

Package `curve2e` user manual

Claudio Beccari

`claudio dot beccari at gmail dot com`

Version 2.3.1 – Last revised 2023-07-04

Contents		3 Summary and examples of new commands	6
1 Installation	1	4 Remarks	27
2 Package <code>pict2e</code> and this extension <code>curve2e</code>	2	5 Acknowledgements	28

Abstract

This file contains the user manual of the `curve2e` extension package to the `pict2e` bundle; the latter was described by Lamport himself in the 1994 second edition of his \LaTeX handbook.

Please take notice that on April 2011 a package `pict2e` upgraded version has been released that incorporates some of the commands defined in early versions of this package `curve2e`; apparently there are no conflicts, because this package contains only the advanced features that extend the above package.

Since this extension redefines some commands and introduces some more drawing facilities (that allow to draw circular arcs and arbitrary curves with the minimum of user intervention) the user needs a user manual that contains several actual examples; this auxiliary manual is contained in file `curve2e-manual.pdf`. The software available to show the drawing code and its result after compilation is incompatible with the usual \LaTeX `ltxdoc` class for code documentation, therefore a separate user manual has been made available. If the user wants to explore the `curve2e` code and its documentation s/he has available the `curve2e.pdf` file. Either file is readable by entering in a terminal window the command `texdoc curve2e-manual` or `texdoc curve2e.pdf`; please do not forget the extension when you want to read the code documentation.

1 Installation

If \TeX system was installed with a \TeX Live or a $\text{MiK}\text{\TeX}$ complete and updated distribution this package is already installed; in order to verify open a terminal or command prompt window and use it to enter the `texdoc curve2e-manual`; as soon as the command is executed a screen window should open displaying this manual. If this does not happen, either you misspelt the command (it happens more often than not), or your installation is not complete and updated.

I suggest you to use your computer installation facilities to install this bundle. Otherwise, download this `curve2e.zip` package from one of the CTAN (Comprehensive TeX Archive Network) archives, to your downloads folder. Before doing anything else verify if you have a *personal texmf* tree; if not, create one reading you distribution instruction; let us assume that your personal `texmf` archive is in `HOME/texmf` (on Windows change the slash with a backslash) and `HOME` is a path starting from the root of you hard disk and going through several other folders. On Windows 10 it might be `C:\Users\YourName`; on Linux it might simply be `~`; on Mac it would be `~/Library`, but sometimes the `HOME` might be different, especially on Windows platforms. Create the following subfolders:

1. `HOME/texmf/source/latex/curve2e/`
2. `HOME/texmf/doc/latex/curve2e/`
3. `HOME/texmf/tex/latex/curve2e/`

Now move file `curve2e.zip` to the `.../source/latex/curve2e/` folder; then decompress the `.zip` file with the software you have available on your platform. Run `pdflatex` on the `.dtx` file; then do the same with the `curve2e-manual.tex`. You might need to repeat these compilations two or three times in order to have the table of contents and all the references correctly connected.

This done move the `.pdf` files to the `.../doc/latex/curve2e/` folder. and move the `.sty` files to the `.../tex/latex/curve2e/` folder.

Clear the `.../source/latex/curve2e/` folder from the auxiliary files, all those remaining in the folder except those that have the extensions `.zip`, `.dtx`, `.tex`, and `.txt`. Read the `README.txt` file.

If your `TEX` system is correctly set up, your files in your personal tree should be immediately usable; probably you have to create the file-name database with `MiKTeX`; in this case read the documentation of your `MiKTeX` installation to discover how to do it.

Remember to delete all these subfolders if you decide to install a complete updated version of your favourite distribution, and you'd better keep it updated approximately once every 7 or 10 days. This is much simpler than to struggle with these manual operations.

2 Package `pict2e` and this extension `curve2e`

Package `pict2e` was announced in issue 15 of `latexnews` around December 2003; it was declared that the new package would replace the dummy one that had been accompanying every release of `LATEX 2ε` since its beginnings in 1994. The dummy package was just issuing an info message that simply announced the temporary unavailability of the real package.

Eventually Gäßlein and Niepraschk implemented what Lamport himself had already documented in the second edition of his `LATEX` handbook, it was a `LATEX` package that contained the macros capable of removing all the limitations contained in the standard commands of the original `picture` environment; specifically what follows.

1. The line and vector slopes were limited to the ratios of relative prime one-digit integers of magnitude not exceeding 6 for lines and 4 for vectors.
2. Filled and unfilled full circles were limited by the necessarily limited number of specific glyphs contained in the special `LATEX picture` fonts.

