# The euclideangeometry package

Claudio Beccari

claudio dot beccari at gmail dot com

Version v.0.1.3 – Last revised 2020-02-09.

## Contents



## Preface

This file contains the documented code of euclideangeometry. The user manual source file euclideangeometry-man.tex and the readable document is euclideangeometry.pdf; it should already be installed with your updated complete TEX system installation.

Please refer to the user manual before using this package.

# 1 The code

## 1.1 Checking the date of a sufficiently recent curve2e package

This package has been already identified by the commands extracted by the docstrip package, during the .dtx file compilation. In any case, if the test checks that the curve2e file date is too old; it warns the user with an emphasised error message on the console, loading this euclideangeometry package is stopped and the whole job aborts. The emphasised error message appears like this:

```
************************************************************
Package curve2e too old
Be sure that your TeX installation is complete and up to date
************************************************************
Input of euclideangeometry is stopped and job aborted
************************************************************
```

This message should be sufficiently strong in order to avoid using this package with a vintage version of TEX Live or MikTEX.

```
\RequirePackage{curve2e}
\@ifpackagelater{curve2e}{2020/01/18}{}%
 {%
  \typeout{************************************************************}
  \typeout{Package curve2e too old}
  \typeout{Be sure that your TeX installation is complete and up to date}
  \typeout{************************************************************}
  \typeout{Input of euclideangeometry stopped and job aborted}
  \typeout{************************************************************}
  \@@end
 }%

```

## 1.2 Service macros

The next code is used to typeset the GUIT logo; if necessary the `\RequirePackage` macro loads the `xspace` package; therefore these macros do not require any special terminator of the control sequence name, because the `\xspace` command takes care of the necessary space; this command avoids inserting any space if macros are followed by any reasonable sign different form a space, such as punctuation marks, parentheses, quotation marks, and so on.

```

\definecolor{verdeguit}{rgb}{0, 0.40, 0}
\def\GuIT{\mbox{\color{verdeguit}\def\I{\textcolor{black}{I}}%
  \fontfamily{lmr}\fontseries{m}\fontshape{sc}\selectfont
g\raisebox{-0.715ex}{\kern-0.26em u}\kern-0.13em\I\kern-0.14em t}\xspace}
```

The following LATEX related logos are provided so as to make them coherent with the smart final space we have discussed above. The real difference is that it is necessary to use tue `etoolbox` facilities; therefore the package is (possibly) loaded. With it we add at the end of their original definition the `\xspace` command. But what is the "original definition"? Of course `\TeX` and `\LaTeX` macros are defined within the LATEX kernel, but the user might have loaded other packages that have redefined them; therefore these patches are applied only at the preamble end, so if other definitions were provided by other packages loaded in the preamble, such redefinitions are patched. Only `\XeLaTeX` is completely redefined, because the other existing definitions work correctly only with OpenType fonts; with Type 1 fonts and oblique fonts (italics, slanted, etc.) the reversed "E" generally is inclined

in the wrong direction; by rotating it, instead of reflecting it, the “E” might not be the best, but al least slants in the right direction.

```
\RequirePackage{etoolbox}
\AfterEndPreamble{%
\DeclareRobustCommand{\TeX}{T\kern-.1667em
  \lower.5ex\hbox{E}\kern-.125emX\@\xspace}
\DeclareRobustCommand{\LaTeX}{L\kern-.36em%
  {\sbox\z@ T%
   \vbox to\ht\z@{\hbox{\check@mathfonts
                        \fontsize\sf@size\z@
                        \math@fontsfalse\selectfont
                        A}%
                  \vss}%
  }%
  \kern-.15em%
  \TeX}}

\DeclareRobustCommand*\TeXLive{\TeX\ Live\xspace}
\DeclareRobustCommand*\MikTeX{Mik\TeX}
\DeclareRobustCommand*\MacTeX{Mac\TeX}
\DeclareRobustCommand*\pdfLaTeX{pdf\/\LaTeX}
\DeclareRobustCommand*\LuaLaTeX{Lua\LaTeX}
\DeclareRobustCommand*\XeLaTeX{X\ifdim\fontdimen1\font=0pt\kern-0.15em\fi
\lower.5ex\hbox{\rotatebox[origin=c]{180}{E}}%
\ifdim\fontdimen1\font=0pt\kern-0.15em\else\kern-0.275em\fi
\LaTeX}
\DeclareRobustCommand*\TikZ{Ti\emph{k}Z\xspace}

```

Here we have a command that allows to display some framed code; it is usable also to display the syntax of some commands; and when doing this action the full range of service macros at the beginning of this section play the best of their role. The English aliases for the opening and closing environments are also provided. Attention: do not use `\\` commands to specify new lines, unless you want to produce empty/blank lines; within this environment the source code “end of line” characters are not treated as spaces, but are actually executed according to their name.

```
\newenvironment{ttsintassi}{\begin{lrbox}{0}
\minipage{\dimexpr\linewidth-2\fboxrule-2\fboxsep}\ttfamily\obeylines}%
{\endminipage\end{lrbox}\center\fbox{\box0}\endcenter}
\let\ttsyntax\ttsintassi \let\endttsyntax\endttsintassi
```

The following macro probably will migrate to `curve2e`; meanwhile this redefinition is useful in order to save some input and to make a more flexible macro. We used it virtually in every picture we inserted in the user manual of this package; of course the user does not need to have any drawing superimposed onto a red grid. Actually the grid is useful while drawing; when the image is complete, the grid command may be deleted or its line commented out. The purpose of this macro is to avoid the `\put` command to place the grid; and since its reference point in the

original definition is arbitrary, the connection with real coordinates is very small and might become confusing or of little help. Now the syntax is the following:

| `\(GraphGrid)(`⟨*reference point or dimensions*⟩`)(`⟨*overall dimensions*⟩`)` |
|---|

where the second argument is optional; if it is missing, the first argument contains the ⟨*overall dimensions*⟩, otherwise it contains the lower left corner coordinate that represent the ⟨*reference point*⟩ that an internal `\put` command will use to place the grid; if no ⟨*reference point*⟩ is specified, the reference point is `0,0`, and the grid is `\put` with its lower left corner in the origin of the `picture` coordinates.

```
48
49 \unless\ifcsname Gr@phGrid\endcsname
50 \let\originalGraphGrid\GraphGrid
51 \RenewDocumentCommand\GraphGrid{r() d()}{%
52 \IfValueTF{#2}{\put(#1){\originalGraphGrid(#2)}}%
53               {\put(0,0){\originalGraphGrid(#1)}}}\fi
```

## 1.3 Labelling

While doing any graphical geometrical drawing it is necessary to label points, lines, angles and other such items. Non measurable labels should be in upright sans serif font, according to the ISO regulations, but here we are dealing witt point identified by macros the contain their (cartesian or polar) coordinates that very often are both labels and math variables.

