intersectionpoints(conic,line) is also defined.

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

Return, in an array form, the intersection points of the triangle t with the conic co. If extended is true, the sides of the triangle are regarded as extended lines; look at the section Triangles. The routine intersectionpoints(conic,triangle,bool) is also defined.

### 8.1.3. Operators

As with the previous routines, operators described here can be used by replacing an object of type conic by one of the types circle, ellipse, parabola or hyperbola.

- **bool operator @**(point M, conic co)
  
  Allow the code point @ conic.
  Return true if and only if the point M belongs to the object co.

- **conic operator ***(transform t, conic co)
  
  Allow the code transform*conic.

- **conic operator +(transform t, conic co)**
  
  Return the image of the conic co by the translation of vector \( \overrightarrow{Oz} \).
  The routine -(conic,explicit point) is also defined.

- **conic operator +(conic co, explicit point M)**
  
  Allow the code conic+point.
  Return the image of the conic co by the translation of vector \( \overrightarrow{Om} \); m represents the coordinates of a point defined relatively to the coordinate system in which the conic is defined.
  The routine -(conic,explicit pair) is also defined.

- **conic operator +(conic co, explicit pair m)**
  
  Allow the code conic+pair.
  Return the image of the conic co by the translation of vector \( \overrightarrow{Oz} \); \( m \) represents the coordinates of a point defined relatively to the coordinate system in which the conic is defined.
  The routine -(conic,explicit pair) is also defined.

- **conic operator +(conic co, explicit vector u)**
  
  Allow the code conic+vector.
  Return the image of the conic co by the translation of vector \( \overrightarrow{U} \).
  The routine -(conic,explicit vector) is also defined.

### 8.1.4. Conics equations

The type \( \text{bqe} \), for *Bivariate Quadratic Equation*, allows to instantiate an object representing an conic equation in a given coordinate system. His structure follows:

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

where:

- **a** is an arrow containing six coefficients of a conic equation given in the form
  
  \[
  \]

- **coordsys** is the coordinate system in which this equation is defined.

Here the list of routines regarding the objects of type \( \text{bqe} \):

- **bqe bqe(coordsys R=currentcoordsys, real a, real b, real c, real d, real e, real f)**
  
  Return an object of type \( \text{bqe} \) representing the equation \( ax^2 + bxy + cy^2 + dx + ey + f = 0 \) relatively to the coordinate system \( R \).
• \texttt{bqe changecoordsys(coordsys R, bqe bqe)}

Return an object of type \texttt{bqe} relatively to the coordinate system \texttt{R} and representing the same conic than that represented by the parameter \texttt{bqe}. This routine allows to change the coordinate system of a bivariate quadratic equation.

• \texttt{bqe bqe(point M1, point M2, point M3, point M4, point M5)}

Return an equation of the conic passing through the five points \texttt{M1}, \texttt{M2}, \texttt{M3}, \texttt{M4} and \texttt{M5}. If the points are defined relatively to the same coordinate system, the returned equation is relative to this coordinate system; else the equation is relative to the default coordinate system \texttt{defaultcoordsys}.

• \texttt{string conictype(bqe bqe)}

Return the type of conic represented by \texttt{bqe}. The possible returned values are "degenerated", "ellipse", "parabola" and "hyperbola".

• \texttt{bqe equation(explicit conic co)}

Return, in the form of object of type \texttt{bqe}, an equation of the conic \texttt{co}. The routines \texttt{equation(ellipse)}, \texttt{equation(parabola)} and \texttt{equation(hyperbola)} are also available.

• \texttt{bqe canonical(bqe bqe)}

Return an equation of the conic represented by \texttt{bqe} relatively to the canonical coordinate system of the considered conic.

• \texttt{conic conic(bqe bqe)}

Return the conic represented by \texttt{bqe}

\section*{8.1.5. Conics and casting}

As already mentioned in the previous sections, the type \texttt{conic} can instantiate an object representing any conic. It is however possible, and often recommended, to convert an object representing any conic in a specific type in order to use its own properties and routines.

The specific types of conics are \texttt{circle}, which is a particular case of the type \texttt{ellipse}, \texttt{parabola} and \texttt{hyperbola}; theses types are described in the sections below.

So in the example below the conical \texttt{co} is defined by a focus and the corresponding directrix with an eccentricity less than 1. Since this conic is an ellipse, one can assign to a variable of type \texttt{ellipse} the variable \texttt{co} in order to extract its 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=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);

From the point of view of the internal functioning of the package geometry.asy some routines that apply to an object of type conic call in fact equivalent routines applying to a specific conic; this allows to optimize some calculations. Conversely, routines for a specific type of conic use underlying routines for any conics.
The specific types of conics are described below.

8.2. Circles
8.2.1. basic routines
Besides the routines for objects of type conic, here are some other routines to define an object of type circle:

• \textbf{circle circle(\textbf{explicit point} C, real r)}

Return the circle of radius \( r \) centered on \( r \).
Remark that the routine \textbf{circle circle(pair C, real r)} in no longer redefined since the ASYMPTOTE version 2.10; one must use the casting of pair to point with the syntax \textbf{circle cle = circle((point)(1,2), 2)} whose center (1,2) represents the coordinates of a point in the current coordinate system \textbf{currentcoordsys}.

The following example illustrates the difference between the code \textbf{circle((0,0),R)} which defines the blue circle in the current coordinate system and \textbf{circle(point(defaultcoordsys,(0,0)), R)} which defines the red circle in the default coordinate system; evidently, if the variable \textbf{currentcoordsys} is not modified, the two codes are equivalent.
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)
  Return the circle of diameter AB.

• circle circle(point A, point B, point C)
  Return the circle passing through the distinct points A, B et C.
  An alias of this routine is circle circumcircle(point A, point B, point C).

• circle incircle(point A, point B, point C)
  Return the inscribed circle of the triangle ABC.

• circle excircle(point A, point B, point C)
  Return the excircle of the triangle ABC tangent to (AB).

In the example below we can see the use of the routine clipdraw which draw a path by restricting the size of the final picture.

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);

Specific routines to the geometry of the triangle will achieve the same result in a more elegant way, see the section Triangles.
8.2.2. From the type "circle" to the type "path"

The casting of an object of type circle to type path is done according to the following rules:

- the path is cyclic counterclockwise;
- the first node of the path, as returned by the routine pair point(path g, real t) with t=0, is the intersection point of the circle with the half-line through the center and parallel to axis of the coordinate system in which the circle is defined;
- the number of points of the path depends on the radius circle; it is calculated by the routine int circlenodesnumber(real r) which depends itself of the variable circlenodesnumberfactor.

![Diagram showing the conversion from a circle to a path.](image)

8.2.3. The operators

Apart from operators applying to objects of type conic, here are the list of others operators defined for the objects of type circle.

- **circle operator *(real x, explicit circle c)**

  Allow the code real*circle.
  Return the circle with same center as c and with radius x time that of c.
  The operator circle operator /(explicit circle c, real x) is also defined.

- **real operator ^(point M, explicit circle c)**

  Allow the code point^circle.
  Return the circle power of M with respect to c.

- **bool operator @(point M, explicit circle c)**

  Allow the code point @ circle.
  Return true if and only if the point M is on the circumference of c.

- **ellipse operator cast(circle c)**

  Allow the casting from circle to ellipse.
  The casting from ellipse to circle is also defined.
  One will note that the operator *(transform t, circle c) does not exist; through the casting, this is the operator ellipse operator *(transform t, ellipse el) which is used while executing the code transform*circle.
  Thus the code scale(2)*circle return an object of type ellipse but it is possible to write circle=scale(2)*circle while the code circle=xscale(2)*circle generates an error.
8.2.4. Other routines

Apart routines applied to objects of type conic, here is the list of specific routines for objects of type circle.

