Claudio Beccari

claudio dot beccari at gmail dot com

The euclideangeometry package

user manual

Version 0.2.2 of 2023-07-11

Abstract

The euclideangeometry package further extends the functionalities of the curve2e package, which, on turn, is an extension of the pict2e package for the standard ***picture*** environment as defined in the LaTeX kernel source file.

The curve2e package was upgraded in 2020 and again in 2023; some material of this package, might have been included in the former pict2e one, but it is so specific, that we preferred defining a standalone one; this package takes care of requesting the packages it depends from.

The purpose is to provide the tools to draw most of the geometrical constructions that a high school teacher or bachelor degree professor might need in order to teach (plane) geometry. The connection to Euclide depends on the fact that in its times calculations were made with ruler, compass, and, apparently, also with the ellipsograph.

The user of this package has available all the machinery provided by the pict2e and curve2e packages, in order to define new functionalities and build macros that draw the necessary lines, circles, and other such objects, as they would have done in the old times. Actually just one macro is defined in this package to solve a linear system of equations

Contents

Warning

The euclideangeometry package requires the advanced functionalities of the LaTeX 3 (L3) language; if such functionalities are not available for any reason (incomplete/basic installation of the TeX system; legacy installation of the TeX system; the TeX system has not been updated; ...) input of this package is stopped, the whole job is aborted, and a visible message is issued.

1 Introduction

The picture environment has been available since the very beginning of LaTeX in 1985. At that time it was really a simple environment that allowed to draw very simple line graphics with many limitations. When LaTeX was upgraded from LaTeX 2.09 to LaTeX 2_ε in 1994, Leslie Lamport announced an upgrade that eventually became available in 2003 with package pict2e; in 2006 I wrote the first version of the curve2e package that added many more functionalities; both packages were upgraded during these years; and now line graphics with the ***picture*** environment can perform pretty well. The package euclideangeometry adds even more specific functionalities in order to produce geometric drawings as they were possible in the old times, when calculus and analytic geometry were not available.

In these years other drawing programs were made available to the TeX community; PSTricks and TikZ are the most known ones, but there are other less known packages, that perform very well; among the latter I would like to mention xpicture, that relies on pict2e and curve2e, but extends the functionalities with a very smart handling of coordinate systems, that allow to draw many line drawings suitable for teaching geometry in high schools and introductory courses in the university bachelor degree programs. It is worth mentioning that an extension of TikZ, called tkz-euclide, is also available in a complete and updated TeX system installation; at the moment its documentation needs some refinements, at least to consistently use a single language, without switching from English to French and viceversa. It aims at the same readership, but it allows to do many more geometrical constructions, than euclideangeometry. The real difference is that euclideangemetry may be easily expanded without the need of knowing the complex machinery and coding of the tkz-euclide underlaying TikZ package.

This package euclideangeomery apparently follows the same path of xpicture, but it avoids defining a new user language interface; rather it builds new macros by using the same philosophy of the recent curve2e package.

It is worth mentioning that now curve2e accepts coordinates in both cartesian and polar form; it allows to identify specific points of the drawing with macros, so the same macro can be used over and over again to address the same point. The package can draw lines, vectors, arcs with no arrow tips, or with one arrow tip, or with arrow tips at both ends, arcs included. The macros for drawing polylines, polygons, circles, generic curves (by means of Bézier cubic or quadratic splines) are already available; such facilities are documented and exemplified in the user manual of the curve2e package.

In what follows there will be several figures drawn with this package; in the background there is a red grid where the meshes are 10 `\unitlenth` apart in both directions; they should help to understand the position of the various drawings on the picture canvas. This grid is useful also to the end user, while s/he is working on a particular drawing, but when the drawing is finished, the user can delete the grid command or comment out that line of code. For what regards the commands used to render the images, their *codes* can be found in the documented code file euclideangeometry.pdf.

2 Installing euclideangeometry

You are not supposed to manually install package euclideangeometry. In facts you have to work with a complete and updated/upgraded TeX installation, otherwise this package won't work; this means that you have done your updating after 2020-01-18. And this package is already present in any modern updated complete installation of the TeX system. If curve2e has a date earlier then 2020-01-018, the curve2e itself will load curve2ev161, an older version, and this package euclideangeomentry will abort its own loading, besides aborting the whole job.

We remember the package dependencies; the primary dependence is package curve2e with a version date more recent or equal to 2020-01-18. On turn curve2e requires packages xparse and xfp[1]; missing these two package, it loads its own previous version, that does not use such packages, but their absence forbids it working, so that, after a very visible error message, it directly aborts. It also depends on etoolbox. The chain of dependencies of the above first level packages may be controlled directly on those packages documentation

3 Loading euclideangeometry

If you want to use the euclideangeometry package, we suggest you load it with the following command:

`\usepackage[⟨options⟩]{euclideangeomery}`

The package will take care of managing the possible ⟨*options*⟩ and to call curve2e with such specified options; on turn curve2e calls pict2e passing on the ⟨*options*⟩; such ⟨*options*⟩ are only those usable by pict2e because neither curve2e nor euclideangeometry use any option. If the user is invoking euclideangeometry, it is certain s/he does not want to use the modern extended picture environment, not the native one; therefore the only meaningful possible options are *latex* and *pstricks*; such options influence only the shape of the arrow tips; with option *latex* they are triangular, while with *pstricks* they have the shape of a stealth aircraft. The difference is very small; therefore we imagine that even if these options are available, they might never be used.

Nothing happens if the user forgets this mechanism; therefore if s/he loads curve2e and/or pict2e, before euclideangeomentry the only problem that might arise is an "Option clash" error message; if two of these packages are selected with different arrow tips; not impossible, of course, by we deem it very unlikely.

---

[1] Most functionalities of xfp are already included into the LaTeX 2$_\varepsilon$ kernel, but this package uses also some functionalities that have not made their way to the kernel.

4 Available commands

The commands available with the first extension `pict2e` to the native ***picture*** environment, maintain their names but do not maintain the same restrictions; in particular there are the following improvements.

1. Lines and vectors are drawn as usual by `\put`ting in place their forms, but their inclinations are not limited to a small set of slope parameters, originally specified with reciprocally prime single digit values not exceeding 6 for lines, and 4 for vectors; the length of these sloped objects is still their horizontal component; now, the slopes may be described with any signed fractional number not exceeding $2^{30} - 1$ in absolute value; it still is a limited number of slopes, but their combinations are practically countless.
2. There is no restriction on the minimum length of lines and vectors.
3. Circles and dots can be drawn at any size, not at that dozen or so finite sizes that were accepted with the original environment.
4. Ovals may be specified the corner curvature; the default size of the quarter circles that make up the oval corners may be specified; if no specification is given the radius of such corners is the maximum that can fit into the oval; in practice it is half the shortest value between the oval height and width.
5. The quadratic Bézier splines do not require the specification of the number of dots that were used by the native environment to draw "arbitrary" curves; now they are drawn with continuous curved lines.

Some new commands were added by `pict2e`

1. The third degree (cubic) Bézier splines are sort of new; certainly now they are traced with continuous lines; if it is desired, it is possible to replace the continuous line with a number of dots so as to have a (unevenly) dotted curve. It suffices to specify the number of dots the curve should be made with.
2. `\arc` and `\arc*` draw an arc or a filled circular sector, with their centers at the axes origin; therefore they need to be put in place somewhere else by means of the usual `\put` command.
3. The new command `\Line` traces a segment from one given point to another point; it is very convenient to specify the end points instead of the slope the line must have to go from the starting to the ending point. The command does not require the `\put` command to put the segment in place; nevertheless it can be shifted somewhere else with `\put` if it becomes necessary.
4. The new command `\polyline` draws a sequence of connected segments that form a piecewise linear "curve"; the way segments are joined to one another depend from the "join" specifiers that `pict2e` has introduced; they will be described further on.

5. `\polygon` and `\polygon*` produce closed paths as it would be possible when using `\polyline` and specifying its last point coincident with its first one of that curve. If the asterisk is used the closed path is filled with the default color.

There were also the low level commands user interfaces to the various drivers; these drivers really exist, but pict2e knows how to detect the correct language of the necessary driver; the user is therefore allowed to pretend to ignore the existence of such drivers, and s/he can simply use these low level commands; their names are almost self explanatory.

1. `\moveto` Sets the start of a line to an initial point.
2. `\lineto` traces a segment up to a specified point.
3. `\curveto` traces a third degree Bézier spline up to the third specified point, while using the other two ones as control points.[2]
4. `\circlearc` traces a circumference arc from the last line point to a specified destination; its center, its angle amplitude, its initial point are among the specified arguments, but the reader should check on the pict2e documentation for the details.
   Warning! Notice that these commands produce just information to trace lines, but by themselves they do not trace anything; in order to actually trace the curve or do other operations with what has been done after the user finished describing the line to be traced, the following low level commands must be used.
5. A `\closepath` is necessary if it is desired to join the last position to the initial one. But if the last point specified coincides with the very first one, a closed loop is effectively already completed.
6. If a `\strokepath` command is used the line is drawn.
7. If a `\fillpath` command is used, the line loop is filled with the current color. Notice, if the described line is not a closed loop, this filling command acts as if the line first and last points were joined by a straight line.

While describing a line with the above low level commands, or with the previous high level commands, lines and segments join and finish as described hereafter; the following commands must be used before actually tracing a specific line made up with several joined lines or curves. Notice that their effect is just visible with lines as thin as 1 pt, and very visible with thicker lines.

1. `\buttcap` truncates each line with a sharp cut perpendicular to the line axis exactly through the line end point (default).
2. `\roundcap` adds a semicircle to the very end of each line.
3. `\squarecap` adds a half square to the very end of each line.

---

[2] If these terms are unfamiliar, please, read the pict2e documentation.

4. `\miterjoin` joins two (generally straight) lines with a miter (or mitre) joint; this means that the borders of the line are prolonged until they meet; it is very nice when the junction angle is not far away from, or is larger than 90°. Apparently for pict2e this type of joint is the default.
5. `\roundjoin` joins each (generally straight) line with an arc on the external part of the bend; it is good in most circumstances.
6. `\beveljoin` joins two (generally straight) lines with a miter joint truncated with a sharp cut perpendicular to the bisector of the lines axes; with acute angles it is better than the miter joint, but when angles are very small, even this joint is not adequate.

Notice that `\buttcap` is the default, but in general it might be better to declare the `\roundcap` for the whole document.

We do not go further in the description of the new pict2e modified and new commands; the reader unfamiliar with programmable drawing and the pic2e extensions can consult that package documentation. Actually all commands have been redefined or modified by curve2e in order to render them at least compatible with both the cartesian and polar coordinates. In oder to have a better understanding of these details, see figure 1[3].

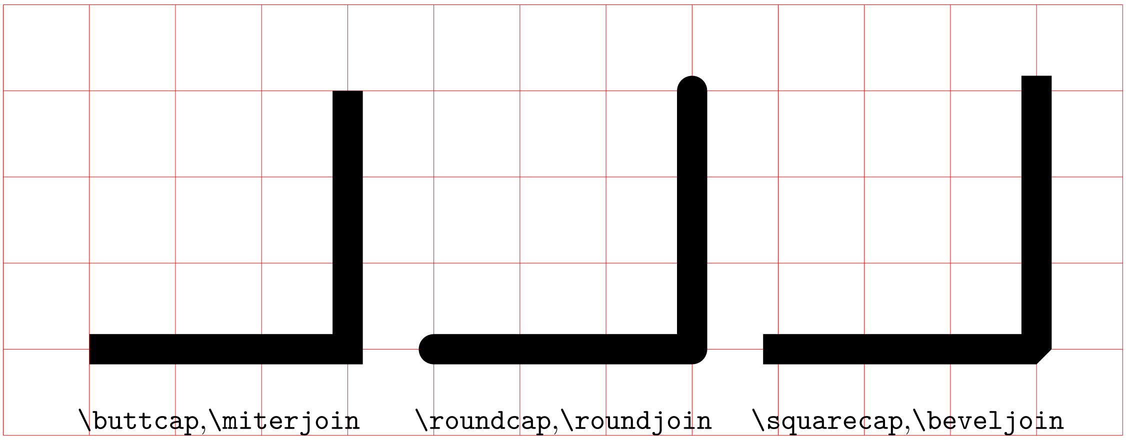


Figure 1: Different caps and joins

5 curve2e extensions

Again we do not enter into the details, because the user can read the new user manual `curve2e-manual.pdf` simply by entering and executing the `texdoc curve2e-manual` command into a terminal or command prompt window; this new manual is available with version 2.2.0 (or higher) of curve2e

---

[3] The `\polyline` macro has the default join of type bevel; remember to specify a different join type if you want a different one.

and it contains the extensions and sample codes for (simple) sample drawings; some examples are not so simple, but show the power of this package upgrade.

The most important two changes are (a) the choice of different coordinates for addressing points on the drawing canvas, and (b) the possibility of using macros to identify specific points. As already mentioned, such changes have been applied also to most, if not all[4] commands defined by pict2e.

curve2e defines a lot of operations the user can do with the point coordinates; this is done by assuming they are complex numbers, or vectors, or rotoamplification operators, and making with such entities a lot of actions compatible with their “incarnation". For example multiplying a vector by a rotoàmplification operator, in spite the fact that internally they are both represented by ordered pairs of (generally) fractional numbers, means simply obtaining a new vector rotated and scaled with respect to the original one; the point addressed by the first vector, becomes another point in a different precise position.

Below you see several examples of usage of such commands; but here space will be saved if a short list is made concerning these “complex number” operations.

Remember the double nature of such complex numbers:

$$z = x + \mathrm{i}y = m\,\mathrm{e}^{\mathrm{i}\varphi}$$

therefore addition and subtraction are simply done with

$$z_1 \pm z_2 = x_1 \pm x_2 + \mathrm{i}(y_1 \pm y_2)$$

Multiplications and divisions are simply done with

$$\begin{aligned} z_1 z_2 &= (m_1 m_2)\,\mathrm{e}^{\mathrm{i}(\varphi_1+\varphi_2)}\\ z_1/z_2 &= (m_1/m_2)\,\mathrm{e}^{\mathrm{i}(\varphi_1-\varphi_2)} \end{aligned}$$

Squares and square roots[5] are simply done with:

$$\begin{aligned} z^2 &= m^2\,\mathrm{e}^{\mathrm{i}2\varphi}\\ \sqrt{z} &= \sqrt{m}\,\mathrm{e}^{\mathrm{i}\varphi/2} \end{aligned}$$

The complex conjugate of a complex number is shown with a superscript asterisk:

$$\text{if } z = x + \mathrm{i}y \text{ then } z^\star = x - \mathrm{i}y$$

---

[4] I assume I have upgraded all such commands; if not, please, send me a bug notice; I will acknowledge your contribution.

[5] The square root of a complex number has two complex values; here we do not go into the details on how curve2e choses which value. In practice, the curve2e macros that use square roots, work mostly on scalars to find magnitudes that are always positive.

and from these simple formal rules many results can be obtained; and therefore several macros have been defined.