Here we provide a versatile macro that can do several things. Its name is `\Pbox` and it produces a box containing the label in math format. By default the point label is typeset with the math font variant produced by command `\mathsf`, but the macro is sufficiently versatile to allow other settings; It accepts several optional arguments, therefore it syntax is particular:

| `\Pbox(`⟨*coordinates*⟩`)[`⟨*alignment*⟩`]{`⟨*label*⟩`}[`⟨*diameter*⟩`]`⟨⋆⟩`<`⟨*angle*⟩`>` |
|---|

where ⟨*coordinates*⟩ are the coordinates where to possibly set a black dot with the specified ⟨*diameter*⟩; in any case it is the reference point of the ⟨*label*⟩; the ⟨*alignment*⟩ is formed by the usual letters `t, b, c, l, r` that can be paired in a coherent way (for example the couple `tb` is evidently incoherent, as well as `lr`), but in absence of this optional specification, the couple `cc` is assumed; most often than not, the label position becomes such that when the user reviews the document drafts, s/he understands immediately that s/he forgot to specify some reasonable ⟨*alignment*⟩ codes. Think of the ⟨*alignment*⟩ letters as the position of the reference point with respect to the the ⟨*label*⟩ optical center. The optional ⟨*angle*⟩ argument produces a rotation of the whole label by that angle; it may be used in several circumstances, especially when the label is just text, to produce, for example, a sideways legend. It is useful also when the labels are produced within a rotated box, in order to counterrotate them.

The optional asterisk draws a frame around the *label*. Notice that the separator between the visible or the invisible frame and the box contents varies according

the the fact the the ⟨*alignment*⟩ specification contains just one or two letter codes; this is useful, because the diagonal position of the label should be optically equal to the gap that exists between the reference point and the ⟨*label*⟩ box.

If the ⟨*diameter*⟩ is zero, no dot is drawn, the whole ⟨*label*⟩ is typeset with the `\mathit` math font; otherwise only the first symbol of a math expression si typeset in sans serif. The presence of subscripts makes the labels appear more distant from their reference point; the same is true when math symbols, even without subscripts, are used, because of the oblique nature of the math letters alphabet.

If some text has to be printed as a label, it suffices to surround it with dollar signs, that switch back to text mode when the default mode is the math one. With this kind of textual labels it might be convenient to use the optional asterisk to frame the text.

```
\providecommand\Pbox{}
\newlength\PbDim
\RenewDocumentCommand\Pbox{D(){0,0} O{cc} m O{0.5ex} s D<>{0}}{%
\put(#1){\rotatebox{#6}{\makebox(0,0){%
\settowidth\PbDim{#2}%
\edef\Rapp{\fpeval{\PbDim/{1ex}}}%
\fptest{\Rapp > 1.5}{\fboxsep=0.5ex}{\fboxsep=0.75ex}%
\IfBooleanTF{#5}{\fboxrule=0.4pt}{\fboxrule=0pt}%
\fptest{#4 = 0sp}%
  {\makebox(0,0)[#2]{\fbox{$\relax#3\relax$}}}%
  {\edef\Diam{\fpeval{(#4)/\unitlength}}%
   \makebox(0,0){\circle*{\Diam}}%
   \makebox(0,0)[#2]{\fbox{$\relax\mathsf#3\relax$}}%
  }}}%
}\ignorespaces}
```

The following command, to be used always within a group, or a environment or inside a box, works only with piecewise continuously scalable font collection, such as, for example, the Latin Modern fonts, or with continuously scalable fonts, such as, for example, the Times ones. They let the operator select, for the scope of the command ,any size, even fractional so as to fine adjust the text width in the space allowed for it; it is particularly useful with the monospaced fonts, that forbid hyphenation, and therefore cannot be adjusted to the current line width.

```
\DeclareRobustCommand\setfontsize[2][1.2]{%
  \linespread{#1}\fontsize{#2}{#2}\selectfont}
```

With OpenType fonts there should not be any problems even with math fonts; with Type 1 fonts the only scalable fonts I know of, are the LibertinusMath fonts, usable through the LibertinusT1math package, are also the only ones that have 8 bit encoded math fonts (256 glyph fonts), while the standard default Type 1 math fonts are just 7 bit encoded (128 glyphs fonts).

Another useful labelling command is `Zbox`; this command is an evolution of a command that I been using for years in several documents of mine. It uses some general text, not necessarily connected to a particular point of the `picture` environment, as a legend; It can draw short text as a simple horizontal box, and

longer texts as a vertical box of specified width and height

Is syntax is the following:

| `\Zbox(`⟨*position*⟩`)(`⟨*(*⟩`)dimensions)[`⟨*alignment*⟩`]{`⟨*text*⟩`}` |
|---|

where ⟨*position*⟩ is where the reference point of the box has to be put in the picture; ⟨*dimensions*⟩ are optional; if not specified, the box is a horizontal one, and it is as wide as its contents; if it is specified, it must be a comma separated list of two integer or fractional numbers that are the width and the height of the box; if the height is specified as zero, the width specifies a horizontal box of that width; ⟨*alignment*⟩ is optional and is formed by one or two coherent letter codes from the usual set `t, b, c, l, r`; if the ⟨*alignment*⟩ is absent, the default alignment letters are `bl`, i.e. the box reference point is the bottom left corner; ⟨*text*⟩ contains general text, even containing some math.

```
\def\EUGsplitArgs(#1,#2)#3#4{\edef#3{#1}\edef#4{#2}}
\newlength\EUGZbox
\providecommand\Zbox{}
\RenewDocumentCommand\Zbox{r() D(){0,0} O{bl} m}{%
\EUGsplitArgs(#2)\ZboxX\ZboxY % splits box dimensions
\fboxsep=2\unitlength
\ifnum\ZboxX=\z@
  \def\ZTesto{\fbox{#4}}%
\else
  \ifnum\ZboxY=\z@
    \def\ZTesto{\fbox{\parbox{\ZboxX\unitlength}{#4}}}%
  \else
    \def\ZTesto{%
    \setbox\EUGZbox=\hbox{\fbox{%
         \parbox[c][\ZboxY\unitlength][c]{\ZboxX\unitlength}{#4}}}%
      \dimen\EUGZbox=\dimexpr(\ht\EUGZbox +\dp\EUGZbox)/2\relax
      \ht\EUGZbox=\dimen\EUGZbox\relax
      \dp\EUGZbox=\dimen\EUGZbox\relax
      \box\EUGZbox%
      }%
  \fi
\fi
\put(#1){\makebox(0,0)[#3]{\ZTesto}}\ignorespaces}
```

## 1.4 Service macros for ellipses

The `\ellisse` has a control sequence name in Italian; it differs for just one letter from the name `ellipse` English name, but we cannot use the latter one because it may conflict with other packages loaded by the user; actually this command and the next one are just shortcuts for executing more general commands with specific sets of arguments. For details and syntax, please refer yourself to section 1.7

```
\NewDocumentCommand\ellisse{ s m m}{%
```

```
\IfBooleanTF{#1}%
  {\let\fillstroke\fillpath}%
  {\let\fillstroke\strokepath}%
\Sellisse{#2}{#3}%
}

\NewDocumentCommand\Xellisse{ s D(){0,0} O{0} m m O{} o}{%
\IfBooleanTF{#1}%
  {\XSellisse*(#2)[#3]{#4}{#5}[#6][#7]}%
  {\XSellisse(#2)[#3]{#4}{#5}[#6][#7]}%
}
```

We do not know if the following macro `\polyvector` may be useful for euclidean geometry constructions, but it may be useful in block diagrams; it is simply a polyline where the last segment is a geometrical vector. As in polyline the number of recursions is done until the last specified coordinate pair; recognising that it is the last one, instead of drawing a segment, the macro draws a vector.

```

\def\polyvector(#1){\roundcap\def\EUGpreviouspoint{#1}\EUGpolyvector}
\def\EUGpolyvector(#1){%
\@ifnextchar({%
    \segment(\EUGpreviouspoint)(#1)\def\EUGpreviouspoint{#1}\EUGpolyvector}%
  {\VECTOR(\EUGpreviouspoint)(#1)}%
}
```

## 1.5 Processing lines and segments

The next macros are functional for the geometric constructions we are going to make: finding the intersection of lines or segments, finding the lengths and arguments of segments, directions, distances, distance of a point from a line or a segment, the symmetrical point of a another one specified with respect to a given center of symmetry; the axes of segments, the solutions of the relationship between the semi axes of an ellipse and the semi focal distance, and so on.