- **point radicalcenter(circle c1, circle c2)**
  
  Return the foot of the radical line of the two circle c1 and c2. The coordinate system in which is defined the returned point is that of c1.

- **point radicalcenter(circle c1, circle c2, circle c3)**
  
  Return the radical center of the three circles c1, c2 and c3.

- **line radicalline(circle c1, circle c2)**
  
  Return the radical line of the two circles c1 and c2.

- **line[] tangents(circle c, point M)**
  
  Return the eventual tangents to c passing through M.

- **line tangent(circle c, abscissa x)**
  
  Return the tangent to c at the point whose the abscissa is x on c.

- **point point(explicit circle c, real x)**
  
  Return the point of c marking the same point than the returned pair by th code point((path)c,x).

- **point relpoint(explicit circle c, real x)**
  
  Return the point of c corresponding to x time “the perimeter of c”.

```plaintext
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);
```
• **point angpoint(explicit circle c, real x)**

Return the point of c corresponding with the angle x degrees.

• **point curpoint(explicit circle c, real x)**

Return the point of c whose the curvilinear abscissa is 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);  

Others routines are defined for the objects of type circle, they are accessible via some routines using the type ellipse.  

To conclude this section note that it is possible to use an object of type circle as an inversion. See the section Inversions for further details.  

Here are some examples of uses of routines previously described:  

- Construction of two circles through the points $A$, $B$ and tangent to a given line $d$.  

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);

• Two points of the plane and their inverses are cocyclic, on the orthogonal circle of the inversion circle.
Given three circles $C_1$, $C_2$ and $C_3$ such as $r_3 < r_1$ and $r_3 < r_2$, how construct circles simultaneously tangent to these circles?

The principle of construction illustrated below is as follows:

- If $S_1$ et $S_2$ are the inversions with positive radius transforming $C_2$ to $C_3$ and $C_1$ to $C_3$ respectively;
- considering a point $M$ on the cercle $C_3$ and if $M_1$ and $M_2$ they images about $S_1$ et $S_2$ respectively;
- if $C$ is the circle passing through $M$, $M_1$ and $M_2$;
- the radical line $(d)$ of circles $C_3$ and $C$ cuts the line passing through the inversions center $(S_1S_2)$ in $H$; if $C'$ is the cercle of diameter $[HO_3]$ where $O_3$ is the center of $C_3$;

then:

- the circle $C'$ cuts $C_3$ in two points $T_0$ et $T_1$;
- the circle passing through $T_0$ and through the images $T'_0$ and $T''_0$ of $T_0$ about $S_1$ et $S_2$ respectively is one solution;
- the circle passing through $T_1$ and through the images $T'_1$ and $T''_1$ of $T_1$ about $S_1$ et $S_2$ respectively is an other solution;
Taking the four possible combinations for the inversions $S_1$ and $S_2$, we get the eight circles tangents to three given circles likewise:
import geometry;
size(18cm,0); int shx=18;
circle C1=circle((point)(0,0),2), C2=circle((point)(6,5), 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*(i+1))*tpic[0]);
  l=(l+2)%4; ++k;
  add(tv"^((shx*(i+1)))*th^((shx*(i+1))*tpic[1]);
}

8.3. Ellipses
The type ellipse does not surprise, it allows to instantiate an object representing an ellipse. As the type circle is a particular case of the type ellipse it is possible to instantiate a circle as an ellipse zero eccentricity and, reciprocally, to instantiate an zero eccentricity ellipse as a circle. Finally, as there is a one-to-one correspondence between objects of type ellipse and those type conic with eccentricity strictly less than 1, objects of type ellipse inherit routines and operators set for those type conic.
8.3.1. Basic routines

Here is a list of other routines to define a type `ellipse`.

- `ellipse ellipse(point F1, point F2, real a)`
  - Return the ellipse foci `F1` and `F2` with semimajor axis `a`.

- `ellipse ellipse(point F1, point F2, point M)`
  - Return the ellipse foci `F1` and `F2` through `M`.

- `ellipse ellipse(point C, real a, real b, real angle=0)`
  - Return the ellipse centered at `C` with semimajor axis `a` in the direction given by `dir(angle)` and semiminor axis `b`.

8.3.2. From the type “ellipse” to the type “path”

The casting of an object of type `ellipse` to `path` is done according to the following rules:

- the path is cyclic, counterclockwise oriented;
- the first point, one returned by the routine `pair point(path g, real t)` with `t=0`, is the intersection point of the half focal line `[F1F2]` with the ellipse;
- the nodes number of the path depends to the axis lengths of the ellipse; it is computed by the routine `int ellipsenodesnumber(real a, real b)` which itself depends on the variable `ellipsenodesnumberfactor`;
- the nodes of the path are defined in polar coordinates with angles relatively to the center of the ellipse and evenly distributed throughout the interval `[0; 360]`. 
8.3.3. Others routines

Apart routines applied to objects of type conic, here is the list of routines specific to objects of type ellipse.

- **real centerToFocus(ellipse el, real a)**
  
  Allow to convert an angle from the center of the ellipse to an angle form the first focus.
  
  The routine **real focusToCenter(ellipse,real)** is also defined.

- **real arclength(ellipse el, real angle1, real angle2,**
  
  bool direction=CCW,
  
  polarconicroutine polarconicroutine=currentpolarconicroutine)**

  Return the length of the ellipse arc defined by el from angle angle1 to angle2 (in degrees) in the direction direction.
  
  The possible values of polarconicroutine are arcfromfocus, which is the default value of currentpolarconicroutine, or arcfromcenter; in the first case the angles are relative to the first focus, in the second case, they are given relatively to the center of the ellipse.

- **line[] tangents(ellipse el, point M)**

  Return the eventual tangents to el through M.

The following example illustrates the definition of ellipse as the image of a circle by a dilatation and a related property of its tangents.
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'$\, $H'$");
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*Np; L=segment(H,M); Lp=segment(Hp,NNp);
dot("$M\$, M", NE); dot("$M'$\, $M'$"); dot("$N\$, NN, SW); dot("$N'$\, $N'$");
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'$\, $P'$", 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+2tgM.u--Mp-2tgM.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)**

Return the point of `el` which marks the same point as the pair returned by the code `point((path)el,x)`.

• **point relpoint(explicit ellipse el, real x)**

Return the point of `el` which corresponds to the fraction `x` of the perimeter of `el`. 50
• **point angpoint(explicit ellipse el, real x, polarconicroutine polarconicroutine=custompolarconicroutine)**

Return the point of el with angle x degrees from the center of the ellipse if polarconicroutine=fromCenter, from the first focus if polarconicroutine=fromFocus.

• **point curpoint(explicit ellipse c, real x)**

Return the point of c whose curvilinear abscissa is x.
8.4. Paraboles

This is the type `parabola` which allows to instantiate a parabola. As there is a one-to-one correspondence between objects of type `parabola` and those of type `conic` with eccentricity equal to 1, objects of type `parabola` inherit routines and operators set for those of type `conic`.

8.4.1. Basic routines

The routines available to define a parabola are:

- **parabola parabola(point F, line l)**
  - Return the parabola with focus F and directrix l.

- **parabola parabola(point F, point vertex)**
  - Return the parabola with focus F and vertex vertex.

- **parabola parabola(point F, real a, real angle)**
  - Return the parabola with focus F, with latus rectum a (the chord through a focus parallel to the conic section directrix) and whose the axis make an angle angle with the abscissa axis in which the point F is defined.