3. Quarter circles radii were also limited for the same reason.
4. Ovals (rectangles with rounded corners) could not be too small because of the unavailability of small radius quarter circles, nor could be too large, in the sense that after a certain radius the rounded corners remained the same curvature and would not increase proportionally to the oval size.
5. Vector tips had only one possible shape and matched the limited number of vector slopes.
6. For circles and inclined lines and vectors just two possible thicknesses were available.

Package `pict2e` removes most if not all the above limitations.

1. Line and vector slopes are virtually unlimited; the only remaining limitation is that the direction coefficients must be three-digit integer numbers; they need not be relatively prime; with the 2009 upgrade even this limitation was removed and now slope coefficients can be any fractional number whose magnitude does not exceed 16 384, the maximum dimension in points that `TeX` can handle.
2. Filled and unfilled circles can be of any size.
3. Ovals can be designed with any specified corner curvature and there is virtually no limitation to such curvatures; of course corner radii should not exceed half the lower value between the base and the height of the oval.
4. There are two shapes for the arrow tips; the triangular one traditional with `LaTeX` vectors, or the arrow tip with PostScript style.
5. The `\linethickness` command changes the thickness of all lines, straight, curved, vertical, horizontal, arrow tipped, et cetera.

This specific extension package `curve2e` adds the following features.

1. Point coordinates can be specified with macros; this is similar to “naming” points; it eases editing the user’s graphic work, because points that are used several times are specified with a single macro; it also eases the transmission of coordinates between different macros and environments. It is also important for the following feature, described in the following entry..
2. Point coordinates may be specified in both cartesian and polar form: internally they are handled as cartesian coordinates, but the user can specify his/her points also in polar form. In order to avoid confusion with other graphic packages, `curve2e` uses the usual comma separated couple $\langle x, y \rangle$ of integer or fractional numbers for cartesian coordinates, and the couple $\langle \theta \rangle : \langle \rho \rangle$ for polar coordinates (the angle preceding the radius).

All graphic object commands accept polar or cartesian coordinates at the choice of the user who may use for each object the formalism s/he prefers. Also the `\put` and `\multiput` commands have been redefined so as to accept cartesian or polar coordinates.

Of course the user should pay attention to the meaning of cartesian vs. polar coordinates. Both imply a displacement with respect to the actual origin of the axes. So when a circle center is placed at coordinates a, b with a normal `\put` command, the circle is placed exactly in that point; with a normal `\put` command the same happens if coordinates $\alpha : \rho$ are specified.

But if the `\put` command is nested into another `\put` command, the current origin of the axes is displaced — this is obvious and the purpose of nesting `\put` commands is exactly that. But if a segment is specified so that its ending point is at a specific distance and in specific direction from its starting point, polar coordinates appear to be the most convenient to use; in this case, though, the origin of the axes becomes the starting point of the segment, therefore the segment might be drawn in a strange way. Attention has been paid to avoid such misinterpretation, but maybe some unusual situation may not have come to my mind; feedback is very welcome. Meanwhile pay attention when you use polar coordinates.

3. At user level most if not all coordinate pairs and slope pairs are treated as *ordered pairs*, that is *complex numbers*; in practice the user does not notice any difference from what s/he was used to, but all the mathematical treatment to be applied to these entities is coded as complex number operations, since complex numbers may be viewed non only as ordered pairs, but also as vectors or as roto-amplification operators.
4. Commands for setting the line terminations were introduced; the user can chose between square or rounded caps; the default is set to rounded caps; now this feature is directly available with `pict2e`.
5. Commands for specifying the way two straight or curves lines join to one another.
6. Originally the `\line` macro was redefined so as to allow large (up to three digits) integer direction coefficients, but maintaining the same syntax as in the original `picture` environment; now `pict2e` removes the integer number limitations and allows fractional values, initially implemented by `curve2e`; now direction coefficients may be specified in polar form.
7. A new macro `\Line` was originally defined by `curve2e` so as to avoid the need to specify the horizontal projection of inclined lines; now this functionality is available directly with `pict2e`; but this `curve2e` macro name now conflicts with the `pict2e` 2009 version; therefore its name is changed to `\LINE` and supposedly it will not be used very often, if ever, by the end user (but it is used within this package macros).
8. A new macro `\LINE` was defined in order to join two points specified with their coordinates; this is now the normal behaviour of the `\Line` macro of `pict2e`, so that in this package `\LINE` is now renamed `\segment`; there is no need to use the `\put` command with this segment specification.
9. A new macro `\DashLine` (alias: `\Dline`) is defined in order to draw dashed lines joining any two given points; the dash length and gap (equal to one another) get specified through one of the macro arguments. The starting point may be specified in cartesiano or polar form; the end point in cartesian format specifies the desired end point; but, if the second point is in polar form, it is meant *relative to the starting point*, not as an absolute end point. See the examples further on.
10. A similar new macro `\Dotline` is defined in order to draw dotted straight lines as a sequence of equally spaced dots, where the gap can be specified by the user; such straight line may have any inclination, as well as the above dashed lines. Polar coordinates for the second point have the same relative meaning as specified for the `\Dashline` macro. The dot diameter may be specified with an optional argument; by default this diameter equals the `1pt` width.