Most of these commands have delimited arguments; the delimiters may be the usual parentheses, but they may be keywords; many commands contain the keyword `to`, not necessarily the last one; the arguments before such keyword may be entered as ordered comma separated numerical couples, or comma separated macros the containing scalar values; or they may be macros that contain the ordered couples representing vectors or directions; they all may be in cartesian or polar form. Remember that such ordered couples are complex numbers, representable by vectors applied to the origin of the axes; therefore sometimes it is necessary that the underlying commands execute some vector differences so as to work with generic vectors.

On the opposite the output values, i.e. the argument after that `to` keyword, should be tokens that can receive a definition, in general macros, to which the user should assign a mnemonic name; s/he should use such macros for further computations or for drawing commands.

The first and principal command is `\IntersectionOfLines` and it has the following syntax:

`\IntersectionOfLines(`⟨*point1*⟩`)(`⟨*dir1*⟩`)and(`⟨*point2*⟩`)(`⟨*dir2*⟩`)to`⟨*crossing*⟩

where ⟨*point1*⟩ and ⟨*dir1*⟩ are respectively a point of the first line and its *direction*, not a second point, but the *direction* — it is important to stress this point; similarly for the second line; the output is stored in the macro that identifies the ⟨*crossing*⟩ point. The directions do not need to be expressed with unit vectors, but the lines must not be parallel or anti parallel (equal directions or differing by 180°); the macro contains a test that checks this anomalous situation because an intersection at infinity or too far away ($2^{14} - 1$ typographical points, approximately 5,758 m) is of no interest; in case, no warning message is issued, the result is put to `0,0`, and the remaining computations become nonsense. It is a very unusual situation and I never encountered it; nevertheless...

```
\def\IntersectionOfLines(#1)(#2)and(#3)(#4)to#5{\bgroup
\def\IntPu{#1}\def\Uu{#2}\def\IntPd{#3}\def\Ud{#4}%
 \DirOfVect\Uu to\Du
 \DirOfVect\Ud to\Dd
 \XpartOfVect\Du to \a  \YpartOfVect\Du to \b
 \XpartOfVect\Dd to \c  \YpartOfVect\Dd to \d
 \XpartOfVect\IntPu to \xu \YpartOfVect\IntPu to \yu
 \XpartOfVect\IntPd to \xd \YpartOfVect\IntPd to \yd
 \edef\Den{\fpeval{-(\a*\d-\b*\c)}}%
 \fptest{abs(\Den)<1e-5}{% Almost vanishing determinant
   \def#5{0,0}%
 }{% Determinant OK
   \edef\Numx{\fpeval{(\c*(\b*\xu-\a*\yu)-\a*(\d*\xd-\c*\yd))/\Den}}%
   \edef\Numy{\fpeval{(\d*(\b*\xu-\a*\yu)-\b*(\d*\xd-\c*\yd))/\Den}}%
   \CopyVect\Numx,\Numy to\Paux
   \edef\x{\egroup\noexpand\edef\noexpand#5{\Paux}}\x\ignorespaces}}
```

The `IntersectionOfSegments` macro is similar but in input it contains the end points of two segments: internally it uses `\IntersectionOfLines` and to do so it has to determine the directions of both segments. The syntax is the following:

`\IntersectionOfSegments(`⟨*point11*⟩`)(`⟨*point12*⟩`)and(`⟨*point21*⟩`)(`⟨*point22*⟩`)`
`to`⟨*crossing*⟩

The ⟨*crossing*⟩ point might fall outside one or both segments. It is up to the users to find out if the result is meaningful or nonsense. Two non parallel lines are infinitely long in both directions and any ⟨*crossing*⟩ point is acceptable; with segments the situation might become nonsense.

```
\def\IntersectionOfSegments(#1)(#2)and(#3)(#4)to#5{%
\SubVect#1from#2to\IoSvectu \DirOfVect\IoSvectu to\DirIoSVecu
\SubVect#3from#4to\IoSvectd \DirOfVect\IoSvectd to\DirIoSVecd
\IntersectionOfLines(#1)(\DirIoSVecu)and(#3)(\DirIoSVecd)to#5\ignorespaces}
```

An application of the above intersections is formed by the next two macros; they find the axes of a couple of sides of a triangle and use their base point and direction to identify two lines the intersection of which is the circumcenter; the distance of one base point from the circumcenter is the radius of the circumcircle that can be drawn with the usual macros. We have to describe the macros `\AxisOf` and `CircleWithCenter` and we will do it in a little while. Meanwhile the syntax of the whole macro is the following:

`\ThreePointCircle`⟨⋆⟩`(`⟨*vetex1*⟩`)(`⟨*vertex2*⟩`)(`⟨*vertex3*⟩`)`

where the three vertices are the three points where the circle must pass, but they identify also a triangle. Its side axes intersect in one point that by construction is at the same distance from the three vertices, therefore it is the center of the circle that passes through the three vertices. A sub product of the computations is the macro `\C` that contains the center coordinates. If the optional asterisk is used the whole drawing is executed, while if it is missing, only the `\C` macro remains available but the user is responsible to save/copy its value into another macro; for this reason another macro should be more easy to use; its syntax is the following:

`\ThreePointCircleCenter(`⟨*vetex1*⟩`)(`⟨*vertex2*⟩`)(`⟨*vertex3*⟩`)`
`to`⟨*center*⟩

where the vertices have the same meaning, but⟨*center*⟩ is the user chosen macro that contains the center coordinates.

```
\NewDocumentCommand\ThreePointCircle{s r() r() r()}{%
\AxisOf#2and#3to\Mu\Du \AxisOf#2and#4to\Md\Dd
\IntersectionOfLines(\Mu)(\Du)and(\Md)(\Dd)to\C
\SubVect#2from\C to\R
\IfBooleanTF{#1}{\CircleWithCenter\C Radius\R}{}\ignorespaces}

\NewDocumentCommand\ThreePointCircleCenter{r() r() r() m}{%
\ThreePointCircle(#1)(#2)(#3)\CopyVect\C to#4}
```

There are some useful commands that help creating `picture` diagrams in an easier way; for example one of the above described commands internally uses `\CircleWithCenter`. It is well known that the native `picture` command `\circle` requires the specification of the diameter but many `euclideangeometry` commands already get the distance of two points, or the magnitude of a segment, or similar objects that may be used as a radius, rather than the diameter; why should we not have macros that simultaneously compute the require diameter and draw the circle. Here there are two such macros; they are similar to one another but their names differ in capitalisation, but also in the way they use the available input information. The syntax is the following:

`\CircleWithCenter`⟨*center*⟩ `Radius`⟨*Radius*⟩
`\Circlewithcenter`⟨*center*⟩ `radius`⟨*radius*⟩

where in both cases ⟨*center*⟩ is a vector/ordered couple that points to the circle center. On the contrary ⟨*Radius*⟩ is a vector obtained through previous calculations, while ⟨*radius*⟩ is a scalar containing a previously calculated length.

```
\def\CircleWithCenter#1Radius#2{\put(#1){\ModOfVect#2to\CWR
\circle{\fpeval{2*\CWR}}}\ignorespaces}
%
\def\Circlewithcenter#1radius#2{\put(#1){\circle{\fpeval{2*abs(#2)}}}%
\ignorespaces}
```

As announced, here we have a macro to compute the axis of a segment; given two points $P_1$ and $P_2$, for example the end points of a segment, or better the end point of the vector that goes from $P_1$ to $P_2$, the macro determines the segment middle point and a second point the lays on the perpendicular at a distance equal to half the first two points distance; this second point lays at the left of vector $P_2 - P_1$, therefore it is important to select the right initial vector, in order to have the second axis point on the desired side.