- **parabola parabola(point M1, point M2, point M3, line l)**
  - Return the parabola through the points M1, M2, M3 and whose the directrix is parallel to the line l.
8.4.2. From type “parabola” to type “path”

The casting of an object $P$ of type \texttt{parabola} to \texttt{path} is done according to the following rules:

- the path is oriented counterclockwise;

- the path is content, if possible:
  1. in the current picture if the variables $P.bmin$ and $P.bmax$, type \texttt{pair}, have not been altered;
  2. in the rectangle $\text{box}(P.bmin,\text{box}(P.bmax))$ otherwise.

So in the following example, when the first conversion to path, the picture size is symbolized dashed and the path can not contain in this rectangle. During the second conversion, changing variables $p.bmin$ and $p.bmax$ redefines the area of conversion which is plotted in red with a corresponding portion of parabola.

\begin{center}
\begin{itemize}
  \item the number of nodes in the path depends on angles, given from focus in degrees, of the ends of the path and is calculated through the routine \texttt{int parabolanodesnumber(parabola p, real angle1, real angle2)} which itself depends on the variable \texttt{parabolanodesnumberfactor};
  \item the node in the path are defined in polar coordinates with angles given from parabola focus and evenly distributed throughout the interval whose ends are returned by the routine \texttt{real[] bangles(picture pic=\texttt{currentpicture}, parabola p)}.
\end{itemize}
\end{center}

8.4.3. Others routines

Apart routines applied to objects of type \texttt{conic}, here the list of specific routines to objects of type \texttt{parabola}.

\begin{itemize}
\end{itemize}
Return the eventual tangents to \( p \) through \( M \).

\[
\text{line}[] \text{tangents(parabola} \ p, \ \text{point} \ M)\]

Return the tangent to \( p \) at the \( x \) abscissa point of \( p \).

\[
\text{point} \ \text{point(explicit parabola} \ p, \ \text{real} \ x)\]

Return the point of \( p \) which marks the same point as the \text{pair} returned by the code \text{point((path)p,x)}.

\[
\text{point} \ \text{relpoint(explicit parabola} \ p, \ \text{real} \ x)\]

Return the point of \( p \) which marks the same point as the \text{pair} returned by the code \text{relpoint((path)p,x)}.

\[
\text{point} \ \text{angpoint(explicit parabola} \ p, \ \text{real} \ x)\]

Return the point of \( p \) with angle \( x \) degrees.

\[
\text{point} \ \text{curpoint(explicit parabola} \ p, \ \text{real} \ x)\]

Return the point of \( p \) whose the curvilinear abscissa is \( x \), the origin being the vertex of the parabola.

\[
\text{It is possible to obtain an arc of parabola in the form of path through the routine}
\]

\[
\text{path} \ \text{arcfromfocus(conic} \ co, \ \text{real} \ \text{angle}1, \ \text{real} \ \text{angle}2, \ \text{int} \ n=400, \ \text{bool} \ direction=\text{CCW})\]

Although this routine is available for any kind of conic its use has really of interest for the parabolas and hyperbolas; ellipse arcs have a specific type described in section \text{Arcs}. Here an example illustrating the use of routine \text{arcfromfocus} with a parabola.
8.5. Hyperboles

This is the type `hyperbola` which allows to instantiate an hyperbola. As there is a one-to-one correspondence between objects of type `hyperbola` and those of type `conic` with eccentricity strictly greater than 1, objects of type `hyperbola` inherit routines and operators set for those of type `conic`.

8.5.1. Basic routines

The routines available to define an hyperbola are:

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

  If `byfoci=true`: return the hyperbola with semi-major axis `ae` and with foci `P1` and `P2`;
  
  If `byfoci=false`: return the hyperbola with eccentricity `ae` and with vertices `P1` and `P2`;

  For more legibility, the constants `byfoci` and `byvertices` are defined, Their values are `true` and `false` respectively.
import geometry; size(8cm);
point P1=(-3,0), P2=(3,0);
draw(box((-5,-5),(5,5)), invisible);
hyperbola H=hyperbola(P1,P2,2);
draw(H, linewidth(bp));
dot("$F_1$", H.F1);
dot("$F_2$", H.F2, W);
dot("$V_1$", H.V1, E);
dot("$V_2$", H.V2, W);
distance("$a$", H.C, H.V1, 2cm, joinpen=dotted);
distance("$c$", H.C, H.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);

8.5.2. From type “hyperbola” to type “path”

The casting of an object $H$ of type hyperbola to path is done according to the following rules:

- the path is the hyperbola branch with focus $H.F1$ oriented in trigonometric direction;

- the path is contained, if possible:
  1. in the current picture if the variable $H.bmin$ and $H.bmax$, type pair, have not been changed;
  2. in the box box((H.bmin),box(H.bmax)) otherwise.

So in the following example, when the first conversion to path, the picture size is symbolized dashed and the path can not contain in this rectangle. During the second conversion, changing variables $H.bmin$ and $H.bmax$ redefines the area of conversion which is plotted in red with a corresponding portion of hyperbola.
the number of node in the path depends on angles, given from focus in degrees, of the ends of the path and is calculated through the routine `int hyperbolanodesnumber(hyperbola p, real angle1, real angle2)` which itself depends on the variable `hyperbolanodesnumberfactor`;

- the nodes in the path are defined in polar coordinates with angles given from focus `H.F1` and evenly distributed throughout the interval whose ends are returned by the routine `real[][] bangles(picture pic=currentpicture, hyperbola p)`.

8.5.3. Others routines

Apart routines applied to objects of type `conic`, here the list of specific routines to objects of type `hyperbola`.

- `line[] tangents(hyperbola h, point M)`
  Return the eventual tangents to `p` through `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)
    Return the tangent to h at the x abscissa point of el1.

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)
    Return the point of h which marks the same point as the pair returned by the code point((path)h,x).

• point relpoint(explicit hyperbola h, real x)
    Return the point of h which marks the same point as the pair returned by the code relpoint((path)h,x).

• point angpoint(explicit hyperbola h, real x, polarconicroutine polarconicroutine=currentpolarconicroutine)
    Return the point of p with angle x degrees from the hyperbola center if polarconicroutine=fromCenter, from the first focus if polarconicroutine=fromFocus. Two examples are given further.

• It is possible to obtain an arc of hyperbola in the form of path through the following routines
1. **path arcfromfocus**(conic co, real angle1, real angle2, int n=400, bool direction=CCW)

Although this routine is available for any kind of conic its use has really of interest to the parabolas and hyperbolas; ellipse arcs have a specific type described in section Arcs.

Here an example illustrating the use of routine arcfromfocus with an hyperbola.

```plaintext
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=canonicalcartesianystem(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);
```

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

Here an example illustrating the use of routine arcfromcenter with an hyperbola.

```plaintext
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=canonicalcartesianystem(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(arcfromcenter(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);
```
9. Arcs

The type `arc` allows to instantiate an oriented ellipse arc. The main routine to define a such arc is described below.

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

Return an arc of the ellipse `el` from angles (in degrees) `angle1` to `angle2` in the direction `direction` and given relatively to the first focus if `polarconicroutine=fromFocus`, relatively to the center of the ellipse if `polarconicroutine=fromCenter`.

The routine `polarconicroutine polarconicroutine(conic co)` used here to determinate the default value of the parameter `polarconicroutine` returns in this case `fromCenter` if `co` represents a circle, `currentpolarconicroutine`, which is `fromFocus` by default, if `co` represents an ellipse.

It is important to note that, when drawn arc, the value of the variable `addpenarc` is added to the used pen. By default this variable is set to `squarecap`, in order to have the ends straight, which makes the display of a dotted arc inefficient. To circumvent this problem there are three solutions:

1. write `draw(a_arc, roundcap+dotted);` instead of `draw(a_arc, dotted);`

2. set the value of `addpenarc` to `nullpen`.