But let us summarise. Here is a short list with a minimum of explanation of the commands functionalities introduced by curve2e. The user notices that many commands rely on a delimited argument command syntax; the first arguments can generally be introduced with point macros, as well as numerical coordinates (no matter if cartesian or polar ones) while the output(s) should always be in form of point macro(s). Parentheses for delimiting the ordered pairs or the point macros are seldom required. On the other side, the variety of multiple optional arguments, sometimes requires the use of different delimiters, most often than not the signs `< >`, in addition to the usual brackets. These syntax functionalities are available with the xparse and xfp packages, that render the language L3 very useful and effective.

Handling of complex numbers is done with the following commands. New commands to draw special objects, are also described.

1. Cartesian and polar coordinates; they are distinguished by their separator; cartesian coordinates are the usual comma separated pair $\langle x, y\rangle$; polar coordinates are specified with a colon separated pair $\langle\vartheta\colon\varrho\rangle$. In general they are specified within parentheses, but some commands require them without any parenthesis. In what follows a generic math symbol, such as for example P_1, is used to indicate a complex number that addresses a particular point, irrespective of the chosen coordinate type, or a macro defined to contain those coordinates.
2. The complex number/vector operations already available with curve2e are the following; we specify "macro" because in general macros are used, instead of explicit numerical values, but for input vector macros it is possible to use the comma or colon separated ordered pair; "versor" means "unit vector"; angles are always expressed in degrees; output quantities are everything that follows the key word `to`; output quantities are always supposed to be in the form of control sequences.
    - `\MakeVectorFrom`⟨*number,number*⟩`to`⟨*vector macro*⟩
    - `\CopyVect`⟨*vector macro*⟩ `to`⟨*vector macro*⟩
    - `\ModOfVect`⟨*vector macro*⟩ `to`⟨*modulus macro*⟩
    - `\DirOfVect`⟨*vector macro*⟩ `to`⟨*versor macro*⟩
    - `\ModAndDirOfVect`⟨*vector macro*⟩ `to`⟨*modulus macro*⟩ `and`⟨*versor macro*⟩
    - `\ModAndAngleOfVect`⟨*vector macro*⟩ `to`⟨*modulus macro*⟩ `and`⟨*angle macro*⟩
    - `\DistanceAndDirOfVect`⟨P_1⟩ `minus`⟨P_2⟩ `to`⟨*distance macro*⟩ `and`⟨*versor macro*⟩
    - `\XpartOfVect`⟨*vector macro*⟩ `to`⟨*numerical macro*⟩
    - `\YpartOfVect`⟨*vector macro*⟩ `to`⟨*numerical macro*⟩
    - `\DirFromAngle`⟨*angle macro*⟩ `to`⟨*versor macro*⟩
    - `\ArgOfVect`⟨*vector macro*⟩ `to`⟨*angle macro*⟩

   - `\ScaleVect`⟨*vector macro*⟩ `by`⟨*scale factor*⟩ `to`⟨*vector macro*⟩
   - `\ConjVect`⟨*vector macro*⟩ `to`⟨*conjugate vector macro*⟩
   - `\SubVect`⟨*subtrahend vector*⟩ `from`⟨*minuend vector*⟩ `to`⟨*vector macro*⟩
   - `\AddVect`⟨*1st vector*⟩ `and`⟨*2nd vector*⟩ `to`⟨*vector macro*⟩
   - `\Multvect{`⟨*1st vector*⟩`}`⟨$\star$⟩`{`⟨*2nd vector*⟩`}`⟨$\star$⟩⟨*output vector macro*⟩
     the asterisks are optional; either one changes the ⟨*2nd vector*⟩ into its complex conjugate
   - `\MultVect`⟨*1st vector*⟩⟨$\star$⟩⟨*2nd vector*⟩ `to`⟨*vector macro*⟩
     discouraged; maintained for backward compatibility; the only optional asterisk changes the ⟨*2nd vector*⟩ into its complex conjugate
   - `\Divvect{`⟨*dividend vector*⟩`}{`⟨*divisor vector*⟩`}{`⟨*output vector macro*⟩`}`
   - `\DivVect`⟨*dividend vector*⟩ `by`⟨*divisor vector*⟩ `to`⟨*vector macro*⟩
     maintained for backwards compatibility.

3. A new command `\segment(`⟨P_1⟩`)(`⟨P_2⟩`)` draws a line that joins the specified points.
4. Command `\Dashline(`⟨P_1⟩`)(`⟨P_2⟩`){`⟨*dash and gap length*⟩`}` draws a dashed line between the specified points; the ⟨*dash length*⟩ is specified as a coefficient of `\unitlenth` so it is proportioned to the diagram scale. The gap between dashes is just as wide as the dashes; they are recomputed by the command in order to slightly adjust the ⟨*dash and gap length*⟩ so that the line starts at point P_1 with a dash, and ends at P_2 again with a dash.
5. Command `\Dotline(`⟨P_1⟩`)(`⟨P_2⟩`){`⟨*gap*⟩`}[`⟨*diameter*⟩`]` traces a dotted line between the specified points with dots ⟨*gap*⟩ units apart, starting and ending with a dot at the specified points. Optionally the absolute diameter of the dots may be specified: a diameter of 1 pt (default) is visible, but it might be too small; a diameter of 1 mm is really very black, and may be too large; if the diameter is specified without dimensions they are assumed by default to be typographic points.
6. Command `\polyline`, `\polygon` and `\polygon*` are redefined to accept both coordinate kinds.
7. Commands `\VECTOR(`⟨P_1⟩`)(`⟨P_2⟩`)` and `\VVECTOR`, with the same syntax, draw vectors with one arrow tip at the end, or arrow tips at both ends respectively.
8. New commands `\Arc(`⟨*center*⟩`)(`⟨*start*⟩`){`⟨*angle*⟩`}` and, with the same syntax, `\VectorArc` and `\VectorARC` draw arcs with the specified ⟨*center*⟩, starting at point ⟨*start*⟩, with an aperture of ⟨*angle*⟩ degrees (not radians). `\Arc` draws the arc without arrow tips; `\VectorArc` draws the arc with one arrow tip at the end point; `\VectorARC` draws an arc with arrow tips at both ends. `\VVectorArc` is an alias of `\VectorARC`.
9. Command `\multiput` has been redefined to accept optional arguments, besides the use of coordinates of both kinds. The new syntax is the following:

```
\multiput[⟨shift⟩](⟨origin⟩)(⟨step⟩){⟨number⟩}{⟨object⟩}[⟨handler⟩]
```

where, if you neglect the first and the last (optional) arguments, you have the original syntax; the ⟨*origin*⟩ point is where the first ⟨*object*⟩ is placed; ⟨*step*⟩ is the displacement of a new ⟨*object*⟩ relative to the previous one; ⟨*number*⟩ is the total number of ⟨*object*⟩s put in place by the command; it is possible to specify the number trough an integer expression computed with the `\inteval` function of the L3 language, accessed through the `xfp` package functionalities already included into the LaTeX kernel. The new features are ⟨*shift*⟩, that is used to displace the whole drawing somewhere else (in case some fine tuning is required), and ⟨*handler*⟩; the latter is a powerful means to control both the object to be set in place and its position; further on there will be examples that show that the object can be put not only on straight paths, but also un other curves, including parabolas, circles, and other shapes.

10. Another version of repetitive commands `\xmultiput` is very similar to `\multiput` but the iterations are controlled in a different way so that it is possible also to draw continuous curves describing analytical functions even with parametric equations. Further on there will be some examples.
11. The `xfp` package is preloaded because not all functionalities have been made available in the LaTeX kernel; among such functionalities two two are very important, i.e. the L3 "functions", `\fpeval` and `\inteval`; the latter executes expressions on integer numbers containing the usual operators `+, -, *, /`; the division quotient is rounded to the nearest (positive or negative) integer. The former operates with real fractional numbers and, in addition to the usual arithmetical operators as `\inteval`, it can use many mathematical functions, from square roots, to exponentials, logarithms, trigonometric and hyperbolic direct and inverse functions[6], plus other ones. Normally fractional numbers are operated on decimal strings, with 16 fractional places, and 14 integer places but the L3 functions accept also scientific notation. The user can specify truncation or rounding to a specified number of digits. Such integer and fractional mathematical operations are already integrated in most computations performed by `curve2e`.
12. `curve2e` provides three more L3 functions: `\fptest`, `\fpwhiledo`, and `\fpdowhile` with the following syntax:

```
\fptest{⟨test⟩}{⟨true⟩}{⟨false⟩}
\fpdowhile{⟨test⟩}{⟨actions⟩}
\fowhiledo{⟨test⟩}{⟨actions⟩}
```

---

[6] The implementation of inverse hyperbolic functions at the moment is on the L3 Team "to do" list.

For all these macros the ⟨*test*⟩ is a logical L3 expression; its operands are logical constants, logical values, logical numeric comparisons; its operators are the typical `||`, `&&`, and `!`, respectively for OR, AND, and NOT. The logical numerical comparisons are mathematical constants or expressions connected with relation operators, such as `>, =, <`; such operators may be negated with the NOT operator; therefore, for example, `!>` means "not greater than", therefore "lower or equal to"; such operators may be coupled, for example `>=` makes a valid comparison and it is equal to `!<`.

13. The above tests are very useful to control both `\fptest` and `\fpdowhile` and `\fpwhiledo`. The logical ⟨*test*⟩ result lets `\fptest` execute only the ⟨*true*⟩ or the ⟨*false*⟩ code. Before using `\fpdowhile` or `\fpwhiledo` the ⟨*test*⟩ expression must be initialised to be `true`; the ⟨*actions*⟩ should contain some code to be iteratively executed, but they must contain some assignments, typically a change in an iteration counter, such that eventually the ⟨*test*⟩ logical expression becomes `false`. Lacking this assignments, the loop continues to infinity, or better, until a fatal error message is issued that informs that the program working memory is exhausted.
14. The difference between `\fpdowhile` and `\fpwhiledo` is the order in which the ⟨*test*⟩ and the first⟨*action*⟩ are executed; in facts `\fpdowhile` first does the ⟨*action*⟩ then the ⟨*test*⟩, while `\fpwhiledo` first executes the ⟨*test*⟩ then the ⟨*action*⟩. Since the modification of the test logical value is done by the commands contained in the ⟨*actions*⟩ this switching of the ⟨*test*⟩ and ⟨*action*⟩ produces different results. By adjusting the ⟨*test*⟩ it is possible to get the same results, but the expressiveness of the ⟨*test*⟩ may be easier to understand in one way rather than the other. Some of the examples show such different ⟨*test*⟩ syntaxes.
15. Such new commands are already used to code the `\multiput` and `\xmultiput` commands, but they are available also to the user who can operate in a very advanced way; further on, some examples will show certain advanced drawings.
16. General curves can be drawn by pic2e command `\curve` that is sort of difficult to use, because the user has to specify also the control points of the third order Bézier splines. Some other new commands are available with curve2e, that are supposed to be easier to use; they are described in the following items.
17. The new command `\Curve` joins a sequence of third order splines by simply specifying the node-direction coordinates; i.e. at the junction of two consecutive splines, in an interpolation node the final previous spline tangent has the same direction as that at the second spline first node; if a change of direction is required, an optional new direction can be specified. Therefore this triplet of information has the following syntax:

(⟨*node*⟩)<⟨*direction*⟩>[⟨*new direction*⟩]

Evidently the ⟨*new direction*⟩ is specified only for the nodes that correspond to a cusp. A variation of the command arguments is available by optionally specifying the "looseness" of the curve:

(⟨*node*⟩)<⟨*direction;start,end*⟩>[⟨*...*⟩]

where ⟨*start*⟩ is the spline starting "looseness" and ⟨*end*⟩ is the spline ending one. These (generally different) values are an index of how far the control point is from the adjacent node. With this functionality the user has a very good control on the curve shape and curvature. The optional new direction at a cusp point has the sane extended syntax

18. A similar command `\Qurve` works almost the same way, but it traces a quadratic Bézier spline; this one is specified only with two nodes an a single control point, therefore is less configurable than cubic splines; the same final line may require several quadratic splines when just a single cubic spline might do the same job. Notice also that quadratic splines are just parabolic arcs, therefore without inflection points, while a cubic spline can have one inflexion point.
19. A further advanced variation is obtained with the new `\CurveBetween` command that creates a single cubic spline between two given points with the following syntax:

    `\CurveBetween`⟨*node1*⟩ `And`⟨*node2*⟩ `WithDirs`⟨*dir1*⟩ `And`⟨*dir2*⟩

20. A similar variant command is defined with the following syntax:

    `\CbezierBetween`⟨*node1*⟩ `And`⟨*node2*⟩ `WithDirs`⟨*dir1*⟩ `And`⟨*dir2*⟩
    `UsingDists`⟨*dist1*⟩ `And`⟨*dist2*⟩

    Usage examples are shown in section 7

6 Euclidean geometry commands

With the already large available power of `curve2e` there was a push towards specialised applications; the first of which was, evidently, geometry; that kind of geometry that was used in the ancient times when mathematicians did not have available the sophisticated means they have today; they did not even have a positional numerical notation, that arrived in the "western world" we are familiar with, just by the XI-XII century; before replacing the roman numbering system, another couple of centuries passed by; real numbers with the notation we use today with a decimal separator, had to wait till the XVI century (at least); many things that naw are taught in elementary school were still a sort of magic until the end of XVIII century.

Even a simple algebraic second degree equation was a problem. In facts the Renaissance was the artistic period when the classical proportions were brought back to the artists who could not solve the simple equation where a segment of unit length is divided in two unequal parts x and $1-x$ such that the following proportion exists among the various parts and the whole segment:

$$\frac{x}{1}=\frac{1-x}{x}\Longrightarrow x=\frac{1}{x}-1$$

today we cam solve the problem by manipulating that simple proportion to get

$$x^2+x-1=0$$

and we know that the equation has two solution of opposite signs, and that their magnitudes are the reciprocal of one another. Since we are interested in their magnitudes, we adapt the solutions in the form

$$x_{1,2}=\frac{\sqrt{5}\pm1}{2}=\sqrt{1+0.5^2}\pm0.5\Longrightarrow\begin{cases}x_1=1.618\dots\\x_2=0.618\dots\end{cases}\tag{1}$$

The larger number is called the *golden number* and the smaller one the *golden section.*

Luca Pacioli, by the turn of centuries XV–XVI, was the tutor of Guidubaldo, the son and heir of Federico di Montefeltro, Duke of Urbino[7]; he wrote the famous book *De Diuina Proportione* that contained also the theory of the golden section accompanied by beautiful drawings of many Platonic solids and other non convex ones, drawn by Leonardo da Vinci. Everything was executed with perfect etchings, even the construction of the golden section; in its basic form[8] it is reproduced in figure 2.

By the way figure 2 shows also the code that is used for the drawing done completely with the facilities available just with curve2e. It is also a usage example of several commands.

Illiteracy was very widespread; books were expensive and were common just in the wealthy people mansions.