`\AxisOf`⟨*P1*⟩ `and`⟨*P2*⟩ `to`⟨*Axis1*⟩⟨*Axis2*⟩

Macros `\SegmentCenter` and `\MiddlePointOf` are alias of one another; their syntax is:

`\SegmentCenter(`⟨*P1*⟩`)(`⟨*P2*⟩`)to`⟨*center*⟩
`\MiddlePointOf(`⟨*P1*⟩`)(`⟨*P2*⟩`)to`⟨*center*⟩

⟨*P1*⟩, ⟨*p2*⟩ and ⟨*center*⟩ are all vectors.

```

\def\AxisOf#1and#2to#3#4{%
\SubVect#1from#2to\Base \ScaleVect\Base by0.5to\Base
\AddVect\Base and#1to#3 \MultVect\Base by0,1to#4}

\def\SegmentCenter(#1)(#2)to#3{\AddVect#1and#2to\Segm
\ScaleVect\Segm by0.5to#3\ignorespaces}

\let\MiddlePointOf\SegmentCenter
```

## 1.6 Triangle special points

Here we have the macros to find the special points on a triangle side that are the "foot" of special lines from one vertex to the opposite side. We already described the circumcircle and the circumcenter, but that is a separate case, because the circumcenter is not the intersection of special lines from one vertex to the opposite base. The special lines we are interested in here are the height, the median, and the bisector The macros have the same aspect `\Triangle...Base`, where the dots are replaced with each of the (capitalised) special line names. Their syntaxes are therefore very similar:

`\TriangleMedianBase`⟨*vertex*⟩ `on`⟨*base1*⟩ `and`⟨*base2*⟩ `to`⟨*M*⟩
`\TriangleHeightBase`⟨*vertex*⟩ `on`⟨*base1*⟩ `and`⟨*base2*⟩ `to`⟨*H*⟩
`\TrinagleBisectorBase`⟨*vertex*⟩ `on`⟨*base1*⟩ `and`⟨*base2*⟩ `to`⟨*B*⟩

where ⟨*vertex*⟩ contains one of the vertices coordinates, and ⟨*base1*⟩ and ⟨*base2*⟩ are the end points of the side opposite to that triangle vertex; ⟨$M$⟩, metaH, and ⟨$B$⟩ are the intersections of these special lines from the ⟨*vertex*⟩ to the opposite side; in order, they are the foot of the median, the foot of the height; the foot of the bisector. The construction of the median foot ⟨$M$⟩ is trivial because this foot is the base center; the construction of the height foot is a little more complicated, because it is necessary to find the exact direction of the perpendicular from the vertex to the base in order to find the intersection ⟨$H$⟩; the construction of the bisector base implies finding the exact direction of the two sides starting at the ⟨*vertex*⟩, and taking the mean direction, which is trivial if polar coordinates are used; at this point the bisector line is completely determined and the intersection with the base line ⟨$B$⟩ is easily obtained.

```
\def\TriangleMedianBase#1on#2and#3to#4{%
\SubVect#1from#2to\TMBu \SubVect#1from#3to\TMBd
\SubVect\TMBu from\TMBd to\Base
\ScaleVect\Base by0.5to\TMBm\AddVect#2and\TMBm to#4\ignorespaces}
%
\def\TriangleHeightBase#1on#2and#3to#4{%
\SubVect#2from#3to\Base
\ArgOfVect\Base to\Ang \CopyVect\fpeval{\Ang+90}:1 to\Perp
\IntersectionOfLines(#1)(\Perp)and(#2)(\Base)to#4\ignorespaces}
%
\def\TriangleBisectorBase#1on#2and#3to#4{%
\SubVect#2from#1to\Luno \SubVect#3from#1to\Ldue
\SubVect#2from#3to\Base
\ArgOfVect\Luno to\Arguno \ArgOfVect\Ldue to\Argdue
\edef\ArgBis{\fpeval{(\Arguno+\Argdue)/2}}%
\CopyVect \ArgBis:1to \Bisect
\IntersectionOfLines(#2)(\Base)and(#1)(\Bisect)to#4\ignorespaces}
```

Having defined the previous macros, it becomes very easy to create the macros to find the*barycenter*, the *orthocenter*, the*incenter*; for the *circumcenter* and the *circumcircle* we have already solved the question with the `\ThreePointCircleCenter` and the `ThreePointCircle` macros; for homogeneity, we create here their aliases with the same form as the new "center" macros. Actually, for the "circle" macros, once the center is known, there is no problem with the circumcircle, while for the incircle it suffices a macro to determine the distance of the incenter from one of the triangle sides; such a macro is going to be defined in a little while; it is more general than simply to determine the radius of the incircle.

```
\let\TriangleCircumcenter\ThreePointCircleCenter
\let\TriangleCircummcircle\ThreePointCircle
```

The other "center" macros are the following; they all consist in finding two of the specific triangle lines, and finding their intersection. Therefore for the barycenter we intersect two median lines; for the orthocenter we intersect two height lines; for the incenter we intersect two bisector lines;

```
\def\TriangleBarycenter(#1)(#2)(#3)to#4{%
\TriangleMedianBase#1on#2and#3to\Pa
\TriangleMedianBase#2on#3and#1to\Pb
\DistanceAndDirOfVect#1minus\Pa to\ModPa and\AngPa
\DistanceAndDirOfVect#2minus\Pb to\ModPb and\AngPb
\IntersectionOfLines(#1)(\AngPa)and(#2)(\AngPb)to#4}

\def\TriangleOrthocenter(#1)(#2)(#3)to#4{%
\TriangleHeightBase#1on#2and#3to\Pa
\TriangleHeightBase#2on#3and#1to\Pb
\DistanceAndDirOfVect#1minus\Pa to\ModPa and\AngPa
\DistanceAndDirOfVect#2minus\Pb to\ModPb and\AngPb
\IntersectionOfLines(#1)(\AngPa)and(#2)(\AngPb)to#4}

\def\TriangleIncenter(#1)(#2)(#3)to#4{%
\TriangleBisectorBase#1on#2and#3to\Pa
\TriangleBisectorBase#2on#3and#1to\Pb
\DistanceAndDirOfVect#1minus\Pa to\ModPa and\AngPa
\DistanceAndDirOfVect#2minus\Pb to\ModPb and\AngPb
\IntersectionOfLines(#1)(\AngPa)and(#2)(\AngPb)to#4}
```

## 1.7 Other specific service macros

And here it comes the general macro to determine the distance of a point from a segment or from a line that contains that segment; it may be used for determining the radius of the incenter, but it is going to be used also for other purposes. Its syntax is the following:

| `\DistanceOfPoint`⟨*point*⟩ `from(`⟨*P1*⟩`)(`⟨*P2*⟩`)to`⟨*distance*⟩ |
|---|

where ⟨*point*⟩ is a generic point; ⟨*P1*⟩ and ⟨*P2*⟩ are a segment end points, or two generic points on a line; ⟨*distance*⟩ is the macro that receives the computed scalar distance value.

```
\def\DistanceOfPoint#1from(#2)(#3)to#4{%
\SubVect#2from#3to\Base \MultVect\Base by0,1to\AB
\IntersectionOfLines(#1)(\AB)and(#2)(\Base)to\D
\SubVect#1from\D to\D
\ModOfVect\D to#4}
```

The following macros are specific to solve other little geometrical problems that arise when creating more complicated constructions.