3. contact the author of the package `geometry.asy` to inform him of your disagreement with the default of `addpenline`.

Here are a few examples illustrating the use of routine `arc(ellipse, real, real, polarconicroutine, bool)`

- The following example shows how to obtain an ellipse arc whose angles are given from his first focus, that is the default behavior. Note the use of the routine `markarc` which will be described further.

```asy
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("u",blue), Label("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\degree", Ox(Rp), (line) s, radius=2cm, Arrow);
```
The following example shows the effects of the parameters \texttt{polarconicroutine} and \texttt{direction}. Note the use of the routine \texttt{degrees(arc)} which will be described further.

The example below shows the previous code permuting the angles 45° and 210°.
9.1. From type “arc” to type “path”

The casting of an object $A$ of type $arc$ to $path$ is done according to the following rules:

- the path is oriented in the direction $A$.direction;
- the number of nodes is computed by the routine int arcnodesnumber(explicit arc a) which itself depends on the variable ellipsetronesnumberfactor;
- the nodes in the path are defined in polar coordinates with angles given from the first focus or from the ellipse center depending to the value of $A$.polarconicroutine and evenly distributed throughout an adequate interval.

9.2. The operators

- **arc operator *(transform t, explicit arc a)**

  Allow the code $\text{transform} \ast \text{arc}$ whose behaviour is without surprise. In the following example the colored arcs are images of the black arc about affine transformations.
The same example with an arc circle:

```plaintext
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+0.8yellow, 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)

Allow the code `real*arc`.

Return the arc `a` with the angles `a.angle1-(x-1)*degrees(a)/2` and `a.angle2+(x-1)*degrees(a)/2`. The operator `/explicit arc,real` is also defined.

In the following example the yellow arc is obtained multiplying the black arc by 0.5 and the blue arc is obtained dividing it by 3.
The same thing with an ellipse arc defined from the center of the ellipse.

Finally, in the following example, the arc is defined from the first focus of the ellipse:
• **arc operator +(explicit arc a, point M)**
  
  Allow the code `arc+point` which is an alias for `shift(point)*arc`. The operators `-(explicit arc,point),+(explicit arc,vector) and -(explicit arc,vector)` are also defined.

• **bool operator @ (point M, arc a)**
  
  Allow the code `point @ arc`. Return `true` if and only if the point `M` belongs to the arc `a`.

• **arc operator *(inversion i, segment s)**
  
  Allow the code `inversion*segment`. Return the image of `s` about the `i` inversion; one can look at the illustration of `inversion*segment` at the section Inversions.

### 9.3. Others routines

In addition to the routines described in this section added routines to locate a point on an object of type arc are described in the section Abscissas.

• **arc complementary(arc a)**
  
  Return the complementary of the arc `a`.

• **arc reverse(arc a)**
  
  Return the reversed arc of `a` as would the routine `reverse(path)` does.

• **real degrees(arc a)**
  
  Return the angle of the oriented arc `a` in degrees in the interval `[-360; 360]`. The routine `angle(arc)` is also defined for angle in radians.

• **real arclength(arc a)**
  
  Return the length of the arc `a`.  

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

Allow to mark the angle a with an arc circle. The parameter sectorpen is the pen used to mark the segments which links the center or the focus of the arc with its ends. The parameter markpen is the pen used to draw the arc circle which may itself be marked with the parameter marker. Examples of use have already been given.

**point[] intersectionpoints** (arc a1, arc a2)

Return, in a array form, the intersection points of two arcs. The intersection routines of an object arc with other objects defined by the package geometry.asy are also defined; for example intersectionpoints(conic co, arc a), intersectionpoints(arc a, conic co), intersectionpoints(line l, arc a) etc...

**arc arcsubtended** (point A, point B, real angle)

Return the subtended arc of the line segment [AB] with an angle angle. Although the code a_arcsubtended.C allows to recover the center of the subtended arc, it is possible to obtain it using the routine point arcsubtendedcenter(point A, point B, real angle).
• **arc arccircle(point A, point B, real angle, bool direction=CCW)**

Return the arc circle, centered in A, from B to the image of B under the rotation with center A and angle \( \text{angle} \) in the direction \( \text{direction} \).

• **arc arccircle(point A, point M, point B)**

Return the arc circle \( \widehat{AB} \) through M.

• **arc arc(ellipse el, point M, point N, bool direction=CCW)**

Return the arc of \( \text{el} \), with direction \( \text{direction} \), with ends M and N which must belong \( \text{el} \).
**arc arc(ellipse el, explicit abscissa x1, explicit abscissa x2, bool direction=CCW)**

This routine has the same behavior as the previous routine but the points are specified with abscissas relatively to the ellipse.

**arc arc(explicit arc a, point M, point N)**

Return the arc part a between M and N.
This routine has the same behavior as the previous routine but the points are specified with abscissas relatively to the ellipse.

- `arc inverse(real k, point A, segment s)`
  Return the image of \( s \) under the inversion with center \( A \) and radius \( k \); look at the illustration of inversion segment in the section Inversions.

- `line tangent(explicit arc a, point M)`
  Return the tangent to \( a \) at point \( M \) of \( a \).

- `line tangent(explicit arc a, abscissa x)`
  Return the tangent to \( a \) at the point of abscissa \( x \) given relatively to \( a \).

10. Abscissas

The type `abscissa` allows to instantiate an abscissa on an object of type `line`, `segment`, `conic` and `arc`. The structure of an object of type `abscissa` follows:

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

\( x \) is the value of the abscissa.

`system` represents the type of abscissa:

- 0 for an abscissa as fraction of the length of path;
- 1 for a curvilinear abscissa;
- 2 for an angular abscissa;
- 3 for an abscissa relative to the nodes of a path.

For better readability of the code, the following constants are predefined:

```cpp
int relativesystem=0, curvilinearsystem=1, angularsystem=2, nodesystem=3;
```

`polarconicroutine` allow to specify the reference center in case of angular abscissa; the possible values are `fromCenter` and `fromFocus`.

`abscissa copy()` return a deep copy of the abscissa.

10.1. Define an abscissa

There are as many routines to define an abscissa than type of abscissa. Once an abscissa defined, one can recover the point of an object to this abscissa with the routine `point(object,abscissa)`.

- `abscissa relabscissa(real x)`
  Return the abscissa \( x \) as fraction of length of a path. It should be noted that the code `point(object,relabscissa(x))` is equivalent to `relpoint(object,x)`.

```cpp
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(\texttt{real} \ x)**

Return the curvilinear abscissa \(x\). It should be noted that the code \texttt{point(object,curabscissa(x))} is equivalent to \texttt{curpoint(object,x)}.

\[
\begin{aligned}
\text{import geometry; size(4.5cm);} \\
\text{currentcoordsys=rotate(45)*defaultcoordsys;} \\
\text{show(currentcoordsys);} \\
\text{abscissa cur=curabscissa(pi);} \\
\text{ellipse el=ellipse(origin(),2,1,-45); draw(el,linewidth(bp));} \\
\text{circle c=circle(origin(),1); draw(c,linewidth(bp));} \\
\text{dot(point(el,cur), red); dot(point(c,cur), red);} \\
\end{aligned}
\]

\[
\begin{aligned}
\text{import geometry; size(4.5cm);} \\
\text{currentcoordsys=rotate(45)*defaultcoordsys;} \\
\text{show(currentcoordsys);} \\
\text{abscissa cur=curabscissa(pi);} \\
\text{ellipse el=ellipse(origin(),2,1,-45); draw(el,linewidth(bp));} \\
\text{circle c=circle(origin(),1); draw(c,linewidth(bp));} \\
\text{dot(point(el,cur), red); dot(point(c,cur), red);} \\
\end{aligned}
\]