Mathematicians in the classical times B.C. up to the artists in the Renaissance, had no other means but to use geometrical constructions with ruler and compass. Even today in schools where calculus is not yet taught as a normal subject, possibly not in certain high school degree courses, but certainly not in elementary and junior high schools, the instructors have to recourse to geometrical constructions. Sometimes, as in Italy, access to public universities is open with no restrictions to all students with a high school diploma for degree courses that are more vocational than cultural. Therefore such students in some university degree courses have to frequent

---

[7] If you never visited this Renaissance city and its Ducal Palace, consider visiting it; it is one of the many UNESCO Heritage places.

[8] The third formula in equation (1) is written in such a way as to explain the graphical construction in figure 2.

```
\unitlength=0.005\linewidth
\begin{picture}(170,140)(0,-70)
\AutoGrid
\VECTOR(0,0)(170,0)
\Pbox(170,0)[t]{x}[0]
\Pbox(100,0)[t]{\mathrm{1}}[2]
\Pbox(0,0)[r]{O}[2]
\Arc(100,0)(50,0){-90}
\segment(100,0)(100,70)
\segment(0,0)(100,50)
\Pbox(50,0)[tr]{\mathrm{0.5}}[2]
\ModAndAngleOfVect100,50 to\M and\A
\Arc(0,0)(\M,0){\A}\Pbox(\M,0)[bl]{C}[2]
\Arc(\M,0)(\M,-50){90}
\Arc(\M,0)(\M,-50){-90}
\Pbox(\fpeval{\M-50},0)[b]{\mathit{x_2}}[3]
\Pbox(\fpeval{\M+50},0)[b]{\mathit{x_1}}[3]
\put(\M,0){\Vector(-70:50)}
\Pbox(120,-25)[bl]{\mathit{r}=\mathrm{0.5}}[0]
\thicklines
\segment(0,0)(100,0)
\end{picture}
```

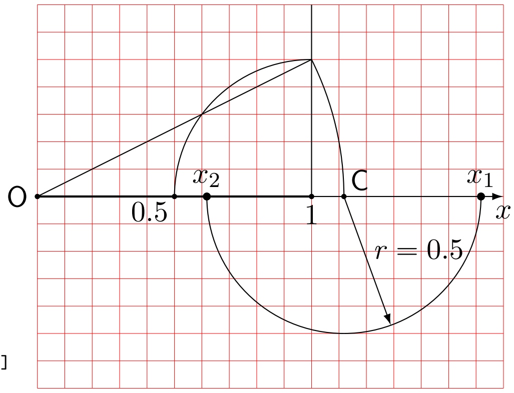


Figure 2: The golden section x_2 and the golden number x_1

upgrading courses in order to master some more mathematics compared to what they studied during their basic education.

The instructors nowadays very often prepare some booklets with their lessons; such documents, especially in electronic form, are a good help for many students. And LaTeX is used to write such documents. Therefore this extension module is mostly dedicated to such instructors.

The contents of this module is not exhaustive; it just shows a way to use the curve2e facilities to extend it to be suited for the kind of geometry they teach.

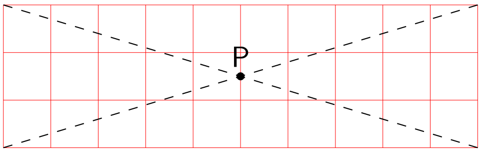
Here we describe the new commands provided by this package; then in section 7 we show their usage by means of examples.

1. Command `\polyvector` is simple extension of the `\polyline` command introduced by pict2e: its syntax is identical to that of `\polyline`; it draws a polyline with an arrow tip at the last segment; it turns out to be very handy while drawing block diagrams. See figure 4.
2. Command `\IntersectionOfLines` is a fundamental one; its syntax is the following:

    ```
    \IntersecionOfLines(⟨point1⟩)(⟨dir1⟩) and(⟨point2⟩)(⟨dir2⟩) to⟨vector⟩
    ```

    were each line is identified with its ⟨*point*⟩ and its *direction*; ⟨*vector*⟩ is the complex number that identifies the coordinates of the intersection point; the intersection coordinates go to the output ⟨*vector*⟩.

```
\unitlength=0.01\linewidth
\begin{picture}(100,30)
\AutoGrid
\Dashline(0,0)(100,30){2}
\Dashline(100,0)(0,30){2}
 \IntersectionOfLines%
  (0,0)(10,3)and%
  (100,0)(-10,3)to\P
\Pbox(\P)[b]{P}[3]
\end{picture}
```

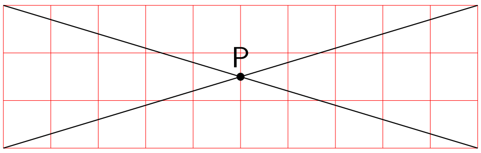


3. A second command `\IntersectionOfSegments` does almost the same work, but the coordinates of a segment end points define also its direction, which is the argument of the difference of the terminal nodes of each segment; the syntax therefore is the following:

```
\IntersectionOfSegments(⟨point11⟩)(⟨point12⟩)
    and(⟨point21⟩)(⟨point22⟩)to⟨intersection point⟩
```

   Again the intersection point coordinates go to the output ⟨*intersection point*⟩. The first segment is between points 11 and 12, and, similarly, the second segment is between points 21 and 22.

```
\unitlength=0.01\linewidth
\begin{picture}(100,30)
\AutoGrid
\segment(0,0)(100,30)
\segment(0,30)(100,0)
 \IntersectionOfSegments%
  (0,0)(100,30)and%
  (100,0)(0,30)to\P
\Pbox(\P)[b]{P}[3]
\end{picture}
```

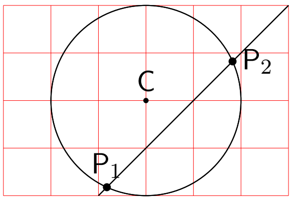


4. Another "intersection" command is `\IntersectionsOfLine` to determine both intersections of a line with a circle. The syntax is:

```
\IntersectionsOfLine(⟨point1⟩)(⟨point2⟩)
    WithCircle(⟨center⟩){⟨radius⟩}to⟨int1⟩ and⟨int2⟩
```

   where ⟨*point1*⟩ and ⟨*point2*⟩ identify a segment joining such points, and therefore a specific line with the segment direction and passing through a given point; the segment should cross the circle; this circle is identified with its ⟨*center*⟩ and ⟨*radius*⟩; the intersection points ⟨*int1*⟩ and ⟨*int2*⟩ are the coordinates of the intersection points; if the line and circle do not intersect, a warning message is issued, shown in the console and written to the `.log` file; the intersection points are assigned the default values `0,0`, which evidently produce strange results in the output document, so as to remind the user to give a look to the `.log` file and to review his/her data.

```
\unitlength=0.01\linewidth\raggedleft
\begin{picture}(60,40)(-10,0)
\AutoGrid
\def\Radius{20}%
\Circlewithcenter 20,20 radius\Radius
\segment(10,0)(50,40)
\IntersectionsOfLine(10,0)(50,40)%
  WithCircle(20,20)(20)%
  to\Pu and\Pd
\Pbox(\Pu)[b]{P_1}[3]
\Pbox(\Pd)[l]{P_2}[3]
\end{picture}
```

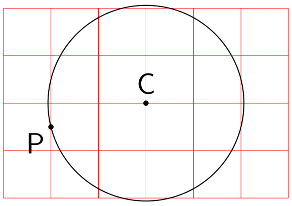


5. It is difficult to numerically determine the coordinates of the intersection points of two circles; it becomes easier if one of the intersections is known; to this end, a macro to draw a circle with a given center and passing through a given point is handy:

```
\CircleThrough⟨point⟩WithCenter{⟨center⟩}
```

draws such circumference.

```
\unitlength=0.01\linewidth\raggedleft
\begin{picture}(60,40)
\AutoGrid
\CopyVect10,15to\P
\Pbox(\P)[tr]{P}[2]
\Pbox(30,20)[b]{C}[2]
\def\Ce{30,20}\relax
\CircleThrough\P WithCenter\Ce
\end{picture}
```

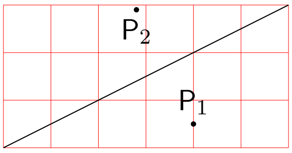


6. With the above macro it becomes easy to draw two circumferences with different centers and passing through the same point; therefore it becomes easy to determine the other intersection point by means of the following macro:

```
\Segment(⟨endpoint1⟩)(⟨endpoint2⟩)SymmetricPointOf⟨p1⟩
to⟨p2⟩
```

The computation is simple, because the second intersection is the symmetrical point ⟨*p2*⟩ of ⟨*p1*⟩ with respect to the segment that joins the centers of the given circles intersecting one another in ⟨*p1*⟩. The following small example displays how to find point P_2 symmetrical to P_1 with respect to a given line.

```
\raggedleft\unitlength=0.01\linewidth
\begin{picture}(60,30)
\AutoGrid
\segment(0,0)(60,30)
\CopyVect40,5to\Pu
\Segment(0,0)(60,30)%
  SymmetricPointOf\Pu to\Pd
\Pbox(\Pu)[b]{P_1}[2]
\Pbox(\Pd)[t]{P_2}[2]
\end{picture}
```



7. It would be interesting to solve the same problem with help of the following macro relating to right triangles identified with their hypothenuse and one of its legs; the other leg is found by means of this macro:

   ```
   \LegFromHypothenuse⟨length1⟩ AndOtherLeg⟨length2⟩
   to{⟨length3⟩}
   ```

   In facts, the intersection points of two circles define their common chord; this chord and the two circle centers define two isosceles triangles with the same base, the chord; therefore the segment joining the circle centers, coincides with the chord axis and divides each isosceles triangle in two right triangles, where the hypotenuse is one of two radii and the first leg is the distance from the chord middle point, intersection of the chord and the segment joining the circle centers; at this point the distance of the second point intersection from the chord middle point and the coordinates of the second intersection may be easily computed; of course this is a much clumsier way to determine the second intersection, but it is useful to solve this right triangle easy problem. See figure 16 where such right triangle legs are used for several tasks.
8. Command `\ThreePointCircle` draws a circle that goes through three given points; the syntax is the following:

   ```
   \ThreePointCircle⟨⋆⟩(⟨point1⟩)(⟨point2⟩)(⟨point3⟩)
   ```

   A sub product of this macro is formed by the vector `\C` that contains the coordinates of the center of the circle, that might be useful even if the circle is not drawn; the optional asterisk, if present, does not draw the circle, but its center is available. See figure 21, where this construction has been used.
9. Alternatively

   ```
   \ThreePointCircleCenter(⟨point1⟩)(⟨point2⟩)(⟨point3⟩)to⟨center vector⟩
   ```

   computes the three point circle center assigning its coordinates to ⟨*center vector*⟩.

10. Command `\CircleWithCenter` draws a circle given its center and it radius; in facts the syntax is the following:

    ```
    \CircleWithCenter⟨center⟩ Radius⟨Radius⟩
    ```

    This macro does not require the `\put` command to put the circle in place; it is very handy when the radius value is saved into a macro.
11. A similar macro `\Circlewithcenter` does almost the same; its syntax is the following:

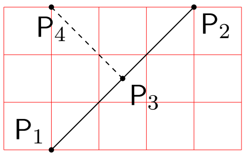
    ```
    \Circlewithcenter⟨center⟩ radius⟨radius⟩
    ```

    Apparently these two commands do the same, but, no, they behave differently: in the former command the ⟨*Radius*⟩ is a vector the modulus of which is computed and used as the radius; in the latter command the ⟨*radius*⟩ is a scalar and (its magnitude) is directly used. See item 4 of this enumeration where this command was used.
12. The command with syntax:

    ```
    \AxisOf⟨point1⟩ and⟨point2⟩ to ⟨point3⟩ and⟨point4⟩
    ```

    is used to determine the axis of a segment; the given segment is specified with its end points ⟨*point1*⟩ and ⟨*point2*⟩ and the axis is determined by point ⟨*point3*⟩ and ⟨*point4*⟩; actually ⟨*point3*⟩ is the middle point of the given segment.

    ```
    \raggedleft
    \unitlength=0.01\linewidth
    \begin{picture}(50,30)
    \AutoGrid
    \segment(10,0)(40,30)
    \AxisOf 10,0 and 40,30 to\Pu\Pd
    \Dashline(\Pu)(\Pd){1}
    \Pbox(10,0)[br]{P_1}[2]
    \Pbox(40,30)[tl]{P_2}[2]
    \Pbox(\Pu)[tl]{P_3}[2]
    \Pbox(\Pd)[t]{P_4}[2]
    \end{picture}
    ```

13. These two commands with syntax:

    ```
    \SegmentCenter(⟨point1⟩)(⟨point2⟩)to⟨center⟩
    \MiddlePointOf(⟨point1⟩)(⟨point2⟩)to⟨center⟩
    ```

    determine just the middle point between two given points. They are totally equivalent, aliases to one another; sometimes it is more convenient to use a name, sometimes the other; it helps reading the code and maintaining it.

14. Given a triangle and a specific vertex, it is possible to determine the middle point of the opposite side; it is not very difficult, but it is very handy to have all the necessary elements to draw the median line. The simple syntax is the following:

    ```
    \TriangleMedianBase⟨vertex⟩ on⟨base1⟩ and⟨base2⟩
        to⟨base middle point⟩
    ```

    A similar command `\TriangleHeightBase` is used to determine the intersection of the height segment from one vertex to the opposite side; with triangles that have an obtuse angle, the height base might lay externally to one of the bases adjacent to such an angle. The syntax is the following

    ```
    \TriangleHeigthtBase⟨vertex⟩ on⟨base1⟩ and⟨base2⟩ to⟨height
    base⟩
    ```

    Similarly there is the `\TriangleBisectorBase` macro with a similar syntax:

    ```
    \TriangleBisectorBase⟨vertex⟩ on⟨base1⟩ and⟨base2⟩
        to⟨bisector base⟩
    ```

    See figure 17 that contains all three special lines: the median (M) the height (H) and the bisector (B) lines.
15. A triangle *barycenter* is the point where its median lines intersect; command `\TriangleBarycenter` determines its coordinates with the following syntax.

    ```
    \TriangleBarycenter(⟨vertex1⟩)(⟨vertex2⟩)(⟨vertex3⟩)
    to⟨barycenter⟩
    ```

16. A triangle *orthocenter* is the point where its height lines intersect; command `\TriangleOrthocenter` determines its coordinates with the following syntax:

    ```
    \TriangleOrthocenter(⟨vertex1⟩)(⟨vertex2⟩)(⟨vertex3⟩)
    to⟨orthocenter⟩
    ```

17. A triangle *incenter* is the point where its bisector lines intersect; command `\TriangleIncenter` determines its coordinates with the following syntax:

    ```
    \TriangleIncenter(⟨vertex1⟩)(⟨vertex2⟩)(⟨vertex3⟩)
    to⟨incenter⟩
    ```

18. The distance of a specified point from a given segment or line is computed with the following command

```
\DistanceOfPoint⟨point⟩ from(⟨point1⟩)(⟨point2⟩) to⟨distance⟩
```

where ⟨*point*⟩ specifies the point and ⟨*point1*⟩ and ⟨*point2*⟩ identify two points on a segment or a line; ⟨*distance*⟩ is a scalar value.