The `\AxisFromAxisAndFocus` is an unhappy name that describes the solution of an ellipse relationship between the ellipse axes and the focal distance

$$a^2 = b^2 + c^2 \tag{1}$$

This relation exists between the "semi" values, but it works equally well with the full values. Evidently $a$ is the largest quantity and refers to the main ellipse

axis, the one that passes through the two foci; $b$ refers to the other shorter ellipse axis and $c$ refers to the foci; $b$ and $c$ are smaller than $a$, but there is no specific relationship among these two quantities It goes by itself that these statements apply to a veritable ellipse, not to a circle, that is the special case where $b = a$ and $c = 0$.

Since to solve the above equation we have one unknown and two known data, but we do not know what they represent, we have to assume some relationship exist between the known data; therefore if $a$ is known it must be entered as the first macro argument; otherwise $a$ is the unknown and the first Argument has to be the smaller one among $b$ and $c$. Since $b$ and$c$ may come from other computation the user has a dilemma: which is the smaller one? But this is a wrong approach; of course if the user knows which is the smaller, s/he can use the macro by entering the data in the proper order; but the user is determining the main axis, therefore it better that s/he uses directly the second macro `\MainAxisFromAxisAndFocus` that directly computes $a$ disregarding the order with which $b$ and $c$ are entered; the macro name suggests to enter $b$ first and $c$ second, but it is irrelevant thanks to the sum properties. Summarising:

- if the main axis is known use `\AxisFromAxisAndFocus` by entering the main axis as the first argument; otherwise
-
  - if it is known which is smaller among $b$ and $c$, it is possible to use `\AxisFromAxisAndFocus` by entering the smaller one as the first argument; otherwise
  - determine the main axis by using `\MainAxisFromAxisAndFocus`

Their syntaxes of these two commands are basically the following:

`\AxisFromAxisAndFocus`⟨*main axis*⟩ `and`⟨*axis or focus*⟩ `to`⟨*focus or axis*⟩
`\MainAxisFromAxisAndFocus`⟨*axis or focus*⟩ `and`⟨*focus or axis*⟩ `to`⟨*main axis*⟩

but it is possible to enter the data in a different way with the first command; the described syntax is the suggested one. Evidently ⟨*axis or focus*⟩ and ⟨*focus or axis*⟩ imply that if you specify the focus in one of the two, you have to specify the axis in the other one.

```
\def\AxisFromAxisAndFocus#1and#2to#3{%
\fptest{abs(#1)>abs(#2)}%
   {\edef#3{\fpeval{sqrt(#1**2-#2**2)}}}%
   {\edef#3{\fpeval{sqrt(#2**2+#1**2)}}}}

\def\MainAxisFromAxisAndFocus#1and#2to#3{%
\edef#3{\fpeval{sqrt(#2**2+#1**2)}}}
```

The following macros allow to determine some scalar values relative to segments; in the second one the order of the segment end points is important, because the computed argument refers to the vector $P_2 - P_1$. Their syntaxes are the following:

| |
|---|
| `\SegmentLength(`⟨*P1*⟩`)(`⟨*P2*⟩`)to`⟨*length*⟩<br>`\SegmentArg(`⟨*P1*⟩`)(`⟨*P2*⟩`)to`⟨*argument*⟩ |

Both ⟨*length*⟩ and ⟨*argument*⟩ are macros that contain scalar quantities; the argument is in the range $-180° < \Phi \leq +180°$.

```
\def\SegmentLength(#1)(#2)to#3{\SubVect#1from#2to\Segm
\ModOfVect\Segm to#3}

\def\SegmentArg(#1)(#2)to#3{\SubVect#1from#2to\Segm
\GetCoord(\Segm)\SegmX\SegmY\edef#3{\fpeval{atand(\SegmY,\SegmX)}}%
\ignorespaces}
```

In the following sections we need some transformations, in particular the affine shear one. The macros we define here are not for general use, but are specific for the purpose of this package.

The fist macro shears a segment, or better a vector that goes from point $P_1$ to point $P_2$ with a horizontal shear factor/angle $\alpha$; the origin of the vector does not vary and remains $P_1$ but the arrow tip of the vector is moved according to the shear factor; in practice this shearing macro is valid only for vectors that start from any point laying on the $x$ axis. The shear factor $\alpha$ is the angle of the *clock wise* rotation vector operator by which the vertical coordinate lines get rotated with respect to their original position. The syntax is the following:

| |
|---|
| `\ShearVect(`⟨*P1*⟩`)(`⟨*P2*⟩`)by`⟨*shear*⟩ `to`⟨*vector*⟩ |

where ⟨*P1*⟩ and ⟨*P2*⟩ are the initial and final points of the vector to be sheared with the ⟨*shear*⟩ angle, and the result is put in the output ⟨*vector*⟩

```
\def\ShearVect(#1)(#2)by#3to#4{%
\SubVect#1from#2to\AUX
\GetCoord(\AUX)\Aux\Auy
\edef\Aux{\fpeval{\Aux + #3*\Auy}}%
\edef\Auy{\fpeval{\Auy}}%
\AddVect\Aux,\Auy and#1to#4\ignorespaces}

```

Again we have another different `\ScaleVector` macro that takes in input the starting and ending points of a vector, and scales the vector independently of the initial point.

```
\def\ScaleVector(#1)(#2)by#3to#4{%
% Scala per il fattore #3 il vettore da #1 a #2
\SubVect#1from#2to\AUX
\ScaleVect\AUX by#3to\AUX
\AddVect\AUX and#1to#4\ignorespaces}
```

The following macro to draw a possibly sheared ellipse appears complicated; but in reality it is not much different from a “normal” ellipse drawing command. In oder to do the whole work the ellipse center is set in the origin of the axes,

therefore it is not altered by the shearing process; everything else is horizontally sheared by the shear angle $\alpha$. In particular the 12 nodes and control point that are required by the Bézier splines that draw the four ellipse quarters. It is this multitude of shearing commands that makes the macro mach longer and apparently complicated. The syntax is the following:

| \Sellisse⟨⋆⟩{⟨*h-axis*⟩}{⟨*v-axis*⟩}[⟨*shear*⟩] |
|---|

where the optional asterisk is used to mark and label the Bézier spline nodes and the control points of the possibly sheared ellipse; without the asterisk the ellipse is drawn without any "decoration"; the optional ⟨*shear*⟩ is as usual the angle of the sheared vertical coordinate lines; its default value is zero.

```
%
\NewDocumentCommand\Sellisse{s m m O{0}}{\bgroup
\CopyVect#2,#3to\Ptr \ScaleVect\Ptr by-1to\Pbl
\CopyVect#2,-#3to\Pbr \ScaleVect\Pbr by-1to\Ptl
\edef\Ys{\fpeval{tand{#4}}}%
\edef\K{\fpeval{4*(sqrt(2)-1)/3}}%
%
\ShearVect(0,0)(0,#3)by\Ys to\Pmt
\ShearVect(0,0)(0,-#3)by\Ys to\Pmb
\ShearVect(0,0)(#2,0)by\Ys to\Pmr
\ShearVect(0,0)(-#2,0)by\Ys to\Pml
%
\ShearVect(\Pmr)(\Ptr)by\Ys to\Ptr
\ShearVect(\Pml)(\Ptl)by\Ys to\Ptl
\ShearVect(\Pmr)(\Pbr)by\Ys to\Pbr
\ShearVect(\Pml)(\Pbl)by\Ys to\Pbl
%
\IfBooleanTF{#1}{\Pbox(\Ptr)[bl]{P_{tr}}\Pbox(\Pbl)[tr]{P_{bl}}%
\Pbox(\Pbr)[tl]{P_{br}}\Pbox(\Ptl)[br]{P_{tl}}%
\polygon(\Pbr)(\Ptr)(\Ptl)(\Pbl)}{}%
%
\ScaleVector(\Pmr)(\Ptr)by\K to\Crt
\ScaleVector(\Pmr)(\Pbr)by\K to\Crb
\ScaleVector(\Pml)(\Ptl)by\K to\Clt
\ScaleVector(\Pml)(\Pbl)by\K to\Clb
\ScaleVector(\Pmt)(\Ptr)by\K to\Ctr
\ScaleVector(\Pmt)(\Ptl)by\K to\Ctl
\ScaleVector(\Pmb)(\Pbr)by\K to\Cbr
\ScaleVector(\Pmb)(\Pbl)by\K to\Cbl
%
\IfBooleanTF{#1}{%
 \Pbox(\Crt)[l]{C_{rt}}\Pbox(\Crb)[l]{C_{rb}}
 \Pbox(\Clt)[r]{C_{lt}}\Pbox(\Clb)[r]{C_{lb}}
 \Pbox(\Ctr)[b]{C_{tr}}\Pbox(\Ctl)[b]{C_{tl}}
 \Pbox(\Cbr)[t]{C_{br}}\Pbox(\Cbl)[t]{C_{bl}}
%
\Pbox(\Pmr)[l]{P_{mr}}\Pbox(\Pmt)[b]{P_{mt}}%
```

```
\Pbox(\Pml)[r]{P_{ml}}\Pbox(\Pmb)[t]{P_{mb}}%
%
\polygon(\Pbr)(\Ptr)(\Ptl)(\Pbl)\thicklines}{}%
%
\moveto(\Pmr)
\curveto(\Crt)(\Ctr)(\Pmt)
\curveto(\Ctl)(\Clt)(\Pml)
\curveto(\Clb)(\Cbl)(\Pmb)
\curveto(\Cbr)(\Crb)(\Pmr)
\fillstroke
\egroup}

```

This user macro is used to call the `\Sellisse` macro with the desired parameters, but also to act with it on order to fill or stroke the ellipse contour, and to select some settings such as the contour line thickness, or the color of the ellipse contour or interior. the syntax is the following:

`\XSellisse`⟨*⋆1*⟩`(`⟨*center*⟩`)[`⟨*angle*⟩`]<`⟨*shear*⟩`>{`⟨*h-axis*⟩`}{`⟨*v axis*⟩`}`⟨*⋆2*⟩`[`⟨*settings1*⟩`][`⟨*settings2*⟩`]`

where there are two optional asterisks, ⟨*⋆1*⟩ and ⟨*⋆2*⟩; the first one controls the coloring of the ellipse: if present the interior is filled, if absent the contour is stroked; the second one controls the way a possibly sheared ellipse appears: if present, the construction is shown, if absent only the final result is shown; ⟨*center*⟩ is optional: if present, the ellipse center is specified; if absent, its center is at the origin of the picture axes; ⟨*angle*⟩ is optional with default value zero: if absent, the ellipse is not rotated and the ⟨*h-axis*⟩ remains horizontal, while the ⟨*v-axis*⟩ remains vertical, while if present and with a non vanishing value, the ellipse is rotated counterclockwise the amount specified, and, of course, if the value is negative, the rotation is clockwise. The optional parameter ⟨*shear*⟩, if present, shears the ellipse paralle the ⟨*h-axis*⟩ direction; the ⟨*settings1*⟩ and ⟨*settings2*⟩ operate as described for command `\Xellisse`.

```

\NewDocumentCommand\XSellisse{ s D(){0,0} O{0} D<>{0} m m s O{} o }%
    {\IfBooleanTF#1{\let\fillstroke\fillpath}%
                   {\let\fillstroke\strokepath}%
    \put(#2){\rotatebox{#3}{#8\relax
    \IfBooleanTF{#7}{\Sellisse*{#5}{#6}[#4]}%
                  {\Sellisse{#5}{#6}[#4]}%
    \IfValueTF{#9}{\let\fillstroke\strokepath
                   #9\Sellisse{#5}{#7}[#4]}{}}}%
    \ignorespaces}
```

## 1.8 Regular polygons and special ellipses

We finally arrive to more complex macros used to create special polygons and special ellipses.

### 1.8.1 Regular polygons

Regular polygons are not that special; it is possible to draw them by using the `\multiput` or `\xmultiput` commands, but a single command that does everything by itself with more built in functionalities is much handier. The new command `\RegPolygon` has the following syntax:

`\RegPoligon`⟨⋆⟩`(`⟨*center*⟩`){`⟨*radius*⟩`}{`⟨*number*⟩`}[`⟨*angle*⟩`]<`⟨*settings*⟩`>`

where ⟨⋆⟩ is an optional asterisk; its presence means that the polygon interior is filled, instead of the polygon contour being stroked; the ⟨*center*⟩ specification of the polygon is optional; if it is omitted, the polygon center goes to the origin of the `picture` coordinates; ⟨*radius*⟩ is the mandatory radius of the circumscribed circle, or, in other words, the distance of each polygon vertex form the ⟨*center*⟩; the mandatory ⟨*number*⟩ is an integer that specifies the number of polygon sides; the first vertex that is being drawn by this command, has an angle of zero degrees with respect to the ⟨*center*⟩; if a different initial ⟨*angle*⟩ different from zero is desired, it is specified through this optional argument; possibly the angle bracketed optional ⟨*setting*⟩ parameter may be used to specify, for example, the line thickness for the contour, and/or the color for the polygon contour or interior. See the documentation `euclideangeometry-man.pdf` for more information and usage examples.

```
\newcount\RPI
\NewDocumentCommand\RegPolygon{s D(){0,0}  m m O{0} D<>{\relax} }{{%
%\countdef\RPI=258
\RPI=0
\CopyVect#5:#3to\P
\CopyVect\fpeval{360/#4}:1to\R
\put(#2){#6\relax
         \moveto(\P)\fpdowhile{\RPI < #4}%
         {\MultVect\P by\R to\P
         \lineto(\P)\advance\RPI by 1}%
         \IfBooleanTF{#1}%
         {\fillpath}{#6\strokepath}}}\ignorespaces}
%%%%%%%%%
\def\DirOfVect#1to#2{\GetCoord(#1)\t@X\t@Y
\ModOfVect#1to\@tempa
\unless\ifdim\@tempa\p@=\z@
   \DividE\t@X by\@tempa to\t@X
   \DividE\t@Y by\@tempa to\t@Y
\fi\MakeVectorFrom\t@X\t@Y to#2\ignorespaces}%
```

### 1.8.2 The Steiner ellipse

The construction of the Steiner ellipse is very peculiar; it is almost intuitive that any triangle has infinitely many internal tangent ellipses; therefore it is necessary to state some other constraints to find one specific ellipse out from this unlimited set.