• **abscissa angabscissa(\texttt{real} \ x, \ \texttt{polarconicroutine polarconicroutine=currentpolarconicroutine})**

Return the angular abscissa \(x\). It should be noted that the code \texttt{point(object,angabscissa(x))} is equivalent to \texttt{angpoint(object,x)}.

\[
\begin{aligned}
\text{import geometry;} \\
\text{size(6cm);} \\
\text{abscissa x=angabscissa(30);} \\
\text{ellipse el=ellipse(origin(),2,1,45);} \\
\text{draw(el,linewidth(bp));} \\
\text{point M=point(el,x);} \\
\text{draw(M--el.F1--el.F2);} \\
\text{dot("$F_1$", el.F1, SE); dot("$F_2$", el.F2, NW);} \\
\text{markangle((string)x.x, el.F2, el.F1, M);} \\
\text{draw(tangent(el,x), linewidth(bp));} \\
\end{aligned}
\]

• **abscissa nodabscissa(\texttt{real} \ x)**

Return the abscissa \(x\) relative to the nodes of a path. It should be noted that the code \texttt{point(object,nodabscissa(x))} is equivalent to \texttt{point(object,x)}.

10.2. **Obtain an abscissa of a point**

The routines for obtaining an abscissa of a point belongs to a given object have the same name as those described in the previous section. The following routines return respectively the relative abscissa, the curvilinear abscissa, the angular abscissa and the “nodal abscissa” of the point \(M\) belongs the specified object.

**Relative abscissa**

- **abscissa relabscissa(line \ l, point \ M)**
- **abscissa relabscissa(ellipse \ el, point \ M)**
- **abscissa relabscissa(arc \ a, point \ M)**

**Curvilinear abscissa**

- **abscissa curabscissa(line \ l, point \ M)**
- **abscissa curabscissa(ellipse \ el, point \ M)**
- **abscissa curabscissa(parabola \ p, point \ M)**

**Angular abscissa**
10.3. Operators

\textbf{abscissa operator} \( + (\text{real } x, \text{explicit abscissa } a) \)

Return the copy of \( a \) with abscissa \( x + a.x \).

The following operators are also defined:

\begin{itemize}
  \item \text{operator} \( + (\text{explicit abscissa}, \text{real}) \)
  \item \text{operator} \( -(\text{real}, \text{explicit abscissa}) \)
  \item \text{operator} \( -(\text{explicit abscissa}, \text{real}) \)
  \item \text{operator} \( -(\text{explicit abscissa} a) \)
  \item \text{operator} \( \times (\text{real } x, \text{explicit abscissa } a) \)
  \item \text{operator} \( \times (\text{explicit abscissa } a, \text{real } x) \)
  \item \text{operator} \( \div (\text{real } x, \text{explicit abscissa } a) \)
  \item \text{operator} \( \div (\text{explicit abscissa } a, \text{real } x) \)
\end{itemize}

11. Triangles

11.1. Structure

The \textit{triangle} structure is more complex than the previous ones already developed in this document. Indeed it defines new types which create instances of some objects which are inextricably linked to a triangle. Moreover these objects possess the reference of the associated triangle. In other words, an object \( TR \) of type \textit{triangle} contains as a structure member the object \( VA \) of type \textit{vertex} which is attained via the code \( TR.VA \) and the object \( TR.VA \) contains the object \( t \) of type \textit{triangle} which value is \( TR \); it follows that the value of \( TR.VA.t \) is \( TR \).

Since an object of type \textit{vertex} is a triangle vertex, it is a straightforward task to write a routine that returns for instance the internal bisector of the angle of a triangle: it is sufficient to give an object of type \textit{vertex} since this object contains the reference of the triangle of which it is a vertex.

Here is a simplified version of the \textit{triangle} structure; the complete structure is detailed separately.

\begin{verbatim}
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; }
\end{verbatim}

\( A, B \) and \( C \) represent the points marking the triangle vertices;
**struct vertex** defines the vertex structure which allows to create the instance of an object which represents the vertex of a triangle. Since we need to manipulate this structure in a very few cases, it is useful to know his properties.

The property \( n \) allows to associated a vertex to a point, of type point, marking this vertex:

- if \( n = 1 \), the vertex is associated to the point \( A \);
- if \( n = 2 \), the vertex is associated to the point \( B \);
- if \( n = 3 \), the vertex is associated to the point \( C \);
- if \( n = 4 \), the vertex is associated to the point \( A \);
- etc. The value of the property \( t \) is “the object of type **triangle** to which the vertex belongs”.

For more details on the use of this structure see Triangles vertices.

VA, VB et VC represent the vertices of the triangle in an abstract way as opposed to the objects point \( A \), \( B \) and \( C \) that are the points which mark each vertex; see Section Triangles vertices.

**struct side** defines the side structure which allows to create the instance of an object which represents the side of a triangle. Since we need to manipulate this structure in a very few cases, it is useful to know his properties.

The property \( n \) allows to associated the object of type side to a triangle’s side:

- if \( n = 1 \), the side represents \( AB \) oriented from \( A \) to \( B \);
- if \( n = 2 \), the side represents \( BC \) oriented from \( B \) to \( C \);
- if \( n = 3 \), the side represents \( CA \) oriented from \( C \) to \( A \);
- if \( n = 4 \), the side represents \( AB \);
- etc.

If \( n \) is negative the orientation is reversed.

The value of the property \( t \) is “the object of type **triangle** to which the side belongs”.

Routines on **side** are described into Section Triangles sides.

AB, BC, CA, BA, AC et CB represent the sides of the triangle in a abstract way as opposed to the objects line line(TR.AB), line(TR.BC), line(TR.CA), etc, that are the lines marking the sides of the triangle TR; see Section Triangles sides.

### 11.2. Defining and drawing a triangle

This section is devoted to describe the basic routines to define and to draw a triangle. Another routines will be given as of reading.

- **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)

  Draw the labels LA, LB and LC at the vertices of the triangle t. The position depends on the internal bisector of the corresponding vertex. The parameters alignAngle and alignFactor allow to modify the direction and the length of the alignment.

- **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)

  Draw the triangle t and place the labels at the vertices of the triangle and the lengths of the sides. When coding this routine is useful to locate the vertices t.A, t.B and t.C.

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

  Draw the triangle t; sides are drawn as segments.

- **void drawline**(picture pic=currentpicture, triangle t, pen p=currentpen)

  Draw the triangle t; sides are drawn as lines.

- **triangle triangle**(point A, point B, point C)

  Return the triangle whose vertices are A, B et C.
• `triangle triangleabc(real a, real b, real c, real angle=0, point A=(0,0))`

Return the triangle ABC such that $BC = a$, $AC = b$, $AB = c$ and $\left(\vec{AB}\right) = \angle$.

```
import geometry;
size(4cm);
currentcooridnatsys=cartesiancoordinates(i=(1,0.5), j=(-0.25,.75));
show(currentcooridnatsys);
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))`

Return the triangle ABC such that $\left(\vec{AB}\right; \vec{AC}) = \alpha$, $AC = b$, $AB = c$ et $\left(\vec{\alpha}; \vec{AC} = \angle$.

```
import geometry;
size(5cm);
currentcooridnatsys=cartesiancoordinates(i=(1,0.5), j=(-0.25,.75));
show(currentcooridnatsys);
triangle t=triangleabc(-60,2,3,angle=45,(1,1));
show(Lb="2", Lc="3",t);
markangle("60\degree",t.C,t.A,t.B, Arrow);
```

• `triangle triangle(line 11, line 12, line 13)`

Return the triangle whose sides are 11, 12 and 13.

11.3. Vertices triangles

Let $t$ be an object of type `triangle`. Its properties $t.VA$, $t.VB$ and $t.VC$ are of type `vertex` and describe the vertices of the triangle $t$. The package `geometry.asy` allows routines with a vertex of triangle as parameter without specifying explicitly the triangle to which it refers.

As an example in the following code the routine `line bisector(vertex V, real angle=0)` returns the image by the rotation of center $V$ and angle $\angle$ of the internal bisector passing through $V$.