19. In a construction that will be examined in section 7 we need to determine an ellipse axis if the other axis and the focal distance are know; actually it solves the relation

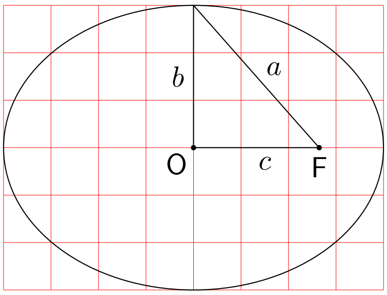
$$a^2 = b^2 + c^2 \tag{2}$$

that connects such three quantities; a is always the largest of the three quantities; therefore the macro tests if the first entry is larger than the second one: if is is, it computes a Pytagorean difference, otherwise the user should pay attention to use as the first entry the smaller among b and c, so as to compute a Pytagorean sum. The command is the following:

```
\AxisFromAxisAndFocus⟨axis or focus⟩ and⟨focus or axis⟩
    to⟨other axis or focus⟩
```

The word “axis” stands for “semi axis length”; the word “focus" stands for “focal semi distance”; actually the macro works equally well with full lengths, instead of half lengths; it is important not to mix full and half lengths. Such lengths are expressed as factors of `\unitlength`, not as absolute values. This command is described again when dealing with the specific problem referred to at the beginning of this list item; the description is going to be more detailed and another macro is added to avoid possible errors.

```
\raggedleft\unitlength=0.01\linewidth
\begin{picture}(80,60)(-40,-30)
\AutoGrid
\ellisse{40}{30}
\CopyVect\fpeval{sqrt(700)},0to\Focus
\segment(0,0)(0,30)
\segment(0,0)(\Focus)
\segment(\Focus)(0,30)
\Pbox(0,0)[tr]{O}[2]
\Pbox(\Focus)[t]{F}[2]
\Pbox(0,15)[r]{b}[0pt]
\Pbox(15,0)[t]{c}[0pt]
\Pbox(14,14)[bl]{a}[0pt]
\end{picture}
```



20. Given a segment, i.e. the coordinates of its end points, it is useful to have a macro that computes its length; at the same time it is useful to to compute its direction; this operation is not the same as to compute modulus and argument of a vector, but consists in computing such

quantities from the difference of the vectors pointing to the segment end points. Possibly better names should be devised for such macros, for example `\DistanceBetweenPoints...` and `\AngleOf..` But I could not find anything really more expressive. These two macros are the following:

```
\SegmentLength(⟨point1⟩)(⟨point2⟩) to⟨length⟩
\SegmentArg(⟨point1⟩)(⟨point2⟩) to⟨angle⟩
```

The ⟨*angle*⟩ is computed in the interval $-180^\circ < \varphi \leq +180^\circ$; it represents the with respect to the positive real axis of the vector that goes from ⟨*point1*⟩ to ⟨*point2*⟩, therefore the user must pay attention to the order s/he enters the end point coordinates.

21. The next command `\SymmetricalPointOf` is used to find the reflection of a specified point with respect to a fixed point; of course the latter is the middle point of the couple, but the unknown to be determined is not the center of a segment, but one of its end points. The syntax is the following: In a sense this macro completes the `\AxisOf` by finding the other extremity of the axis.

```
\SymmetricalPointOf⟨point1⟩ respect⟨fixed⟩ to⟨point2⟩
```

22. Command `\RegPolygon` draws a regular polygon inscribed within a circle of given radius and center, with a specified number of sides; optional arguments allow to specify color and thickness of the sides, or the polygon interior color; this macro operates differently from the one for drawing ellipses, that draws simultaneously an ellipse with the border of a color and the interior of another one; with this macro the user who wants to achieve this effect must superimpose to polygons with different settings; but it would not be too difficult to arrange a new macro or to modify this one in order to get “bicolor” polygons. It is not necessary for the purpose of this package, therefore we let the user express his/her phantasy by creating other macros. The actual syntax is the following:

```
\RegPolygon⟨⋆⟩(⟨center⟩){⟨radius⟩}{⟨sides⟩}[⟨angle⟩]<⟨settings⟩>
```

The initial optional asterisk specifies if the interior has to be coloured; if yes, the ⟨*settings*⟩ refer to the color of the interior; if not, the ⟨*settings*⟩ refer to the thickness and color of the sides; no ⟨*settings*⟩ imply sides drawn with the default line thickness, generally the one corresponding to `\thinlines`, and the default color (generally black) for the sides or the interior. By default the first vertex is set to an angle of 0° with respect to the ⟨*center*⟩; the optional ⟨*angle*⟩ modifies this value to what is necessary for a particular polygon. The ⟨*center*⟩ itself is optional, in the sense that if it is not specified the center lays in the origin of the ***picture*** axes; if this argument is specified, the polygon center is displaced accordingly. The number of sides in theory may be very high, but it is not wise to exceed a couple of dozen sides; if the number of sides is too high, a polygon (completely contained in an A4 page) may become undistinguishable from a circumference. Some examples are shown in figures 6 and 7.

23. Several macros are dedicated to ellipses; their names are spelled in Italian, “ellisse”, because the name “ellipse” is already taken by other packages; with Italian command names there should be no interference with other packages, or the risk is reduced to a minimum. The various macros are `\ellisse`, `\Sellisse`, `\Xellisse`, `\XSellisse`, `\EllisseConFuoco` `\EllisseSteiner`; the last two control sequence names are aliased with the corresponding English ones `\EllipseWithFocus` and `\SteinerEllipse`. For the other four ones it is wise to avoid English names for the reasons explained above. After all the Italian and the English names are very similar and are pronounced almost identically.

    Actually `\ellisse` is practically a shorthand for `\Sellisse` because some optional arguments are already fixed, but the meaning of `\fillstroke` depends on the presence or absence of an initial asterisk; similarly `\Xellisse` is a sort of a shorthand for `\XSellisse`; in facts those commands, that contain an ‘S’ in their names, can optionally perform also the affine *shear* transformation, while those without the ‘S’ do not execute such transformation. Figure 3 displays a normal ellipse with its bounding rectangle, and the same ellipse to which the shear affine transformation is applied; the labeled points represent the third order Bézier spline nodes and control points.

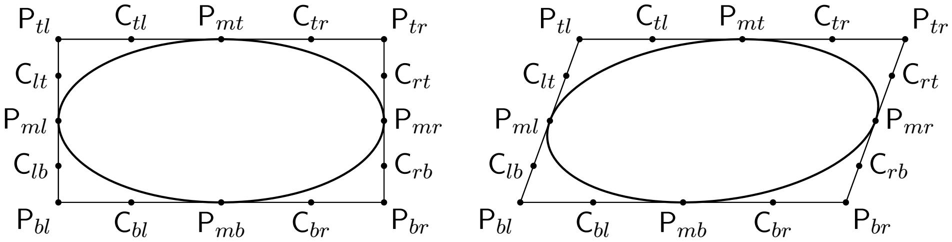


Figure 3: The effect of shearing an ellipse with its bounding rectangle

24. The syntax of those six commands are the following:

```
\Sellisse⟨⋆⟩{⟨semiaxis-h⟩}{⟨semiaxis-v⟩}[⟨shear⟩]
\ellisse⟨⋆⟩{⟨semiaxis-h⟩}{⟨semiaxis-v⟩}
\XSellisse⟨⋆⟩(⟨center⟩)[⟨angle⟩]%
    <⟨shear⟩>{⟨semiaxis-h⟩}{⟨semiaxis-v⟩}⟨⋆⟩%
    [⟨settings1⟩][⟨settings2⟩]
\Xellisse⟨⋆⟩(⟨center⟩)[⟨angle⟩]{⟨semiaxis-h⟩}%
    {⟨semiaxis-v⟩}[⟨settings1⟩]{⟨settings2⟩}
\EllipseWithFocus⟨⋆⟩(⟨vertex1⟩)(⟨vertex2⟩)(⟨vertex3⟩)(⟨focus⟩)
\SteinerEllipse⟨⋆⟩(⟨vertex1⟩)(⟨vertex2⟩)(⟨vertex3⟩)[⟨diameter⟩]
```

All require the semi axis lengths; the ⟨*semiaxis-h*⟩ and ⟨*semiaxis-v*⟩ refer to the semi axes before possible rotation by ⟨*angle*⟩ degrees, and do not make assumptions on which axis is the larger one. The optional parameter ⟨*shear*⟩ is the angle in degrees by which the vertical coordinate lines are slanted by effect of shearing. If ⟨*shear*⟩, that by default equals zero, is not set to another value, the asterisks of command `\Sellisse` and `\XSellisse` do not have any effect. Otherwise the asterisk of `\Sellisse` forces to draw the ellipse bounding box (rectangle before shearing, parallelogram after shearing) as shown together with some marked special points (the vertices, spline nodes and control points of the quarter circle or quarter ellipse Bézier splines) in figure 3. For `\ellisse` the asterisk implies filling, instead of stroking the ellipse contour. The ⟨*setting*⟩ 1 and 2 refer to the color filling and/or border color, and contour thickness, as already explained. For the `\EllipseWithFocus`, the ⟨*focus*⟩ contains the coordinates of one of the two ellipse foci; such coordinates should point to some position *inside* the triangle. The `\SteinerEllipse` requires less data, in the sense that such ellipse is unique; it is the ellipse internally tangent to the triangle at its side middle points. See detailed examples in the following section.

7 Examples

Here we can show some examples of the advanced curve2e commands and of what can be done with this euclideangeometry extension.

7.1 Straight and curved vectors

Figure 4 shows some vectors and vector arcs with the code used to draw them; some points are described with cartesian coordinates and some with polar ones.

```
\unitlength=0.01\linewidth
\begin{picture}(100,60)
\AutoGrid
\put(0,30){\vector(1,2){10}}
\put(20,30){\Vector(10,20)}
\VECTOR(40,30)(50,50)
\VVECTOR(60,30)(70,60)
\Arc(100,60)(100,40){-90}
\VectorArc(0,0)(0:20){90}
\VectorARC(100,0)(0:80){-90}
\polyvector(30,0)(35,10)(55,20)(60,0)
\end{picture}
```

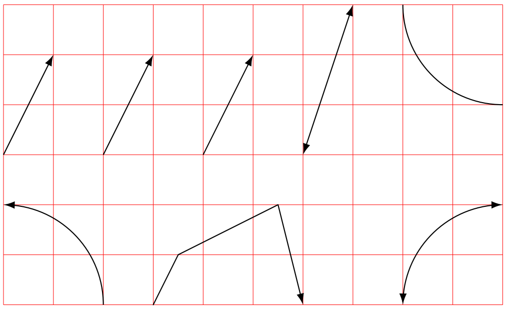


Figure 4: Some vectors and vector arcs

7.2 Polygons

Figures 6 and 7 display a normal and a color filled pentagon with their codes. Figure 5 shows a variety of polygons with their codes.

7.3 Dashed and dotted lines

For dashed lines it is mandatory to specify the dash length, that is just as long as the gap between dashes. For dotted lines it is mandatory to specify the dot gap. For dotted lines it is also possible to specify the dot size; it can be specified with an explicit unit of measure, or, if no unit is specified, it is assumed to be "points". The `\Dotline` takes care of transforming the implied or the explicit dimension in multiples of `\unitlength`. Figure 8 shows some examples with their codes.

7.4 Generic curves

With the `\Curve` macro it is possible to make line art or filled shapes. The hearts drawn in figure 9 show the same shape, the first just stroked and the second color filled.

7.5 The `\multiput` command

The new `\multiput` and `\xmultiput` commands are extensions of the original `\multiput` macro; both are used to put a number of objects according to a discrete law; but they can produce surprising effects. Figure 10 displays several examples. As it possible to see, the black dots are evenly distributed along the canvas diagonal; the green filled squares are along a sloping down line inclined by 15° as specified by the polar coordinates of the ⟨*increment*⟩; the blue filled triangles are distributed along a parabola; the red stroked diamonds are distributed along a half sine wave.

Another interesting construction is a clock quadrant; this is shown in figure 11

```
\centering
\unitlength=0.006\linewidth\begin{picture}(120,90)
%
\RegPolygon(9,20){20}{6}<\linethickness{3pt}\color{red}>
\RegPolygon(55,20){20}{7}[90]
\RegPolygon(100,20){20}{8}[22.5]<\linethickness{0.5ex}\color{blue}>
%
\put(0,50){%
  \RegPolygon(9,20){20}{3}\RegPolygon(9,20){20}{3}[30]
  \RegPolygon(9,20){20}{3}[60]\RegPolygon(9,20){20}{3}[90]
%
  \RegPolygon*(55,20){20}{4}<\color{green}>
  \RegPolygon(55,20){20}{4}<\linethickness{1ex}>
%
  \RegPolygon*(100,20){20}{4}[45]<\color{orange}>
  \RegPolygon(100,20){20}{4}[45]<\linethickness{1ex}\color{blue}>
}
\end{picture}
```

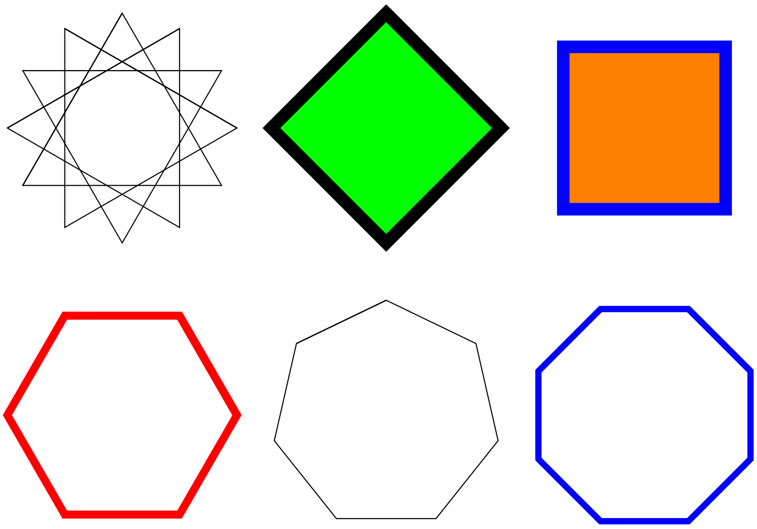


Figure 5: A variety of polygons and their codes

```
\unitlength=0.5mm
\begin{picture}(40,32)(-20,-17)
\polyline(90:20)(162:20)(234:20)(306:20)(378:20)(90:20)
\end{picture}
```

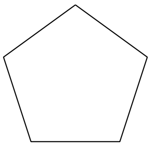


Figure 6: A normal polygon drawn with `\polyline`

7.6 Drawing mathematical functions

Figure 12 shows an equilateral hyperbola; since it has asymptotes, the drawing must be carefully done avoiding overflows, and parts of drawing out of

```
\unitlength=.5mm
\begin{picture}(40,32)(-20,-20)
\color{magenta}
\polygon*(90:20)(162:20)(234:20)(306:20)(378:20)
\end{picture}
```

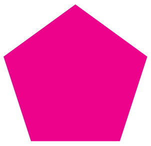


Figure 7: A filled polygon drawn with `\polygon`

```
\unitlength=0.02\linewidth
\begin{picture}(40,40)
\AutoGrid
\Dashline(0,0)(40,10){4}
\put(0,0){\circle*{1}}
\Dashline(40,10)(0,25){4}
\put(40,10){\circle*{1}}
\Dashline(0,25)(20,40){4}
\put(0,25){\circle*{1}}
\put(20,40){\circle*{1}}
\Dotline(0,0)(40,40){2}[2]
\end{picture}
```

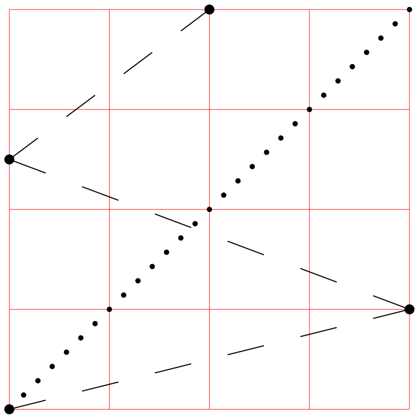


Figure 8: Dashed and dotted lines

the ***picture*** area. Nevertheless the possibility of describing mathematical functions in terms of L3 functions (in spite of the same name, they are completely different things) makes it possible to exploit the ⟨*settings*⟩ argument to do the job with `\xmultiput`.