One such ellipse is the Steiner one, obtained by adding the constraint that the ellipse be tangent to the median points of the triangle sides. But one thing is the definition, and another totally different one is to find the parameters of such an ellipse; and working with ruler and compass, it is necessary to find a procedure to draw such an ellipse.

The construction described here and implemented with the `SteinerEllipse` macro is based on the following steps, each one requiring the use of some of the commands and/or transformations described in the previous sections.

1. Given a generic triangle (the coordinates of its three vertices) it is not necessary, but it is clearer to explain, if the triangle is shifted and rotated so as to have one of its sides horizontal, and the third vertex in the upper part of the `picture` drawing. So we first perform the initial shift and rotation and memorise the parameters of this transformation so that, at the end of the procedure, we can put back the triangle (and its Steiner ellipse) in its original position. Let us call this shifted and rotated triangle with the symbol $T_0$.
2. We transform $T_0$ with an affine shear transformation into an isosceles triangle $T_1$ that has the same base and the same height as $T_0$. We memorise the shear "angle" so as to proceed to an inverse transformation when the following steps are completed: let be $\alpha$ this shear angle; geometrically it represents the angle of the sheared vertical coordinate lines with respect to the original vertical position.
3. With another affine vertical scaling transformation we transform $T_1$ into an equilateral triangle $T_2$; the ratio of the vertical transformation equals the ratio between the $T_2$ to the $T_1$ heights; we memorise this ratio for the reverse transformation at the end of the procedure.
4. The Steiner ellipse of the equilateral triangle $T_2$ is its incircle. We are almost done; we just have to proceed to the inverse transformations; getting back from $T_2$ to $T_1$ first implies transforming the incircle of $T_2$ into an ellipse with its vertical axis scaled by the inverse ratio memorised in step 3.
5. The second inverse transformation by the shear angle is easy with the passage from $T_1$ to $T_0$, but it would be more difficult for transforming the ellipse into the sheared ellipse. We have already defined the `\Sellipse` and the `\XSellipse` macros that may take care of the ellipse shear transformation; we already memorised the shear angle in step 2, therefore the whole procedure, except for putting back the triangle, is almost done.
6. Eventually we perform the last shifting and rotating transformation and the whole construction is completed.

The new macro Steiner ellipse has therefore the following syntax:

| `\SteinerEllipse`⟨⋆⟩`(`⟨*P1*⟩`)(`⟨*P2*⟩`)(`⟨*P3*⟩`)[`⟨*diameter*⟩`]` |
|---|

where ⟨*P1*⟩, ⟨*P2*⟩, ⟨*P3*⟩ are the vertices of the triangle; ⟨⋆⟩ is an optional asterisk; without it the maro draws only the final result, that contains only the given triangle and its Steiner ellipse; on the opposite, if the asterisk is used the whole

construction from $T_0$ to its Steiner ellipse is drawn; the labelling of points is done with little dots of the default ⟨*diameter*⟩ or a specified value; by default it is a 1 pt diameter, but sometimes it would be better to use a slightly larger value (remembering that 1 mm — about three points — is already too much). Please refer to the documentation file `euclideangeometry-man.pdf` for usage examples and suggestions.

```
%

\NewDocumentCommand\SteinerEllipse{s d() d() d() O{1}}{\bgroup
%
\IfBooleanTF{#1}{}{\put(#2)}{%
  \CopyVect0,0to\Pu
  \SubVect#2from#3to\Pd
  \SubVect#2from#4to\Pt
  \ModAndAngleOfVect\Pd to\M and\Rot
  \MultVect\Pd by-\Rot:1 to\Pd  \MultVect\Pt by-\Rot:1 to\Pt
  \IfBooleanTF{#1}{}{\rotatebox{\Rot}}{\makebox(0,0)[bl]{%
  \Pbox(\Pu)[r]{P_1}[#5]<-\Rot>\Pbox(\Pd)[t]{P_2}[#5]<-\Rot>
  \Pbox(\Pt)[b]{P_3}[#5]<-\Rot>%
  \polygon(\Pu)(\Pd)(\Pt)%
  \edef\B{\fpeval{\M/2}}\edef\H{\fpeval{\B*tand(60)}}
  \IfBooleanTF{#1}{\Pbox(\B,\H)[b]{H}[#5]
  \polygon(\Pu)(\B,\H)(\Pd)}{}%
  \edef\R{\fpeval{\B*tand(30)}}
  \IfBooleanTF{#1}{\Pbox(\B,\R)[bl]{C}[#5]
    \Circlewithcenter\B,\R radius{\R}}{}%
  \GetCoord(\Pt)\Xt\Yt\edef\VScale{\fpeval{\Yt/\H}}
  \IfBooleanTF{#1}{\polyline(\Pu)(\B,\Yt)(\Pd)
  \Pbox(\B,\Yt)[b]{V}[#5]}{}%
  \edef\Ce{\fpeval{\R*\VScale}}
  \IfBooleanTF{#1}{\Xellisse(\B,\Ce){\R}{\Ce}
    \Pbox(\B,\Ce)[r]{C_e}[#5]\Pbox(\B,0)[t]{B}[#5]}{}%
  \SubVect\B,0 from\Pt to\SlMedian
  \IfBooleanTF{#1}{\Dotline(\B,0)(\Pt){2}[1.5]}{}%
  \ModAndAngleOfVect\SlMedian to\Med and\Alfa
  \edef\Alfa{\fpeval{90-\Alfa}}
    \IfBooleanTF{#1}{\Dotline(\B,\Yt)(\B,0){2}[1.5]
    \Pbox(\fpeval{\B+\Ce*tand{\Alfa}},\Ce)[l]{C_i}[#5]
    \VectorArc(\B,0)(\B,15){-\Alfa}
    \Pbox(\fpeval{\B+2.5},14)[t]{\alpha}[0]}{}%
  \edef\a{\R}\edef\b{\Ce}%
\CopyVect\fpeval{\B+\Ce*tand{\Alfa}},\Ce to\CI
\XSellisse(\CI)<\Alfa>{\R}{\Ce}
}}}%
\egroup\ignorespaces}
\let\EllisseSteiner\SteinerEllipse
```

### 1.8.3 The ellipse that is internally tangent to a triangle while one of its foci is prescribed

We now are going to tackle another problem. As we said before, any triangle has an infinite set of internally tangent circles, unless some further constraint is specified.

Another problem of this kind is the determination and geometrical construction of an internally tangent ellipse when one focus is specified; of course since the whole ellipse is totally internal to the triangle, we assume that the user has already verified that the coordinates of the focus fall inside the triangle. We are not going to check this feature in place of the user; after all, if the user draws the triangle within a `picture` image, together with the chosen focus, is suffices a glance to verify that such focus lays within the triangle perimeter.

The geometrical construction is quite complicated, but it is described in a paper by Estevão V. Candia on *TUGboat* 2019 **40**(3); it consists of the following steps.