11. Similar macros are redefined for vectors; `\vector` redefines the original macro but with the vector slope limitations removed and the vector direction may be given in polar form; `\Vector` gets specified with its two horizontal and vertical components in analogy with `\Line`; `\VECTOR` joins two specified points (without using the `\put` command) with the arrow pointing to the second point. `\VVECTOR` may be available if used with a sufficiently recent L^AT_EX kernel version; it draws a vector between two given points, with arrow tips at both ends.
12. A new macro `\polyline` for drawing polygonal lines is defined that accepts from two vertices up to an arbitrary (reasonably limited) number of them (available now also in `pict2e`); here it is redefined so as to allow an optional specification of the way segments for the polyline are joined to one another. Vertices may be specified with polar coordinates and are always relative to the preceding point.
13. The `pict2e` `\polygon` macro draws closed polylines (in practice general polygons) has been redefined in such a way that it can accept the various vertices specified with (relative) polar coordinates. The `\polygon*` macro produces a color filled polygon; the default color is black, but a different color may be specified with the usual `\color` command given within the same group where `\polygon*` is enclosed.
14. A new macro `\Arc` is defined in order to draw an arc with arbitrary radius and arbitrary aperture (angle amplitude); this amplitude is specified in sexagesimal degrees, not in radians; a similar functionality is now achieved with the `\arc` macro of `pict2e`, which provides also the starred version `\arc*` that fills up with the current color the sector generated by a circular arc. It must be noticed that the syntax is slightly different, so that it's reasonable that these commands, in spite of producing identical arcs, might be more comfortable with this or that syntax.
15. Two new macros `\VectorArc` and `\VectorARC` (alias `\VVectorArc`) are defined in order to draw circular arcs with an arrow at one or both ends.
16. A new macro `\Curve` is defined so as to draw arbitrary curved lines by means of cubic Bézier splines; the `\Curve` macro requires only the curve nodes and the directions of the tangents at each node. The starred version fills up the interior of the curve with the current color.
17. The above `\Curve` macro is a recursive macro that can draw an unlimited (reasonably limited) number of connected Bézier spline arcs with specification of the tangent direction at the interpolation nodes. It is possible to use a lower level macro `\CbezierTo` that does the same but lets the user specify the control points of each arc; it is more difficult to use but it is more performant. `\Curve` recognises also an optional argument to set a direction change, that is a cusp.
18. The basic macros used within the cumulative `\Curve` macro can be used individually in order to draw any curve, one cubic arc at the time; but they are intended for internal use, even if it is not prohibited to use them; by themselves such arcs are not different from those used by `\Curve`, but the final command, `\FillCurve`, should be used in place of `\CurveFinish`, so as to fill up the closed path with the locally specified color; see figure 11. It is much more convenient to use the starred version of the `\Curve` macro.

The `pict2e` package already defines macros such as `\moveto`, `\lineto`, `\curveto`,

`\closepath`, `\fillpath`, and `\strokepath`; `curve2e` just redefines them so as to accept also polar coordinates; of course these macros can be used by the end user, and sometimes they perform better than the macros defined in this package, because the user has a better control on the position of the Bézier splines control points; in this case the control points are sort of rigid. It would be very useful to resort to the `hobby` package, but its macros are compatible with those of the `tikz` and `pgf` packages, not with `curve2e`; an interface should be created in order to deal with the `hobby` package, but this has not been done yet.

In order to make the necessary calculations many macros have been defined so as to use complex number arithmetics to manipulate point coordinates, directions (unit vectors, also known as ‘versors’), rotations and the like. In the first versions of this package the trigonometric functions were also defined in a way that the author believed to be more efficient than those defined by the `trig` package; in any case the macro names were sufficiently different to accommodate both definition sets in the same \LaTeX run. With the progress of the \LaTeX 3 language, package `xfp` functionalities have become available, and any sort of calculations can be done floating point decimal numbers; therefore the most common algebraic, irrational and transcendental functions can be computed in the background with the stable internal floating point facilities. We maintain some computation with complex number algebra, but use the `xfp` functionalities for other computations.