A more complicated drawing can be done by expressing the function to draw with parametric equations; the idea is to code the math formulas

$$\begin{cases} x(t) = f_1(t)\\ y(t) = f_2(t)\end{cases}$$

because it is easy to code the x and the y component and use the `\fpdowhile` command to trace the curve with a piecewise continuous line; actually a continuous line with a piecewise continuous derivative; it is important to sample the curve in a sufficiently dense way. A heart shaped mathematical function taken from the internet[9] is the following

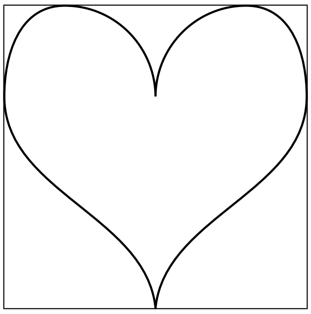
$$\begin{aligned} x(t) &= \sin^3(t)\\ y(t) &= \frac{13\cos(t) - 5\cos(2t) - 2\cos(3t) - \cos(4t)}{16}\end{aligned}$$

Figure 13 displays the graph, and its code, and, most important, the L3 definition of the parametric equations. Compared to the previous equations

---

[9] `http://mathworld.wolfram.com/HeartCurve.html` reports several formulas, including the cardioid, but the one we use here is a different function.

```
\unitlength=8mm\relax
\begin{picture}(5,5)
\put(0,0){\framebox(5,5){}}\thicklines\roundcap
\Curve(2.5,0)<0.1,1>(5,3.5)<0,1>%
  (4,5)<-1,0>(2.5,3.5)<-0.01,-1.2>[-0.01,1.2]%
  (1,5)<-1,0>(0,3.5)<0,-1>(2.5,0)<0.1,-1>
\end{picture}
```



```
\unitlength=8mm\relax
\begin{picture}(5,5)
\put(0,0){\framebox(5,5){}}\thicklines\roundcap
\color{orange}\relax
\Curve*(2.5,0)<0.1,1>(5,3.5)<0,1>%
(4,5)<-1,0>(2.5,3.5)<-0.01,-1.2>[-0.01,1.2]%
(1,5)<-1,0>(0,3.5)<0,-1>(2.5,0)<0.1,-1>
\end{picture}
```

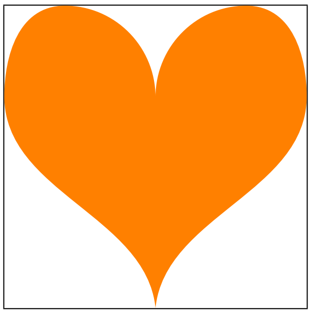


Figure 9: A stroked and a filled heart shaped contour

```
\unitlength=0.01\linewidth
\begin{picture}(100,100)
\AutoGrid
\multiput(0,0)(10,10){11}{\circle*{2}}
\color{blue!70!white}
\multiput(0,0)(10,0){11}{%
\RegPolygon*{2}{3}<\color{blue!70!white}>}%
  [\GetCoord(\R)\X\Y
  \edef\X{\fpeval{\X+10}}
  \edef\Y{\fpeval{(\X/10)**2}}
  \CopyVect\X,\Y to\R]
\multiput(0,0)(10,1){11}{%
\RegPolygon{2}{4}<\color{magenta}>}%
  [\GetCoord(\R)\X\Y
   \edef\X{\fpeval{\X+10}}
   \edef\Y{\fpeval{sind(\X*1.8)*100}}
   \CopyVect\X,\Y to\R]
\multiput(50,50)(-15:5){11}{%
\RegPolygon*{2}{4}[45]<\color{green!60!black}>}
\end{picture}
```

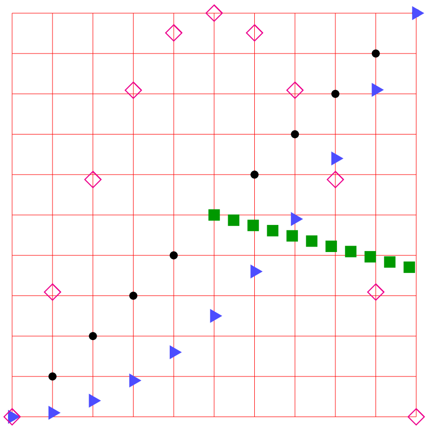


Figure 10: Some examples of the ⟨*handler*⟩ optional argument

we applied a scale factor and added the final term (2.4) in order to shift a little bit the drawing so as to vertically center it .

```
\unitlength=0.0095\linewidth
\begin{picture}(100,100)
\AutoGrid
\put(50,50){\thicklines\circle{100}}
\xmultiput[50,50](60:35)(-30:1){12}%
  {\makebox(0,0){\circle*{2}}}%
    [\MultVect\R by\D to\R]%
\xmultiput[50,50](60:40)(-30:1){12}%
  {\ArgOfVect\R to\Ang
    \rotatebox{\fpeval{\Ang-90}}%
    {\makebox(0,0)[b]{%
       \Roman{multicnt}}}}%
      [\Multvect{\R}{\D}\R]
\thicklines\put(50,50){\circle*{4}}
\put(50,50){\Vector(37.5:30)}
\put(50,50){\Vector(180:33)}
\end{picture}
```

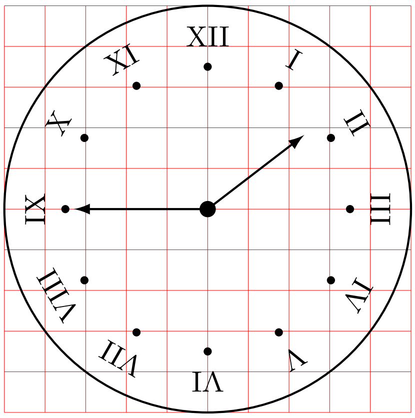


Figure 11: Usage example of the `\xmultiput` command

```
\unitlength=0.008\linewidth
\begin{picture}(100,100)
\AutoGrid
\VECTOR(0,0)(100,0)\Pbox(100,0)[tr]{x}[0]
\VECTOR(0,0)(0,100)\Pbox(0,100)[tr]{y}[0]
\Pbox(0,0)[r]{O}[3pt]
\thicklines
\moveto(10,100)\countdef\I=2560 \I=11
\xmultiput(0,0)(1,0){101}%
  {\lineto(\I,\fpeval{1000/\I})}%
  [\advance\I by1 \value{multicnt}=\I]
\strokepath
\end{picture}
```

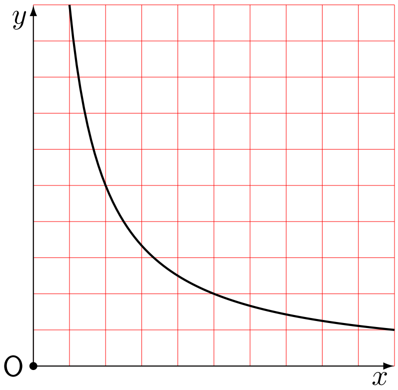


Figure 12: An equilateral hyperbola drawn with a thinly sampled piecewise continuous line

7.7 Intersections involving circles

Determining the intersection of two circles is difficult; algebraically it requires the solution of second degree equations whose coefficients are sort of complicated expressions of the centers and radii; furthermore such equations might not have real roots. The problems are much simpler even geometrically when one circle is intersected by a line, and when two circles share a common point. In the first case there are two intersections, possibly coincident, if the centers distance is shorter than the sum of their radii; in the second case its easy to determine the chord common to both circles and the problem of finding the second intersection becomes that of finding the second end point of the chord.

Figure 14 shows the simple geometrical construction that leads to the determination of the intersections; one of the lines is tangent to the circle

```
%             Parametric equations of the "heart"
\providecommand\heart[3]{%
\edef\X{\fpeval{#1*16*(sind(#2)^3)}}
\edef\Y{\fpeval{#1*(13*cosd(#2) - 5*cosd(2*#2)
  - 2*cosd(3*#2) -cosd(4*#2)+2.4)}}
\CopyVect\X,\Y to#3}
%
%                                       Drawing
\unitlength=0,005\linewidth
\begin{picture}(200,200)(-100,-100)
\AutoGrid
\VECTOR(-100,0)(100,0)\Pbox(100,0)[tr]{x}[0]
\VECTOR(0,-100)(0,100)\Pbox(0,100)[tl]{y}[0]
\Pbox(0,0)[tr]{O}\linethickness{1pt}\bgroup
\edef\scala{\fpeval{100/16}}
\countdef\I=2560 \I=0\roundjoin
\fpdowhile{\I !>360}{\heart\scala\I\Punto
\ifnum\I=0 \moveto(\Punto)\else \lineto(\Punto)\fi
\advance\I by 3}\strokepath\egroup
\end{picture}
```

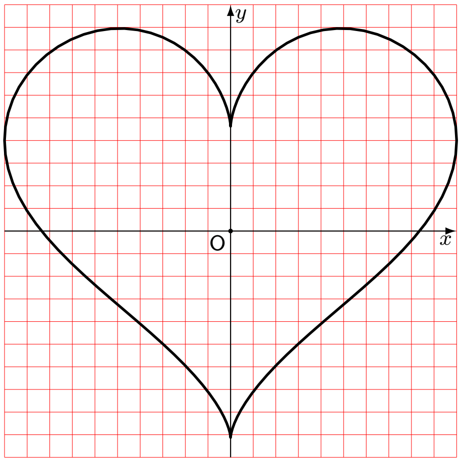


Figure 13: A heart shaped mathematical function drawn with a thinly sampled piecewise continuous line

and the intersection points P_3 and P_4 coincide.

```
\unitlength=0.007\linewidth
\begin{picture}(100,100)
\AutoGrid
\IntersectionsOfLine(100,50)(0,20)%
  WithCircle(40,40){30}to\Puno and\Pdue
\Pbox(\Puno)[tl]{P_1}[2]
\Pbox(\Pdue)[t]{P_2}[2]
\Dotline(\C)(\Pt){2}[1.5]
\Dotline(\C)(\Pq){2}[1.5]
\Pbox(\Int)[t]{M}[2]
\Dotline(\C)(\Int){2}[1.5]
%
\IntersectionsOfLine(0,70)(100,70)%
  WithCircle(40,40){30}to\Ptre and\Pquat
\Pbox(\Ptre)[bl]{P_3}[2]
\Pbox(\Pquat)[br]{P_4}[2]
\IntersectionsOfLine(0,40)(100,100)%
  WithCircle(40,40){30}to\Pcin and\Psei
\Pbox(\Pcin)[br]{P_5}[2]
\Pbox(\Psei)[t]{P_6}[2]
\end{picture}
```

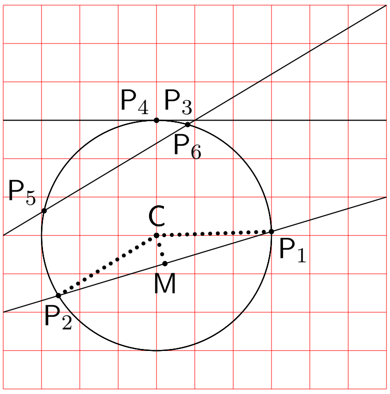


Figure 14: Intersection of a circle with several lines

Figure 15 shows the geometrical construction to determine the second intersection point P_2 of two circles that already have a first common point P_1. The common chord and the segment joining the centers are not shown, but the code, although with “strange” point names, shows all the steps necessary to find the second intersection point

The following macro allows to determine both intersections, if they exist, of two generic circles; of course the macro is a little more complicated than

```
\unitlength0.007\linewidth
\begin{picture}(100,100)
\AutoGrid
\edef\PCCuno{30,40}
\edef\PCCdue{70,60}
\edef\PCCzero{40,55}
\Pbox(\PCCuno)[t]{C_1}[2]
\Pbox(\PCCdue)[t]{C_2}[2]
\Pbox(\PCCzero)[l]{P_1}[2.5]
\CircleThrough\PCCzero WithCenter\PCCuno
\CircleThrough\PCCzero WithCenter\PCCdue
\Segment(\PCCuno)(\PCCdue)%
  SymmetricPointOf\PCCzero to\PCCquat
\Pbox(\PCCquat)[l]{P_2}[2.5]
\end{picture}
```

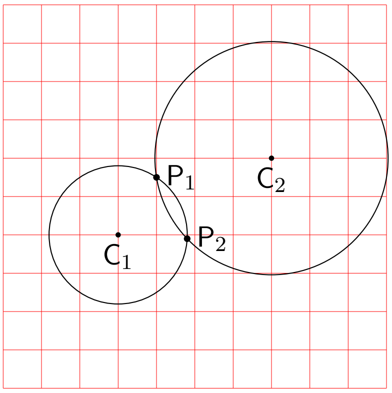


Figure 15: Second intersection point P_2 of two circles sharing a first intersection point P_1

the above macro that has to find the second intersection when one is already known. Even the reasoning behind the macro is quite different and slightly “creative”.

In facts the reasoning is to avoid any or most analytical computations; analytically it would be quite simple to set up a system of two second degree polynomial equations; after processing such a system it is necessary to solve a second degree equation, but in order to control if there are intersections it should be necessary to discuss the value and sign of the discriminant; it is nothing special when doing all this by hand, but it involves a complicated code in terms of the LaTeX language.