1. Suppose you have specified a triangle by means of its three vertices, and a point inside it to play the role of a focus; it is necessary to find the other focus and the main axis length in order to have a full description of the ellipse.
2. To do so, it is necessary to find the focus three symmetrical points with respect to the three sides.
3. The center of the three point circle through these symmetrical points is the second focus.
4. The lines that join the second focus to the three symmetrical points of the first focus, intersect the triangle sides in three points that result to be the tangency points of the ellipse to the triangle.
5. Chosen one of these tangency points and computing the sum of its distances from both foci, the total length of the ellipsis main axis is found.
6. Knowing both foci, the total inter focal distance is found, therefore equation (1) allows to find the other axis length.
7. The inclination of the focal segment gives us the the rotation to which the ellipse is subject, and the middle point of such segment gives the ellipse center.
8. At this point we have all the necessary elements to draw the ellipse.

We need another little macro to find the symmetrical points; if the focus $F$ and its symmetrical point $P$ with respect to a side/segment, the intersection of such segment $F-P$ with the side is the segment middle point $M$; from this property we derive the formula $P = 2M - F$. Now $M$ is also the intersection of the line passing through $F$ and perpendicular to the side. Therefore it is particularly simple to compute, but its better to have available a macro that does the whole work; here it is, but it assumes the the center of symmetry is already known:

```
355
356 \def\SymmetricalPointOf#1respect#2to#3{\ScaleVect#2by2to\Segm
357 \SubVect#1from\Segm to#3\ignorespaces}
```

And its syntax is the following:

```
\SymmetricalPointOf⟨focus⟩ respect⟨symmetry center⟩
    to⟨symmetrical point⟩
```

where the argument names are self explanatory.

The overall macro that executes all the passages described in the above enumeration follows; the reader can easily recognise the various steps, since the names of the macros are self explanatory; the $G_i$ point names are the symmetrical ones to the first focus $F$; the $M_i$ points are the centers of symmetry; the $F'$ point is the second focus; the $T_i$ points are the tangency points. The macro `\EllipseWithFOcus` has the following syntax:

```
\EllipseWithFocus⟨⋆⟩(⟨P1⟩)(⟨P2⟩)(⟨P3⟩)(⟨focus⟩)
```

where ⟨*P1*⟩, ⟨*P2*⟩, ⟨*P3*⟩ are the triangle vertices and ⟨*focus*⟩ contains the first focus coordinates; the optional asterisk, as usual, selects the construction steps versus the final result: no asterisk, no construction steps.

```
\NewDocumentCommand\EllipseWithFocus{s d() d() d() d()}{\bgroup%
\CopyVect#2to\Pu
\CopyVect#3to\Pd
\CopyVect#4to\Pt
\CopyVect#5to\F
\polygon(\Pu)(\Pd)(\Pt)
\Pbox(\Pu)[r]{P_1}[1.5pt]\Pbox(\Pd)[t]{P_2}[1.5pt]
\Pbox(\Pt)[b]{P_3}[1.5pt]\Pbox(\F)[b]{F}[1.5pt]
\SegmentArg(\Pu)(\Pt)to\At
\SegmentArg(\Pu)(\Pd)to\Ad
\SegmentArg(\Pd)(\Pt)to\Au
\IntersectionOfLines(\Pu)(\At:1)and(\F)(\fpeval{\At+90}:1)to\Mt
\IntersectionOfLines(\Pd)(\Ad:1)and(\F)(\fpeval{\Ad+90}:1)to\Md
\IntersectionOfLines(\Pd)(\Au:1)and(\F)(\fpeval{\Au+90}:1)to\Mu
\IfBooleanTF{#1}{\Pbox(\Mt)[br]{M_3}[1.5pt]\Pbox(\Md)[t]{M_2}[1.5pt]
  \Pbox(\Mu)[b]{M_1}[1.5pt]}{}
\SymmetricalPointOf\F respect\Mu to\Gu
\IfBooleanTF{#1}{\Pbox(\Gu)[l]{G_1}[1.5pt]}{}
\SymmetricalPointOf\F respect \Md to\Gd
\IfBooleanTF{#1}{\Pbox(\Gd)[t]{G_2}[1.5pt]}{}
\SymmetricalPointOf\F respect \Mt to\Gt
\IfBooleanTF{#1}{\Pbox(\Gt)[r]{G_3}[1.5pt]}{}
\IfBooleanTF{#1}{\ThreePointCircle*(\Gu)(\Gd)(\Gt)}%
                {\ThreePointCircle(\Gu)(\Gd)(\Gt)}
\CopyVect\C to\Fp \Pbox(\Fp)[l]{F'}[1.5pt]
\IfBooleanTF{#1}{%
\Dotline(\F)(\Gt){2}[1.5pt]
\Dotline(\F)(\Gd){2}[1.5pt]
\Dotline(\F)(\Gu){2}[1.5pt]}{}
\IntersectionOfSegments(\Pu)(\Pt)and(\Fp)(\Gt)to\Tt
```

```
\IntersectionOfSegments(\Pu)(\Pd)and(\Fp)(\Gd)to\Td
\IntersectionOfSegments(\Pd)(\Pt)and(\Fp)(\Gu)to\Tu
\IfBooleanTF{#1}{\Pbox(\Tu)[l]{T_1}[1.5pt]
\Pbox(\Td)[b]{T_2}[1.5pt]
\Pbox(\Tt)[tl]{T_3}[1.5pt]
\Dashline(\Fp)(\Gu){1}\Dashline(\Fp)(\Gd){1}\Dashline(\Fp)(\Gt){1}}{}
\DistanceAndDirOfVect\Fp minus\Tt to\DFp and\AFu
\DistanceAndDirOfVect\F  minus\Tt to\DF and\AF
\SegmentCenter(\F)(\Fp)to\CE \Pbox(\CE)[b]{C}[1.5pt]
\edef\a{\fpeval{(\DFp+\DF)/2}}
\SegmentArg(\F)(\Fp)to\AngFocalAxis
\SegmentLength(\F)(\CE)to\c
\AxisFromAxisAndFocus\a and\c to\b
\Xellisse(\CE)[\AngFocalAxis]{\a}{\b}[\thicklines]
\VECTOR(-30,0)(120,0)\Pbox(120,0)[t]{x}[0]
\VECTOR(0,-20)(0,130)\Pbox(0,130)[r]{y}[0]\Pbox(0,0)[tr]{O}[1.5pt]
\egroup\ignorespaces}
\let\EllisseConFuoco\EllipseWithFocus
```

# 2 Comments on this package

In general we found very comfortable to draw ellipses and to define macros to draw not only such shapes or filled elliptical areas, but also to create "legends" with coloured backgrounds and borders; such applications found their way in other works. But here we dealt with other geometrical problems. The accompanying document `euclideangeometry-man.pdf` describes much clearly with examples what you can do with the macros described in this package. In facts, this file just describes the package macros, and it gives some ideas on how to extend the ability of `curve2e` to draw geometrical diagrams. The users who would like to modify or to add some functionalities are invited to do so; I will certainly acknowledge their contributions and even add their names to the list of authors.

As long as I can, I enjoy playing with LaTeX and its wonderful facilities; but, taking into consideration my age, I would invite the users to consider the possibility of assuming the maintenance of this package.


# Aknowledgements

I am very grateful to Enrico Gregorio who let me know the several glitches I made in my first version; besides being a real TeX wizard, he is a wise person and suggested me several things that was important to change, because they could offer risks of confusion with other packages.