Many aspects of this extension could be fine tuned for better performance; many new commands could be defined in order to further extend this extension. If the new service macros are accepted by other \TeX and \LaTeX programmers, this version could become the start for a real extension of the `pict2e` package or even become a part of it. Actually some macros have already been included in the `pict2e` package. The `\Curve` algorithm, as said before, might be redefined so as to use the macros introduced by the `hobby` package, that implements for the `tikz` and `pgf` packages the same functionalities that John Hobby implemented for the `METAFont` and `METAPOST` programs.

For these reasons I suppose that every enhancement should be submitted to Gäßlein, Niepraschk, and Tkadlec who are the prime maintainers of `pict2e`; they are the only ones who can decide whether or not to incorporate new macros in their package.

3 Summary and examples of new commands

This package `curve2e` extends the power of `pict2e` with the following modifications and the following new commands.

1. This package `curve2e` calls directly the \LaTeX packages `color` and `pict2e`; it passes to the latter one any possible option that it can receive; actually the only options that make sense for `pict2e` are those concerning the arrow tip shapes, either \LaTeX (default) or PostScript styled, because it is assumed that if this package is used, the original \LaTeX commands are of no interest: see the `pict2e` documentation in order to find the correct options `pict2e` can receive. If the user wants to use the `xcolor` package, it has to load it *before* `curve2e`.
2. Since they are used very much in the following examples, we recall some commands to label certain drawing elements, such as points, lines, arcs,

and similar lines, and to insert legends in some figures. They are `\legenda`, `\Zbox` and `\Pbox`; their syntax is the following:

```
\legenda(<coordinates>){<formula>}
\Zbox(<coordinates>)[<position>]{<formula>}[<dot diameter>]
\Pbox(<coordinates>)[<position>]{<text>}[<dot diameter>][<*><angle>]
```

They are described in any up to date documentation of `curve2e`.

3. The user is offered new commands in order to control the line terminators and the line joins; specifically:
 - `\roundcap`: the line is terminated with a semicircle;
 - `\squarecap`: the line is terminated with a half square;
 - `\roundjoin`: two lines are joined with a rounded join;
 - `\beveljoin`: two lines are joined with a bevel join;
 - `\miterjoin`: two lines are joined with a miter join.

All the above commands should respect the intended range; but since they act at the PostScript or PDF level, not at \TeX level, it might be necessary to issue the necessary commands in order to restore the previous terminator or join; in other words, groups and environments do not have any influence on these commands.

4. The commands

```
\linethickness{<thickness>}
\thicklines
\thinlines
\defaultlinethickness{<thickness>}
```

always redefine the internal `\@wholewidth` and `\@halfwidth` so that the latter ones always refer to a full width and to a half of it in this way if you issue the command `\defaultlinethickness{2pt}` all thin lines will be drawn with a thickness of 1 pt while, if a drawing command directly refers to the internal value `\@wholewidth`, its line will be drawn with a thickness of 2 pt. If one issues the declaration `\thinlines` all lines will be drawn with a 1 pt width, but if a command refers to the internal value `\@halfwidth` the line will be drawn with a thickness of 0.5 pt. The command `\linethickness` redefines the above internals but does not change the default width value; all these width specifications apply to all lines, straight ones, curved ones, circles, ovals, vectors, dashed lines, and so on. It's better to recall that `\thinlines` and `\thicklines` are declarations that do not take arguments; on the pposite the other commands, as shown in the above syntax medallion, accept a specific thickness value, that is either a length specification complete of its units, or a dimensional expression.

5. Straight lines and vectors are redefined in such a way that fractional slope coefficients may be specified; the zero length line and vector does not produce errors and is ignored; the zero length vectors draw only the arrow tips.
6. New line and vector macros are defined that avoid the necessity of specifying the horizontal component; `\put(3,4){\Line(25,15)}` specifies a

segment that starts at point (3, 4) and goes to point (3 + 25, 4 + 15); the command `\segment(3,4)(28,19)` achieves the same result without the need of using the `\put` command. Therefore `\Line` is just for internal usage, rather than a user command. Now `curve2e` has available also the “arc vectors” with the arrow tips at one or at both ends. The `\segment` syntax is

```
\segment(<starting point>)(<ending point>)
```