It is much simpler to reason geometrically; imagine to draw two circles; let $a > 0$ be the distance of their centers[10], and let R_1 and R_2 be their radii. Then if

$$|R_1 - R_2| \le a \le R_1 + R_2$$

the intersections do exist, even if it is possible that the circles are tangent to one another and the two intersection points become a (double) one; this takes place when either ‘equals’ sign applies. If the left boundary is not satisfied the centers are too close to one another and the internal circle is too small compared to the external one. On the opposite if the upper bound is not satisfied the second circle is outside the first one and too far away.

The macro controls the above range, and if the the input data do not satisfy the range boundaries, there are no intersections: a warning message is issued, but computations go on with non sense values for both output coordinates; may be other errors are produced, but in any case the successive drawing lines will not be acceptable; a good sign to the user who may have not noticed the warning message in his/her console, but is immediately

---

[10] If $a = 0$ the circles are concentric and do not intersect.

"forced" to consult this manual and find out this explanation; s/he will then review his/her code in oder to change the drawing data.

If the data cope with the above range, the computations go on along this simple reasoning, that it drawn in the left part of figure 16. There you see the segment that joins the two centers, and the intersection points to be found. They are the end points of the chord common to both circles; using as vertices the two centers and such chord as the base, two isosceles triangles are formed; the segment joining the centers bisects both triangles forming four right triangles where the hypotenuse is formed by the pertinent radius, and one leg is half the chord; with reference to the triangles IC_1P_1 and IC_2P_1, Pythagoras' theorem lets us determine the relation between the common leg IP_1 and the other triangle sides; this is the small analytical computation we have to execute, so as to compute the distance c from the center C_1 and the common leg length h. These two values are sufficient, together with the direction of segment C_1C_2 to find the intersection points coordinates.

The syntax il the following

```
\TwoCirclesIntersections(⟨C1⟩)(⟨C2⟩)withradii{⟨R1⟩} and{⟨R2⟩}
to⟨P1⟩ and⟨P2⟩
```

The symbols are self explanatory; as usual, input data (those entered before the keyword `to`) may be control sequences (defined with the necessary data), or explicit data; on the opposite the output ones must be control sequences.

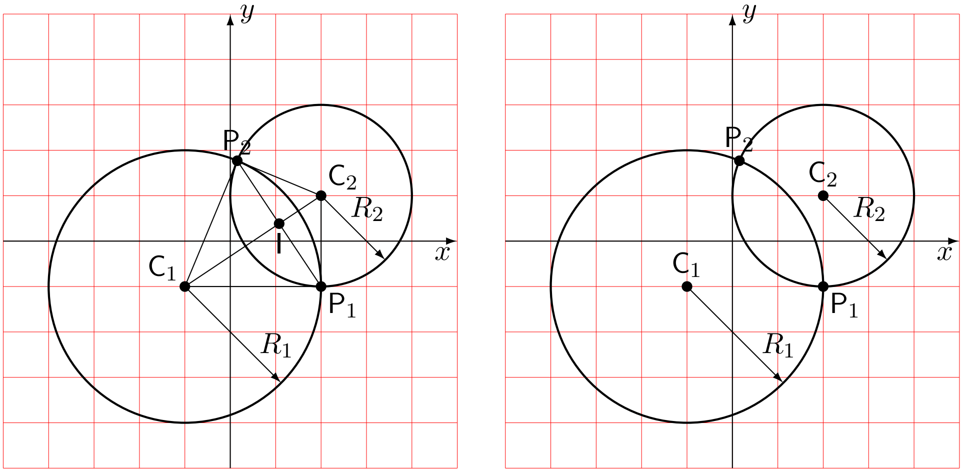


Figure 16: Intersections of two generic circles

The code for drawing figure 16 is the following.

```
\begin{minipage}{0.475\linewidth}% Geometrical construction
\unitlength0.01\linewidth
\begin{picture}(100,100)(-50,-50)
```

```
\AutoGrid
\VECTOR(-50,0)(50,0) \Pbox(50,0)[tr]{x}[0]
\VECTOR(0,-50)(0,50) \Pbox(0,50)[l]{y}[0]
\edef\Kuno{-10,-10}\edef\RKuno{30}%
\edef\Kdue{20,10}\edef\RKdue{20}
\thicklines
\Circlewithcenter\Kuno radius\RKuno
\Circlewithcenter\Kdue radius\RKdue
\thinlines
\TwoCirclesIntersections(\Kuno)(\Kdue)withradii\RKuno
   and\RKdue to\Puno and\Pdue
\Pbox(\Kuno)[br]{C_1}[4]        \Pbox(\Kdue)[bl]{C_2}[4]
\Pbox(\Puno)[tl]{P_1}[4]        \Pbox(\Pdue)[bc]{P_2}[4]
\put(\Kuno){\Vector(-45:\RKuno)}\Pbox(5,-27)[bl]{R_1}[0]
\put(\Kdue){\Vector(-45:\RKdue)}\Pbox(25, 3)[bl]{R_2}[0]
                                \Pbox(\CI)[t]{I}[4]
%
\segment(\Kuno)(\Kdue)\segment(\Puno)(\Pdue)
\segment(\Kuno)(\Pdue)\segment(\Kdue)(\Pdue)
\segment(\Kuno)(\Puno)\segment(\Puno)(\Kdue)
\end{picture}
\end{minipage}
\hfill
\begin{minipage}{0.475\linewidth}% Clean final result
\unitlength0.01\linewidth
\begin{picture}(100,100)(-50,-50)
\AutoGrid
\VECTOR(-50,0)(50,0) \Pbox(50,0)[tr]{x}[0]
\VECTOR(0,-50)(0,50) \Pbox(0,50)[l]{y}[0]
\edef\Kuno{-10,-10}\edef\RKuno{30}%
\edef\Kdue{20,10}\edef\RKdue{20}
\thicklines
\Circlewithcenter\Kuno radius\RKuno
\Circlewithcenter\Kdue radius\RKdue
\thinlines
\TwoCirclesIntersections(\Kuno)(\Kdue)withradii\RKuno
   and\RKdue to\Puno and\Pdue
\Pbox(\Kuno)[b]{C_1}[4] \Pbox(\Kdue)[b]{C_2}[4]
\Pbox(\Puno)[tl]{P_1}[4] \Pbox(\Pdue)[b]{P_2}[4]
\put(\Kuno){\Vector(-45:\RKuno)}\Pbox(5,-27)[bl]{R_1}[0]
\put(\Kdue){\Vector(-45:\RKdue)}\Pbox(25,3)[bl]{R_2}[0]
\end{picture}
\end{minipage}
```

```
\unitlength=0.008\linewidth
\begin{picture}(100,100)(0,-10)
\AutoGrid
\def\Puno{0,0} \def\Pdue{0,80} \def\Ptre{100,55}
 {\thicklines\polygon(\Puno)(\Pdue)(\Ptre)}%
 \Pbox(\Puno)[tc]{P_1}[2]
\Pbox(\Pdue)[bc]{P_2}[2]\Pbox(\Ptre)[bc]{P_3}[2]
% Median
 \TriangleMedianBase\Puno on \Pdue and \Ptre to\M
 \Pbox(\M)[bc]{M}[2]\segment(\Puno)(\M)
% Height
 \TriangleHeightBase\Puno on \Pdue and\Ptre to\H
 \Dotline(\Puno)(\H){2}[1.5]\Pbox(\H)[bc]{H}[1.5]
% Bisector
 \TriangleBisectorBase\Puno on\Pdue and\Ptre to\B
 \Dashline(\Puno)(\B){1.5}\Pbox(\B)[b]{B}[2]
\end{picture}
```

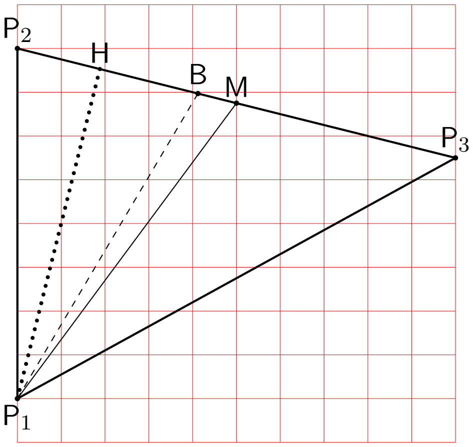


Figure 17: A triangle with the median, the height, and the bisector lines from a specific vertex

7.8 Triangles and their special lines

Triangles have special lines; they are the median, the height, and the bisector lines. They join each vertex with a specific point of the apposite side, respectively with the middle point, the intersection with the side perpendicular line, and the intersection with the bisector line. Figure 17 displays the construction of the three special lines relative to a specific vertex. Thanks to the macros described earlier in the preceding section, this drawing is particularly simple; most of the code is dedicated to labelling the various points and to assign coordinate values to the macros that are going to be used in a symbolic way. The generic triangle (not a regular polygon) requires one line of code, and the determination of the intersections of the lines with the suitable triangle side, and their tracing requires two code lines each.

7.9 Special triangle centers

Three triangle special lines of the same kind intersect each other in a special point; the median lines intersect in the *barycenter*, the height lines in the *orthocenter*, the bisector lines in the *incenter*; these centers may be those of special circles: see figures 18 to 21; the *incircle*, centred in the incenter, has a special name, because it has the property of being tangent to all three triangle sides; there is also the circumcircle that passes through the three vertices, its center is the intersection of the three side axes. Figures 18, 19, 20, and 21 display the necessary constructions and, possibly, also the special circles they are centers of. There is also the *nine point circle*, figure 22. It is worth noting the alla centers fall inside the various triangles, except the orthocenter that falls outside when the triangle is obtuse.

Given the triangle with vertices P_1, P_2, P_3, the nine point circle is shown

```
\unitlength=0.008\linewidth
\begin{picture}(100,100)(0,-10)
\AutoGrid
\def\Puno{0,0}\def\Pdue{0,80}\def\Ptre{100,55}
{\linethickness{0.6pt}\polygon(\Puno)(\Pdue)(\Ptre)}%
\Pbox(\Puno)[tl]{P_1}[1.5]%
\Pbox(\Pdue)[bl]{P_2}[1.5]\Pbox(\Ptre)[bc]{P_3}[1.5]
 \TriangleMedianBase\Puno on\Pdue and \Ptre to\Mu
 \TriangleMedianBase\Pdue on\Ptre and \Puno to\Md
 \TriangleMedianBase\Ptre on\Puno and \Pdue to\Mt
\Dotline(\Puno)(\Mu){3}[1.5]
\Dotline(\Pdue)(\Md){3}[1.5]
\Dotline(\Ptre)(\Mt){3}[1.5]
\IntersectionOfSegments(\Puno)(\Mu)and(\Pdue)(\Md)to\C
\Pbox(\C)[t]{B}[2]
\end{picture}
```

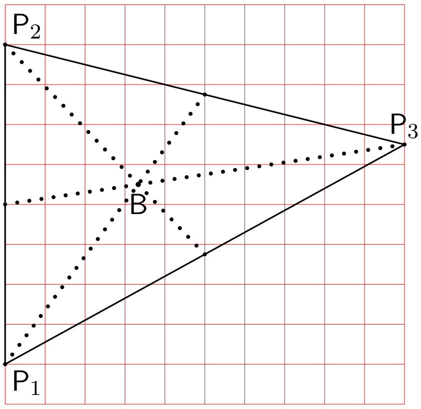


Figure 18: Determination of the barycenter

```
\unitlength=0.008\linewidth
\begin{picture}(100,100)(0,-10)
\AutoGrid
\def\Puno{0,0}\def\Pdue{0,80}\def\Ptre{100,55}
{\linethickness{0.6pt}\polygon(\Puno)(\Pdue)(\Ptre)}%
\Pbox(\Puno)[tl]{P_1}[1.5]%
\Pbox(\Pdue)[bl]{P_2}[1.5]\Pbox(\Ptre)[bc]{P_3}[1.5]
 \TriangleHeightBase\Puno on\Pdue and \Ptre to\Hu
 \TriangleHeightBase\Pdue on\Ptre and \Puno to\Hd
 \TriangleHeightBase\Ptre on\Puno and \Pdue to\Ht
\Dotline(\Puno)(\Hu){3}[1.5]
\Dotline(\Pdue)(\Hd){3}[1.5]
\Dotline(\Ptre)(\Ht){3}[1.5]
\IntersectionOfSegments(\Puno)(\Hu)and(\Pdue)(\Hd)to\C
\Pbox(\C)[t]{H}[2]
\end{picture}
```

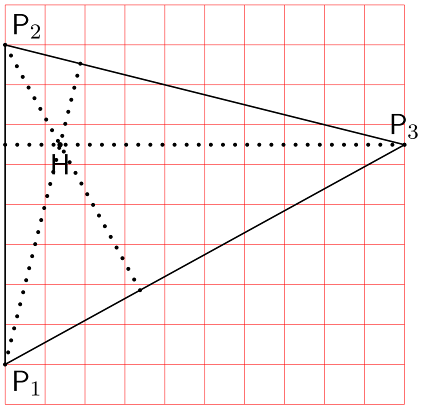


Figure 19: Determination of the orthocenter

```
\unitlength=0.008\linewidth
\begin{picture}(100,100)(0,-10)
\AutoGrid
\def\Puno{0,0}\def\Pdue{0,80}\def\Ptre{100,55}
{\linethickness{0.6pt}%
  \polygon(\Puno)(\Pdue)(\Ptre)}%
\Pbox(\Puno)[tl]{P_1}[1.5]%
\Pbox(\Pdue)[bl]{P_2}[1.5]
\Pbox(\Ptre)[bc]{P_3}[1.5]
 \TriangleBisectorBase\Puno on\Pdue and \Ptre to\Iu
 \TriangleBisectorBase\Pdue on\Ptre and \Puno to\Id
 \TriangleBisectorBase\Ptre on\Puno and \Pdue to\It
\Dotline(\Puno)(\Iu){3}[1.5]
\Dotline(\Pdue)(\Id){3}[1.5]
\Dotline(\Ptre)(\It){3}[1.5]
\IntersectionOfSegments(\Puno)(\Iu)%
  and(\Pdue)(\Id)to\C
\Pbox(\C)[t]{I}[2]
\DistanceOfPoint\C from(\Puno)(\Pdue)to\R
\Circlewithcenter\C radius\R
\end{picture}
```

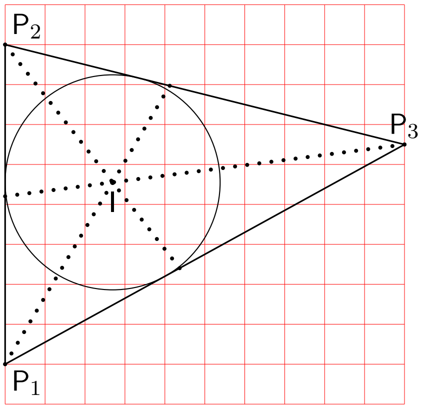


Figure 20: Determination of the incenter and of the incircle

in figure 22 with its code. The nine point are the triples of (a) the height base point if the height lines; (b) the middle points of the fraction of each height line between each vertex and the orthocenter; (c) the middle points

```
\unitlength=0.007\linewidth\centering
\begin{picture}(110,110)(-5,0)
\AutoGrid
\CopyVect20,10to\Pu \Pbox(\Pu)[t]{P_1}
\CopyVect10,90to\Pd \Pbox(\Pd)[br]{P_2}
\CopyVect100,70to\Pt \Pbox(\Pt)[l]{P_3}
{\linethickness{0.6pt}\polygon(\Pu)(\Pd)(\Pt)}%
\AxisOf\Pd and\Pu to\Mu\Du \segment(\Mu)(\Du)
\AxisOf\Pu and\Pt to\Md\Dd \segment(\Md)(\Dd)
\AxisOf\Pt and\Pd to\Mt\Dt\segment(\Mt)(\Dt)
\IntersectionOfSegments(\Mu)(\Du)and(\Md)(\Dd)to\C
\ThreePointCircle*(\Pu)(\Pd)(\Pt)
\Pbox(\C)[l]{C}[3.5]
\end{picture}
```