```
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));
```

Here we have some routines and basic operators on the objects of type `vertex`

• `point operator cast(vertex V)`

Allow the casting of vertex in point.
• **point point(explicit vertex V)**

Return the object of type point corresponding to the object V of type vertex. The code `point(V)` is equivalent to the code `(point)V` which forces the casting of vertex in point.

• **vector dir(vertex V)**

Return the unit vector on the internal bisector of the angle in V and oriented outwards the triangle to which V refers. This routine is especially useful to place the labels at the vertices of a triangle.

Another routines with object of type vertex as parameter are described in the following section and together with routines regarding triangles.

11.4. Sides triangles

Let t be an object of type triangle. Its properties t.AB, t.BC, t.CA, t.BA, t.AC and t.CB, of type side represents the sides of the triangle t. The package geometry.asy allows routines with a side of triangle as parameter without specifying explicitly the triangle to which it refers.

As an example in the following code the routine `circle excircle(side s)` returns the excircle of the triangle to which s refers and tangent to s.

```asy
triangle t=triangle((-1,0), (2,0), (0,2));
draw(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);
```

Here we have some routines and basic operators on the objects of type side:

• **line operator cast(side side)**

Allow the casting of side in line.

• **line line(explicit side side)**

Return the object of type line corresponding to the object side of type side. The code `line(S)` is equivalent to `(line)S` which forces the casting of side to line.

• **segment segment(explicit side side)**

Return the object of type segment corresponding to the object side of type side. The code `segment(S)` is equivalent to `(segment)S` which forces the casting of side in segment.

• **side opposite(vertex V)**

Return the opposite side to V about the triangle to which V refers.

• **vertex opposite(side side)**

Return the opposite vertex to side into the triangle to which side refers.

The another routines with object of type side as parameter are described together with routines regarding triangles.

11.5. Operators

The single operator which can be applied to triangle is `triangle operator *(transform T, triangle t)` that allows the code `transform*triangle`.

74
Another routines

- **point orthocentercenter(triangle t)**
  Return the orthocenter of the triangle t.

- **point foot(vertex V)**
  Return the foot of the altitude from V. The routine **point foot(side side)** is also available.

- **line altitude(vertex V)**
  Return the altitude from V. The routine **line altitude(side side)** is also available.

- **triangle orthic(triangle t)**
  Return the orthic triangle of t; its vertices are the feet of the altitudes of 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)**
  Return the midpoint of side.

- **point centroid(triangle t)**
  Return the centroid of the triangle t.

- **line median(vertex V)**
  Return the median from V. The routine **line median(side side)** is also available.

- **triangle medial(triangle t)**
  Return the medial triangle of t (its vertices are the midpoints of t).
• **triangle anticomplementary(triangle t)**
  Return the anticomplementary triangle of t (t is the median triangle of anticomplementary(t)).

• **line bisector(vertex V, real angle=0)**
  Return the image of the internal bisector through V under the rotation of center V and angle angle. An example has already been given.

• **point bisectorpoint(side side)**
  Return the intersection point of side with the internal bisector of the opposite angle to side.

• **line bisector(side side)**
  Return the perpendicular bisector of the segment corresponding to side.

• **point circumcenter(triangle t)**
  Return the circumcenter of the triangle t.

• **circle circle(triangle t)**
  Return the circumcircle of the triangle t. The routine circumcircle(triangle t) is an alias.

• **triangle tangential(triangle t)**
  Return the tangential triangle of t, that is the triangle whose sides are tangent to the circumcircle of t.
- **point incenter(triangle t)**
  Return the incenter of t, that is the center of the incircle of t.

- **real inradius(triangle t)**
  Return the radius of the incircle of t (also known as the inradius).

- **circle incircle(triangle t)**
  Return the incircle of t.

- **triangle intouch(triangle t)**
  Return the contact triangle or intouch triangle that is the triangle whose vertices are the three points where the incircle touches the triangle t.

- **point intouch(side side)**
  Return the contact point of the side side with the incircle to which side refers.

- **point gergonne(triangle t)**
  Return the Gergonne point of the triangle t.

- **point excenter(side side)**
  Return the center of the excircle of the triangle to which side refers and tangent to side.

- **real exradius(side side)**
  Return the radius of the excircle of the triangle to which side refers and tangent to side.
- **circle excircle(side side)**
  Return the excircle of the triangle to which side refers and tangent to side.

- **triangle extouch(triangle t)**
  Return the extouch triangle of t that is the triangle whose vertices are the contact points of the three excircles to t with its sides.

- **point extouch(side side)**
  Return the contact point of the side side with the excircle given by excircle(side).

- **point symmedian(triangle t)**
  Return the symmedian point (also called the LEMOINE point) of the triangle t.

- **point symmedian(side side)**
  Return the symmedian point of the side side.

- **line symmedian(vertex V)**
  Return the symmedian line of the triangle to which V refers and passing through V.

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

triangle t=triangle((-1,0), (2,0), (0,2));
drawline(t, linewidth(bp));
label(t,alignFactor=4);

circle cl=excircle(t.AB), c2=excircle(t.BC);
circle c3=excircle(t.AC);
clipdraw(cl, 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 cevian(side side, point P)**
  Return the CEVIAN point of P belonging to the side side.

• **triangle cevian(triangle t, point P)**
  Return the CEVIAN triangle of P.

• **line cevian(vertex V, point P)**
  Return the CEVIAN line of P passing through V in the triangle to which V refers.

The following example is an illustration of the property “if the triangle $A'B'C'$ is a CEVIAN triangle of the triangle $ABC$ then the triangle $A''B''C''$ whose vertices are the symmetric of $A'$, $B'$ et $C'$ about the respective midpoints is also a CEVIAN triangle”.