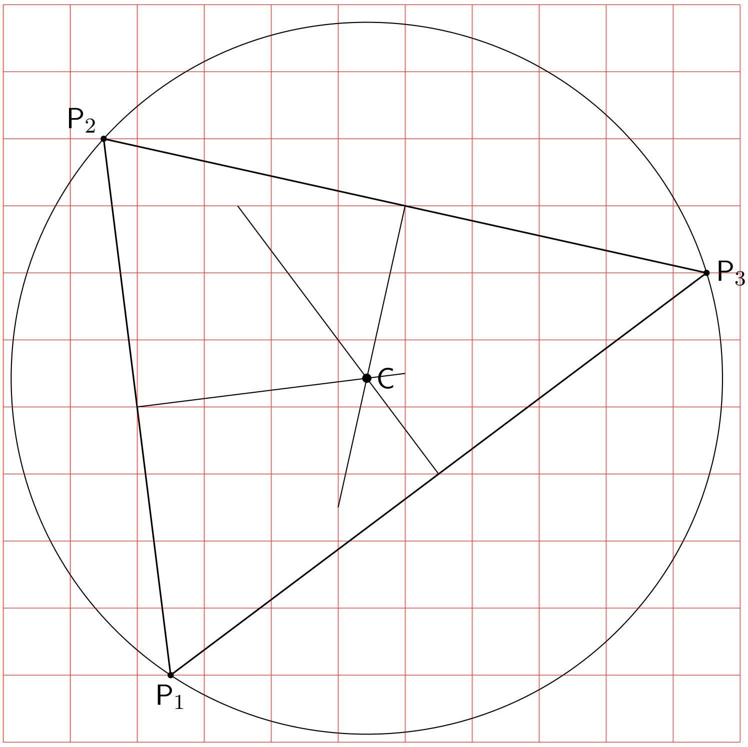


Figure 21: Determination of the circumcenter and of the circumcircle

of the sides. It goes by itself that any point triple among the three ones of the nine point circle is good to draw he final circle.

```
\unitlength=0.007\linewidth\centering
\begin{picture}(100,100)(0,-10)
\AutoGrid
\CopyVect20,0to\Pu \Pbox(\Pu)[t]{P_1}[2pt]
\CopyVect10,80to\Pd \Pbox(\Pd)[b]{P_2}[2pt]
\CopyVect100,60to\Pt \Pbox(\Pt)[b]{P_3}[2pt]
{\polygon(\Pu)(\Pd)(\Pt)\ignorespaces}
\TriangleMedianBase\Pu on\Pd and\Pt to\Pmu\Pbox(\Pmu)[bl]{M_1}[2pt]
\TriangleMedianBase\Pd on\Pt and\Pu to\Pmd\Pbox(\Pmd)[tl]{M_2}[2pt]
\TriangleMedianBase\Pt on\Pu and\Pd to\Pmt \Pbox(\Pmt)[tr]{M_3}[2pt]
 \ThreePointCircle(\Pmu)(\Pmd)(\Pmt)
 \TriangleHeightBase\Pu on\Pd and\Pt to\Phu\Pbox(\Phu)[b]{H_1}[2pt]
 \TriangleHeightBase\Pd on\Pt and\Pu to\Phd\Pbox(\Phd)[tl]{H_2}[2pt]
 \TriangleHeightBase\Pt on\Pu and\Pd to\Pht\Pbox(\Pht)[br]{H_3}[2pt]
 \segment(\Pu)(\Phu) \segment(\Pd)(\Phd) \segment(\Pt)(\Pht)
 \SubVect\Pu from\Phu to\DirHu  \SubVect\Pd from\Phd to\DirHd
 \SubVect\Pt from\Pht to\DirHt
 \IntersectionOfLines(\Pu)(\DirHu)and(\Pt)(\DirHt)to\Po
 \Pbox(\Po)[tl]{O}[2pt]
 \MiddlePointOf(\Po)(\Pu)to\Pou \Pbox(\Pou)[l]{O_1}[2pt]
 \MiddlePointOf(\Po)(\Pd)to\Pod \Pbox(\Pod)[b]{O_2}[2pt]
 \MiddlePointOf(\Po)(\Pt)to\Pot \Pbox(\Pot)[bl]{O_3}[2pt]
 \ThreePointCircle*(\Pou)(\Pod)(\Pot)
 \end{picture}
```

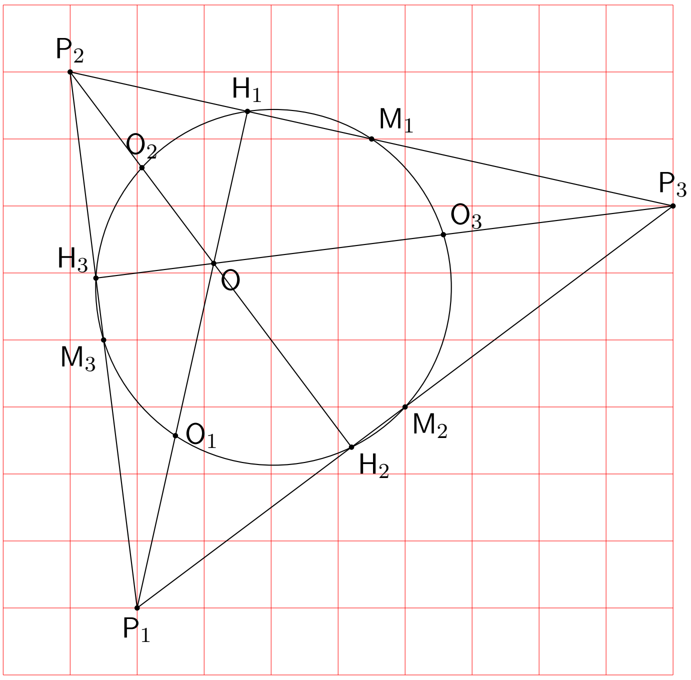


Figure 22: The nine point circle with its code

7.10 Triangles and ellipses

There are some situations where an ellipse should be inserted within a triangle with some constraints. Some examples are described in the following sub subsections.

The following examples require some new simple macros, described in the previous sections; some more more examples can be made that require more complex macros. Even these macros are just examples. For other applications it is probably necessary to add even more macros.

7.10.1 The Steiner ellipse

Let us proceed with the construction of the Steiner ellipse: given a triangle, there exists only one ellipse that is internally tangent to the side mid-points.

The geometrical construction goes on this way; suppose you have to draw the Steiner ellipse of triangle T; finding the side middle points has already been shown, but the process to build the ellipse is still to be found. So let us chose a side to work as the base of triangle T, and perform an affine shear transformation parallel to the base so as to move the vertex of triangle T, opposite to the base, on the base axis, we get another triangle T_1 that is isosceles; if it is not yet so, let us make another compression/expansion affine transformation, so as to get an equilateral triangle T_2; this last triangle is particularly simple to handle, because its Steiner ellipse reduces to its incircle. If we apply in reverse order the above transformations we get the Steiner ellipse we were looking for. The only difficult part is the affine shear transformation.

The L3 functions we already created take care of all such transformations, but with an optional asterisk we can draw the intermediate passages where triangles T_2 and T_1 have their base shifted and rotated to be horizontal, so that some translations and rotations are also necessary. Figure 23 displays the final result and the code necessary to build it.

With just the addition of an asterisk we can draw the whole geometrical construction; see figure 24

7.10.2 The tangent to an ellipse

Ellipses have many interesting properties. One that I was not able to find anywhere in the documentation I examined (of course not the totality of available documents on ellipses) except an on-line document written in Portuguese by Sergio Alvez,

`https://docplayer.com.br/345411-Elipses-inscritas-num-triangulo.html`

is the *director circumference*, literal translation of the Portuguese definition *circunfência diretriz*.

```
\unitlength=0.01\linewidth
\begin{picture}(100,110)
\AutoGrid
\SteinerEllipse(10,10)(90,20)(60,105)[2]
\end{picture}
```

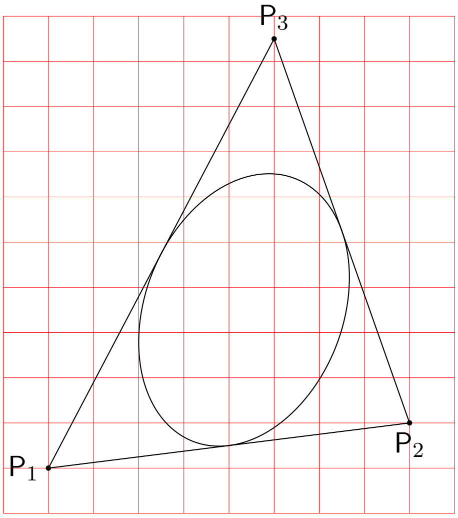


Figure 23: The Steiner ellipse of a given triangle

```
\unitlength=0.01\linewidth
\begin{picture}(100,110)(0,-10)
\AutoGrid
\SteinerEllipse*(10,10)(90,20)(60,105)[2]
\end{picture}
```

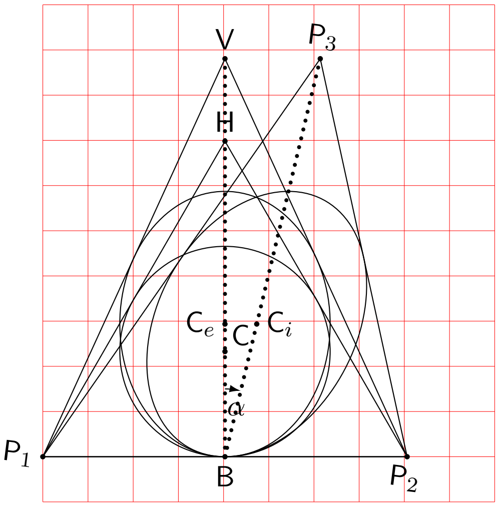


Figure 24: The construction of the Steiner ellipse of a given triangle

Consider an ellipse with its foci F en F', and a generic point P on its contour. Trace a segment from $F\,P$ and another segment for $P\,F'$; these segments measure the distances from point P to each focus: their sum is the length of the main ellipse axis $2a$, where a is the semi axis. Now lengthen the segment $P\,F'$ to point S by the length of $F\,P$; the length of $S\,F'$ is therefore equal to $2a$; now trace the circumference with center in F' and radius $2a$; this is the *director circumference*, that is labelled with $\mathit{\Gamma}$.

By construction, then, the circle γ centred in P and radius equal to $F\,P$ is tangent to $\mathit{\Gamma}$ in S and passes through F. This allows to say that:

```
\unitlength=0.005\linewidth
\begin{picture}(170,160)(-60,-80)
\AutoGrid
\VECTOR(0,-80)(0,80)\Pbox(0,80)[r]{y}[0]
\VECTOR(-60,0)(110,0)\Pbox(110,0)[t]{x}[0]
% Ellipse with given semi axes
\edef\A{40}\edef\B{30}\Xellisse{\A}{\B}[\thicklines]
% Point P on the ellipse
\edef\X{\fpeval{\A*cosd(120)}}\edef\Y{\fpeval{\B*sind(120)}}
\edef\P{\X,\Y}\Pbox(\P)[b]{P}[3]
% Foci coordinates
\edef\C{\fpeval{sqrt(\A**2-\B**2)}}
\CopyVect-\C,0 to\F \CopyVect\C,0 to\Fp\Pbox(\Fp)[t]{F’}[3]
\Pbox(\F)[t]{F}[3]\Pbox(0,0)[tr]{O}[3]
% Director circumference
\edef\Raggio{\fpeval{2*\A}}
\Circlewithcenter\Fp radius\Raggio \Pbox(80,60)[tr]{\ISOGamma{lmss}}[0]
% Tangent construction
\SegmentLength(\P)(\F)to\raggio
\Circlewithcenter\P radius\raggio \Pbox(-10,50)[bl]{\ISOgamma{lmss}}[0]
\SegmentArg(\Fp)(\P)to\Arg\AddVect\Fp and\Arg:\Raggio to\S
\segment(\Fp)(\S)\Pbox(\S)[br]{S}[3]
\segment(\F)(\S)\SegmentCenter(\F)(\S)to\M
\Pbox(\M)[r]{M}[3]\SegmentArg(\F)(\S)to\Arg
% Tangent
\edef\Arg{\fpeval{\Arg-90}}
\AddVect\M and \Arg:50 to\D \Pbox(\D)[l]{D}[3]
\segment(\M)(\D)
\end{picture}
```

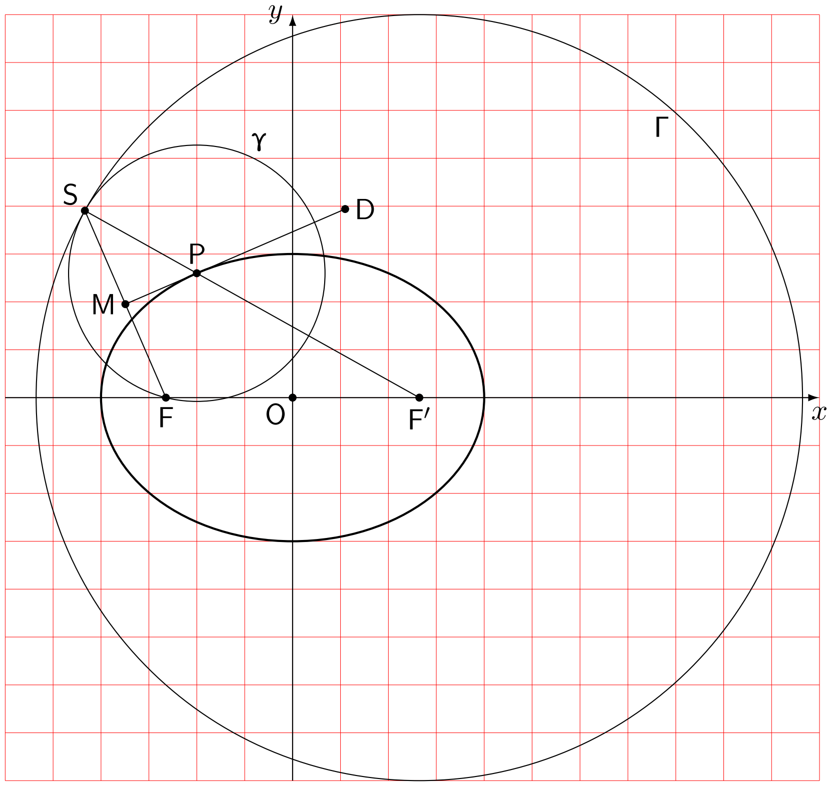


Figure 25: The director circumference

- the ellipse is the locus of the centres of all circles passing through focus F and internally tangent to the circle $\mathit{\Gamma}$ centred in the other focus F' and with radius $2a$;
- the axis of segment SF is tangent to the ellipse;
- the tangency point is the point P;
- since this axis passes through the midpoint M of segment SF and it is perpendicular to it, the segment MP determines the direction of the tangent to the ellipse;
- notice that points S and F are symmetrical with respect to the tangent in point P.

Such properties can be viewed and controlled in figure 25.

Of course the geometrical construction of figure 25 can be used also in reverse order; for example it may be given a line to play the role of the tangent, a point on this line to play the role of tangency, and a point not belonging to the line to play the role of a focus, then it is possible to find the other focus laying on a horizontal line passing through the given focus. It suffices to find the symmetrical point of the first focus with respect with the given line, and to draw a line passing through this symmetrical point and the point of tangency that intersects the horizontal line through the first focus, concluding that this is the second focus and that the ellipse major axis length is that of the segment joining this second focus with the above mentioned symmetrical point.

7.10.3 A triangle internally tangent ellipse given one of its foci

It is possible to draw an ellipse that is internally tangent to a triangle if one of its foci is specified; without this specification the problem is not definite, and the number of such ellipses is countless. But with the focus specification, just one ellipse exists with that tangency constraint. It suffices to find the other focus and at least one point of tangency, because the focal distance and the sum of distances of that tangency point from the foci, is sufficient to determine all the parameters required to draw the ellipse.

As seen in the previous subsection 7.10.2, it is sufficient to find the three symmetrical points of the given focus with respect to the three sides, i.e. the three lines that pass through the triangle vertices; and the constructions gives simultaneously the major axis length and the three tangency points, therefore all the elements required to draw the ellipse to be drawn.

The geometrical construction, with help of what has been explained in subsection 7.10.2, is easy; the steps to follow, therefore are the following:

- draw the triangle and the given focus F;
- find the symmetrical points S_i of this focus with respect to the sides of the triangle;
- use these three points S_i as the vertices of a triangle with which to draw its circumcircle that turns to be the *director circumference*; actually

```
\unitlength=0.0065\linewidth
\begin{picture}(150,150)(-30,-20)
\AutoGrid
\EllipseWithFocus%
  (10,40)(110,10)(0,110)(20,60)
\end{picture}
```

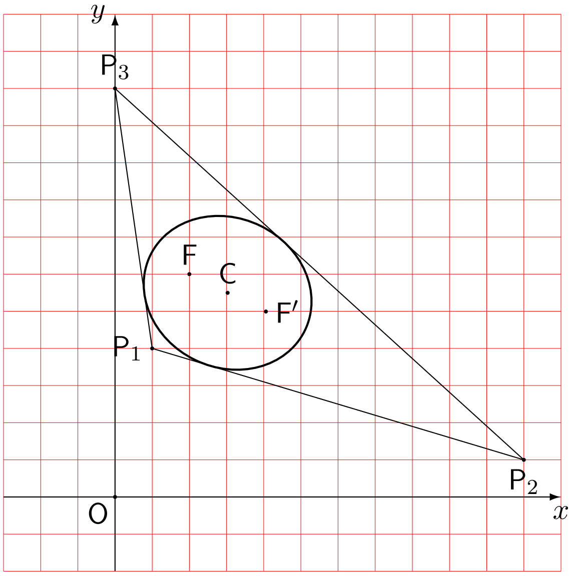


Figure 26: An ellipse internally tangent to a triangle, given a focus

only its center is of interest, because it represents the second focus F'; the inter focal distance $2c$; is just le length of vector $F' - F$;

- join with segments each symmetrical point S_i with the second focus F' and find their intersections T_i with the triangle sides; they represent the tangency points of the ellipse to be drawn;
- the radius of the director circumference is the ellipse major axis;
- equation (2) allows to find the second axis length; the segment that joins the foci has the required inclination of the main axis; its middle point is the ellipse center; therefore all necessary pieces of information to draw the ellipse are known.

Figures 26 and 27 display the construction steps and the final result.

7.11 A simple proof of Pythagoras' theorem

Euclide at his time could not use the algebra symbology we use today; he did not have even an effective way to write numbers; in his time in Greece and in the Near East the numbers were written with letters, the usual alphabetic letters; the Greek alphabet of 24 letters, was extended with three more symbols, stigma (ϛ), qoppa (ϙ) and sampi (ϡ), for a total of 27 seven symbols divided in three groups of 9 symbols. The first group represented the modern values from 1 to 9, the second group represented the values from 10 to 90, the last group represented the values from 100 to 900. therefore they could easily write numbers up to 999‘, an apex denoting the fact that the letter string meant a number, not a word. For the thousands they used the same symbols preceded by an inverted apex.so they could

```
\unitlength0.0065\linewidth
\begin{picture}(150,150)(-30,-20)
\AutoGrid
\EllipseWithFocus*%
  (10,40)(110,10)(0,110)(20,60)
\end{picture}
```

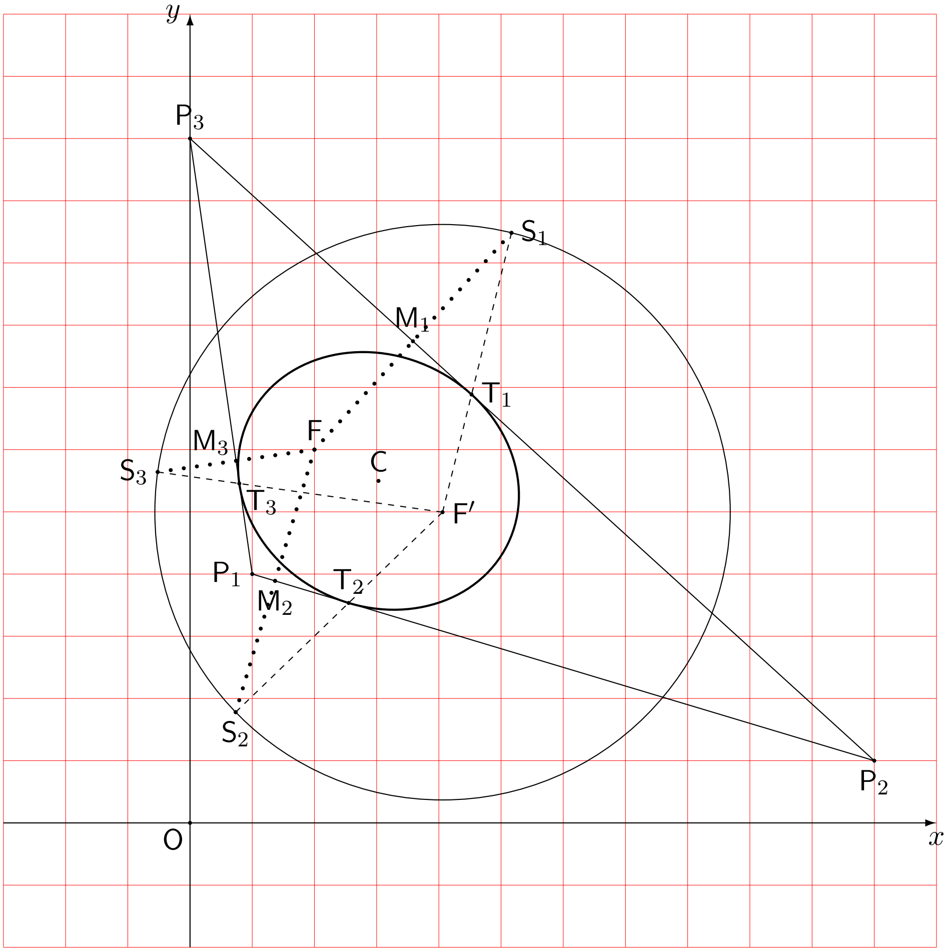


Figure 27: Construction of the ellipse of figure 26

arrive to ‚9,9,9 999‘. The same use of letters for denoting numbers, was in use also among the Phoenicians and the Palestinians; the Kabala is a practice that suggests special informations to those that can interpret such letter/number signs. But it was practically impossible to do any serious calculations, in any case no square roots. This is why geometry was the only means to make “impossible” calculations.

The “Elements” by Euclide contain many theorems and statements that are demonstrated with geometrical constructions; The demonstration of

```
\begin{minipage}{0.475\linewidth}\unitlength=0.00815\linewidth
\begin{picture}(120,120)(-10,-10)
\AutoGrid
\polygon(0,0)(0,100)(100,100)(100,0)
{\color{black!10!white}
\polygon*(0,60)(0,0)(60,0)(60,60)
\polygon*(0,60)(40,60)(40,100)(0,100)}
\linethickness{2pt}
\polygon(0,60)(0,0)(60,0)(60,60)
\polygon(0,60)(40,60)(40,100)(0,100)
\linethickness{1pt}
{\color{red}\polygon(60,0)(100,60)(60,60)
\polygon(40,60)(40,100)(100,60)
\Pbox(75,45)[cc]{T_1}[0pt]\Pbox(60,0)[t]{C_1}[3pt]
\Pbox(55,75)[cc]{T_2}[0pt]\Pbox(40,100)[b]{C_2}[3pt]}
\Pbox(20,100)[b]{a}[0pt]\Pbox(70,100)[b]{b}[0pt]
\Pbox(100,80)[l]{a}[0pt]\Pbox(100,30)[l]{b}[0pt]
\Pbox(30,0)[t]{b}[0pt]  \Pbox(80,0)[t]{a}[0pt]
\Pbox(0,30)[r]{b}[0pt]  \Pbox(0,80)[r]{a}[0pt]
\Pbox(70,80)[bl]{c}[0pt]\Pbox(80,30)[tl]{c}[0pt]
\end{picture}
\end{minipage}
\hfill
\begin{minipage}{0.475\linewidth}\unitlength=0.00815\linewidth
\begin{picture}(120,120)(-10,-10)
\AutoGrid
\polyline(60,0)(100,0)(100,100)(40,100)
\Pbox(20,100)[b]{a}[0pt]\Pbox(70,100)[b]{b}[0pt]
\Pbox(100,80)[l]{a}[0pt]\Pbox(100,30)[l]{b}[0pt]
\Pbox(30,0)[t]{b}[0pt]  \Pbox(80,0)[t]{a}[0pt]
\Pbox(0,20)[r]{a}[0pt]  \Pbox(0,70)[r]{b}[0pt]
{\color{black!10!white}\polygon*(60,0)(100,60)(40,100)(0,40)}%
{\thicklines\color{red}\polygon(60,0)(100,60)(40,100)(0,40)
\Pbox(15,80)[cc]{T_2}[0pt]\Pbox(60,0)[t]{C_1}[3pt]
\Pbox(20,15)[cc]{T_1}[0pt]\Pbox(40,100)[b]{C_2}[3pt]
\polyline(40,100)(0,100)(0,0)(60,0)}
\Pbox(70,80)[tr]{c}[0pt]\Pbox(80,30)[br]{c}[0pt]
\Pbox(20,70)[tl]{c}[0pt]\Pbox(30,20)[bl]{c}[0pt]
\end{picture}
\end{minipage}
```

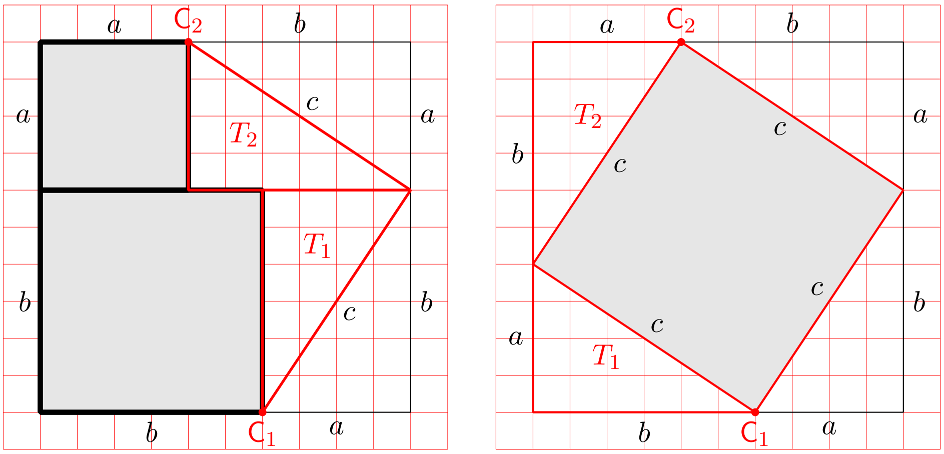


Figure 28: Two squares partitioned in differente ways

Pythagoras' theorem to my eyes appears too complicated and difficult to understand; the geometrical construction developed hereafter is particularly ingenious and simple to understand.

Assume you have a square triangle of legs a and b and hypothenuse c; the theorem states that the area of the square built on the hypotenuse equals the sum of the areas of the squares built on each leg.

Let us prepare two similar squares of equal areas partitioned in differente ways. See figure 28. In both images you see the overall squares have side lengths equal to the sum of the right triangle legs; in the left image the grey squares have side lengths equalling each right triangle leg respectively. The overall square contains also two rectangles with side lengths a and b so that the diagonal equals the right triangle hypothenuse.

These diagonals, therefore, partition the respective rectangles in two right triangles with the same dimensions of the original right triangle. In particular two of such triangles are labeled T_1 and T_2 and the vertices labeled C_1 and C_2 are emphasised. Let us now rotate both labelled triangles around their labelled vertices so as to obtain the right image, where the overall square has the same sides as the original one, although partitioned in a slightly differente way; their areas are the same; but in this second image the grey square has its side equal to the original right triangle hypothenuse; since the four right triangles contained in the overall squares of both images are equal, thus the sum of their areas are he same; therefore the grey areas in each image are the same and this means that the area of the square built on the hypotenuse equals the sum of the area of the squares built on the legs of the original right triangle. Pythagoras' theorem is proved without any use of algebra.

8 Conclusion

We have shown that the ***picture*** environment, extended with this package euclideangeometry (that takes care of loading curve2e and pict2e) can make important diagrams that certainly were not foreseen by Leslie Lamport when he first wrote the code for the initial ***picture*** environment.

The reader can easily understand that this package is far from being exhaustive for all geometrical problems to be solved with ruler and compass; it shows a way to add more commands to approach further problems; if any author, who creates new commands, would like to contribute more macros to this package, I will be happy to integrate his/her contribution to a new version of this package; depending on the contribution, I would be very happy to add its author name to this package author list; for simpler contributions each contributor will be duly acknowledged.

Creating new macros to solve more problems is pleasant; the more difficult the problem, the greater the satisfaction in solving it.

Enjoy using LaTeX and its possible applications!