```python
import geometry; size(10cm,0);
triangle t=triangle((-1,0), (2,0), (0,2)); draw(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));
```
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);

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)**
  Return the isotomic line through V and relative to M about the triangle which V refers.

- **point isotomicconjugate(triangle t, point M)**
  Return the isotomic conjugate of M relatively to t.

- **point isotomic(side side, point M)**
  Return the intersection point of the isotomic line of M, about the triangle which side refers, with the side side.

- **triangle isotomic(triangle t, point M)**
  Return the triangle whose the vertices are the intersection points of the isotomic lines relative to M about t with the sides of t. So, in the previous picture, the triangle $A''B''C''$ is the isotomic triangle relative to P.

Below, the same figure obtained using routines isotomic gains in brevity.
\textbf{point \texttt{isogonalconjugate}(triangle \texttt{t}, point \texttt{M})}

Return the isogonal conjugate of \texttt{M} about \texttt{t}.

\textbf{point \texttt{isogonal}(side side, point \texttt{M})}

Return the intersection point of the isogonal line of \texttt{M}, according to the triangle which \texttt{side} refers, with the side \texttt{side}.

\textbf{triangle \texttt{isogonal}(triangle \texttt{t}, point \texttt{M})}

Return the triangle whose the vertices are the intersection points with the sides of \texttt{t} of the isogonal lines relative to \texttt{M} about \texttt{t}.

The following example illustrates the property "the pedal triangles of two isogonal points \texttt{P} and \texttt{Q} are inscribed in the same circle centered at the midpoint of the segment line [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(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);

• triangle pedal(triangle t, point M)
  Return the pedal triangle of M about t.

• line pedal(side side, point M)
  Return the line through M and through the foot of the perpendicular from M to the side line side.
  The following example shows a few lines of SIMSON; note the use of methods t.side(int) and t.vertex(int) that allows to obtain by theirs numbers the sides and the vertices of the triangle t.
11.7. Trilinear coordinates

The type `trilinear`, whose the structure is given further, allows to instantiate an object representing the trilinear coordinates \(a:b:c\) with respect to the triangle \(t\).
struct trilinear
{
    real a,b,c;
    triangle t;
}

To define the trilinear coordinates \(a:b:c\) with respect to a triangle \(t\) one can use the routine
\[
\text{trilinear(}t, \text{real } a, \text{real } b, \text{real } c)\]
It is also possible to obtain the trilinear coordinates of a point through the routine
\[
\text{trilinear(}t, \text{point } M)\]
It is also possible to define trilinear coordinated through a \text{triangle center function} \(f\) and three parameters \(a, b\) and \(c\) using the following routine:
\[
\text{trilinear(}t, f, \text{real } a=t.a(), \text{real } b=t.b(), \text{real } c=t.c())\]
where the type \text{centerfunction} represents a real function with three real variables.

The casting of an object of type \text{trilinear} to type \text{point} may be done, as usual, through two ways: through the routine
\[
\text{point(}\text{trilinear} t)\]
or through syntax casting \text{point(}trilinear t\text{)}.

For example, using the trilinear coordinates of the isotomic conjugate of a point, here’s how it is defined the routine \text{isotomicconjugate}:
\[
\text{point isotomicconjugate(}t, M)\]
\[
\{
    \text{trilinear } tr=\text{trilinear(}t,M)\;
    \text{return point(}\text{trilinear(}t,1/(t.a()^2*tr.a),1/(t.b()^2*tr.b),1/(t.c()^2*tr.c)));\}
\]

12. Inversions

The type \text{inversion}, whose the structure is given below, allows to instantiate an inversion with center \(C\) and radius \(k\)

\[
\text{struct inversion}
{
    point C;
    real k;
}
\]

12.1. Define an inversion

The following routines and operators allow to define an inversion.

- \text{inversion inversion(real } k, \text{point } C)\]
  Return the inversion with center \(C\) and radius \(k\). The routine \text{inversion(point } C, \text{real } k)\] is also available.

- \text{inversion inversion(circle } c1, \text{circle } c2, \text{real } sgn=1)\]
  - if \(sgn\) is non-zero, this routine returns the inversion whose the radius is the same sign as \(sgn\) and transforming \(c1\) to \(c2\);
  - if \(sgn\) is zero, this routine returns the inversion centered at the foot of the radical axis and leaving globally invariants each of the two circles \(c1\) and \(c2\).

An example using this routine has already been given.

- \text{inversion inversion(circle } c1, \text{circle } c2, \text{circle } c3)\]
  Return the inversion leaving globally invariants the three circles \(c1, c2\) et \(c3\).

- \text{circle operator cast(inversion } i)\]
  Allow the casting of an object of type \text{inversion} to \text{circle}. The returned circle is the principal circle of \(i\).
  It may also force the casting through the routine \text{circle circle(inversion } i)\).

- \text{ inversion operator cast(circle } c)\]
  Allow the casting of a \text{circle} to an \text{inversion}. The returned inversion leaves globally invariant \(c\), with center the center of \(c\) and the sign of the radius is the same as the radius of \(c\).
  It may also force the casting through the routine \text{inversion inversion(circle } c)\).
12.2. Apply an inversion

The following operators allow the code `inversion*object` which return the image of the object `object` under inversion.

- **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)*

It should be noted that the inverse of a circle or a line may be a line. In this case the returned circle C has an infinite radius and the property C.l, whose type is `line`, is set to the correct value; the routines admitting this circle as a parameter use C.l in place as showed by the following example.

```plaintext
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);
```

12.3. Examples

Examples using inversions have already been given, here are some others.

We begin by illustrating the use of a circle, here the circle inscribed in a triangle, as an inversion:

```plaintext
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);
```

Below the construction of the PAPPUS chain.
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);

One can easily get a good representation:

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))));

The following picture, where the lines are not parallel, is a variant:
The following figure is the image of a checkerboard $12 \times 12$ by the inversion whose center is close to the center of the checkerboard and with radius 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

Index

*(explicit point, explicit pair), 9
*(inversion, circle), 85
*(inversion, line), 85
*(inversion, point), 85
*(inversion, segment), 65, 85
*(inversion, triangle), 85
*(point, line), 27
*(real, explicit arc), 63
*(real, explicit circle), 40
*(real, line), 27
*(transform, conic), 36
*(transform, explicit arc), 62
*(transform, explicit point), 9
*(transform, line), 27
*(transform, triangle), 74
+(conic, explicit pair), 36
+(conic, explicit point), 36
+(conic, explicit vector), 36
+(explicit abscissa, real), 71
+(explicit arc, point), 65
+(explicit arc, vector), 65
+(line, vector), 27
+(real x, explicit abscissa a), 71
-(conic, explicit pair), 36
-(conic, explicit point), 36
-(conic, explicit vector), 36
-(explicit abscissa, real), 71
-(explicit arc, point), 65
-(explicit arc, vector), 65
-(line, vector), 27
-(real, explicit abscissa), 71
/(explicit abscissa a, real x), 71
/(explicit arc, real), 63
/(explicit circle, real), 40
/(line, real), 27
/(real x, explicit abscissa a), 71
==(explicit point, explicit point), 9
==(line, line), 28
équation
  conique, 36
^ (point M, explicit circle c), 40
hline, line), 28

abs(coordsys, pair), 9
abs(explicit point), 9
abscissa, 4
abscisse, 69
addpenarc, 60
altitude(side), 75
altitude(vertex), 75
angabscissa(circle, point), 71
angabscissa(ellipse, point, polarconicroutine), 71
angabscissa(hyperbola, point, polarconicroutine), 71
angabscissa(parabola, point), 71
angabscissa(real, polarconicroutine), 70
angle(arc), 65
angle(explicit point, coordsys, bool), 9
angle(line, coordsys), 27
angle(line,line), 27
angpoint(explicit circle,real), 42
angpoint(explicit ellipse,real), 51
angpoint(explicit hyperbola,real,polarconicroutine), 58
angpoint(explicit parabola,real), 54
angularsystem, 69
anticomplementary(triangle), 76
antipedal(triangle,point), 83
arc, 4, 60
arc(arc,abscissa,abscissa), 68
arc(ellipse,abscissa,abscissa,bool), 68
arc(ellipse,point,point,bool), 67
arc(ellipse,real,real,polarconicroutine,bool), 60
arc(explicit arc,point,point), 68
arccircle(point,point,point), 67
arccircle(point,point,real,bool), 67
arccircle(hyperbola,real,real,int,bool), 59
arccircle(parabola,real,real,int, bool), 58
arclength(arc a), 65, 66
arclength(ellipse,real,real,bool,polarconicroutine), 49
arcnodesnumber(explicit arc), 62
arcsubtended(point,point,real), 66
arcsubtendedcenter(point,point,real), 66
bangles(picture,hyperbola), 57
bangles(picture,parabola), 53
bisector(line,line,real,bool), 24
bisector(segment), 31
bisector(side side), 76
bisector(vertex,real), 73, 76
bisectorpoint(side side), 76
bq(e(coordsys,real,real,real,real,real,real), 36
bq(e(point,point,point,point,point), 37
byfoci, 55
byvertices, 55
canonical(bq(e), 37
canonicalcartesiansystem(ellipse), 35
canonicalcartesiansystem(explicit conic), 35
canonicalcartesiansystem(hyperbola), 35
canonicalcartesiansystem(parabola), 35
cast(circle), 84
cast(inversion), 84
cast(side side), 74
cast(vertex V), 73
casting
conique, 37
centerToFocus(ellipse,real), 49
centroid(triangle), 75
circle, 38
cevian(side.point), 79
cevian(triangle,point), 79
cevian(vertex,point), 79
changecoordsys(coordsys,bq(e), 37
changecoordsys(coordsys,conic), 35
changecoordsys(coordsys,line), 28
circle to ellipse, 40
circle(explicit point,real), 38
circle(inversion), 84
circle(point A, point B), 39
ellipsenodesnumberfactor, 62
equation(ellispe), 37
equation(explicit conic), 37
equation(parabola), 37
excenter(side), 77
excircle(point,point,point), 39
excircle(side), 74, 78
extradius(side), 77
extend(line l), 28
extouch(side), 78
extouch(triangle), 78
finite(explicit point), 9
focusToCenter(ellipse,real), 49
foot(side), 75
foot(vertex), 75
gergonne(triangle t), 77
hprojection(line,bool), 32
hyperbola, 37
hyperbola(point,point,real,bool), 55
hyperbola(point,real,real,real), 56
hyperbolanodesnumber(hyperbola,real,real), 57
hyperbolanodesnumberfactor, 57
incenter(triangle), 77
incircle(point,point,point), 39
incircle(triangle), 77
inradius(triangle), 77
intersectionpoint(line,line), 22
intersectionpoints(arc,arc), 66
intersectionpoints(arc,conic), 66
intersectionpoints(conic,arc), 66
intersectionpoints(conic,conic), 35
intersectionpoints(conic,line), 36
intersectionpoints(conic,triangle,bool), 36
intersectionpoints(line l, path g), 23
intersectionpoints(line,arc), 66
intersectionpoints(line,conic), 36
intersectionpoints(triangle,conic,bool), 36
intouch(side), 77
intouch(triangle), 77
inverse(real,point,segment), 69
inversion, 43, 84
inversion(circle), 84
inversion(circle,circle,circle), 84
inversion(circle,circle,real), 84
inversion(point,real), 84
inversion(real,point), 84
isogonal(side,point), 81
isogonal(triangle,point), 81
isogonalconjugate(triangle,point), 81
isotomic(side,point), 80
isotomic(triangle,point), 80
isotomic(vertex,point), 80
isotomicconjugate(triangle,point), 80
label(picture,Label,Label,Label,triangle,real,real,pen,filltype), 72
length(explicit point), 9
line, 3, 22
line perpendicular(point,explicit vector), 26
line(coordsys,real,real), 23
line(coordsys,real,real,real), 23
line(explicit side side), 74
line(point,bool,point,bool), 22
line(point,real), 24
line(real,point), 24
locate, 7
locate(point), 9
locate(vector), 10
markangle(picture,Label,int,real,real,line,line,arrowbar,pen,margin,marker), 29
markarc, 60
mass, 3, 13, 13
mass mass(explicit point), 13
mass(coordsys,explicit pair,real), 13
mass(explicit point), 13
mass(point,real), 13
masscenter([...mass[]]), 13
massformat(string,string,mass), 14
medial(triangle), 75
median(side), 75
median(vertex), 75
midpoint(segment), 31
midpoint(side), 75
nodabscissa(ellipse,point), 71
nodabscissa(line,point), 71
nodabscissa(parabola,point), 71
nodabscissa(real), 70
nodesystem, 69
(point,arc), 65
operator *(explicit abscissa a, real x), 71
operator *(real x, explicit abscissa a), 71
operator -(explicit abscissa a), 71
opposite(side), 74
opposite(vertex), 74
origin, 8
origin(coordsys), 8
orthic(triangle), 75
orthocentercenter(triangle), 75
Ox, 22
Ox(), 22
Oy, 22
Oy(), 22
parabola(point,line), 52
parabola(point,point), 52
parabola(point,point,point,line), 52
parabola(point,real,real), 52
parabolanodesnumber(parabola,real,real), 53
parabolanodesnumberfactor, 53
parabole, 52
parallel(line,line,bool), 23
parallel(point,explicit pair), 23
parallel(point,explicit vector), 23
parallel(point,line), 23
pedal(side side, point M), 82
pedal(triangle,point), 82
perpendicular(line,line), 29
perpendicular(point,explicit pair), 26
perpendicular(point,line), 26
perpendicularmark(picture,line,line,real,pen,int,margin,filltype), 30
point, 3, 6
*, 8
+, 8
-, 8
/, 8
casting, 6
casting, 6
changoordsys, 7
point point(explicit mass), 13
point(point(coordsys,explicit point,real), 8
point(point(coordsys,pair,real), 7
point(explicit circle,real), 41
point(explicit ellipse,real), 50
point(explicit hyperbola,real), 58
point(explicit mass), 13
point(explicit parabola,real), 54
point(explicit vector), 11
point(explicit vertex V), 74
point(line l,real), 29
polar, see coordsys polar
polarconicroutine, 49
polarconicroutine(conic), 60
projection(line), 32
projection(point,point), 17
projection(point,point,point,point,bool), 18
radicalcenter(circle,circle), 41
radicalcenter(circle,circle,circle), 41
radicalline(circle,circle), 41
real dot(point,explicit pair), 9
reflect(line), 31
reflect(line,line,bool), 31
relabscissa(arc,point), 70
relabscissa(ellipse,point), 70
relabscissa(line,point), 70
relabscissa(real), 69
relativesystem, 69
relpoint(explicit circle,real), 41
relpoint(explicit ellipse,real), 50
relpoint(explicit hyperbola,real), 58
relpoint(explicit parabola,real), 54
relpoint(line,real), 29
reverse(arc), 65
reverse(line), 28
rotateO(real), 21
samecoordsys(bool...point[]), 8
sameside(point,line,line), 28
sameside(point,point,line), 28
scale(real,line,line,bool), 31
scale(real,point), 17
scale(real,point,point,point,point,bool), 18
scaleO(real), 21
sector(int,int,line,line,real,bool), 25
segment, 3
segment(explicit side side), 74
segment(point,point), 31
sharpangle(line,line), 27
sharpdegrees(line,line), 27
show(picture,Label,Label,Label,Label,Label,Label,triangle,pen,filltype), 72
show(picture,line,pen), 22
side, 4
triangle, 74
standardizecoordsys(coordsys,bool...M), 9
symmedian(side side), 78
symmedian(triangle), 78
symmedian(vertex V), 78

tangent(circle, abscissa), 41
tangent(ellipse, abscissa), 49
tangent(explicit arc, abscissa), 69
tangent(explicit arc, point), 69
tangent(hyperbola, abscissa), 58
tangent(parabola, abscissa), 54
tangential(triangle), 76
tangents(circle c, point M), 41
tangents(ellipse, point), 49
tangents(hyperbola, point), 57
tangents(parabola, point), 54

transformation, 16, 31
triangle, 4, 71
triangle(line, line, line), 73
triangle(point, point, point), 72
triangleAbc(real, real, real, real, point), 73
triangleabc(real, real, real, real, point), 73

trilinear, 4

coordinates, 83
trilinear coordinates

triangle, 83
trilinear(triangle, center function, real, real, real), 84
trilinear(triangle, point), 84
trilinear(triangle, real, real, real), 84

unit(point), 12
unit(vector), 12

vector, 3, 10
vector vector(point), 12
vertex

triangle, 73
vertex . 4

void label(picture, Label, mass, align, string, pen, fill type), 14
vprojection(line, bool), 32

xscale(real, point), 19
xscaleO(real), 21

yscale(real, point), 20
yscaleO(real), 21