The same applies to the vector commands `\Vector` and `\VECTOR` and `\VVECTOR`; the latter command behaves as `\VECTOR` but draws a vector with arrow tips at both ends; furthermore this command is available only with main versions 2 or higher of `curve2e`.

```
\put(<starting point>){\Vector<vector cartesian or polar components>}
\VECTOR(<starting point>)(<ending point>)
\VVECTOR(<starting point>)(<ending point>)
```

Experience has shown that the commands intended to join two specified points are particularly useful; see figure 1.

```
\unitlength=0.01\linewidth
\begin{picture}(80,20)
\AutoGrid
\put(0,0){\vector(1.5,2.3){10}}
\put(20,0){\VECTOR(10,15.33333)}
\VECTOR(40,0)(50,15.33333)
\ifdefined\VVECTOR \VVECTOR(60,0)(80,10)\fi
\end{picture}
```

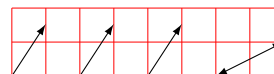


Figure 1: Three (displaced) identical vectors obtained with the three vector macros; a double tipped vector is also shown.

- The `\polyline` command has already been introduced in `pict2e`: in `curve2e` it is redefined so as to accept also polar coordinates; this new version of `\polyline` accepts also an optional argument to specify how two consecutive segments join together; it accepts an unlimited list of point coordinates, possibly stored in macros, enclosed within round parentheses; the command draws a sequence of connected segments that join in order the specified points; the syntax is:

```
\polyline[<optional join style>](<P1>)(<P2>)...(<Pn>)
```

See figure 2 where a regular pentagon is drawn; usage of polar coordinates is also shown; please notice how polar coordinates act in this figure.

Examples of using polar and cartesian coordinates are shown in figure 3. Notice the `\AutoGrid` macro that draws the grid of mesh lines that are very useful to set objects at the right positions. The lines are `10\unitlength` apart.

A similar example may be obtained with the `\polygon` macro that does not require to terminate the polyline at the starting point. Figure 4 shows how to get a coloured filled pentagon.


```

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

```

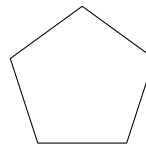


Figure 2: Polygonal line obtained by means of the `\polyline` command; vertex coordinates are in polar form.

```

\unitlength =0.02\textwidth
\begin{picture}(40,30)
\AutoGrid
\Zbox(40,0)[1]{40,0}[1]
\Zbox(90:30)[bc]{90{:}30}[1]
\Zbox(45:30)[bc]{45{:}30}[1]
\Zbox(30,30)[bc]{30,30}[1]
\multitup(0,0)(20:10){5}%
{\makebox(0,0){\rule{1.5mm}{1.5mm}}}
\end{picture}

```

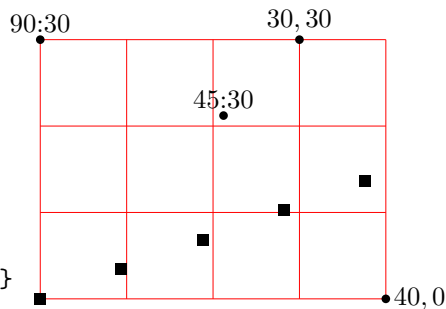


Figure 3: Use of cartesian and absolute polar coordinates. The `\Zbox` macro is just a shortcut to set a small dot with a (math) legend close to it.

```

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

```



Figure 4: A pentagon obtained by means of the `\polygon*` command; vertex coordinates are in relative polar form.

- The new command `\Dashline` (alias: `\Dline` for backwards compatibility):

`\Dashline(\langle first point \rangle)(\langle second point \rangle){\langle dash and gap length \rangle}`

draws a dashed line containing as many dashes as possible, just as long as specified, and separated by a gap exactly the same size; actually, in order to make an even gap-dash sequence, the desired dash length is used to do some computations in order to find a suitable length, close to the one specified, such that the distance of the end points is evenly divided in equally sized dashes and gaps. The end points may be anywhere in the drawing area, without any constraint on the slope of the joining segment. The desired dash length is specified as a fractional multiple of `\unitlength`; see figure 5.

Another example of usage of cartesian and polar coordinates usage is shown in figure 3 together with its code.

- Analogous to `\Dashline`, a new command `\Dotline` draws a dotted line with the syntax:

```

\Dotline(\langle first point \rangle)(\langle end point \rangle){\langle dot gap \rangle}

```

```

\unitlength =1mm
\begin{picture}(40,40)
\AutoGrid(40,40)
\Dashline(0,0)(40,10){4}
\put(0,0){\circle*{2}}
\Dashline(40,10)(0,25){4}
\put(40,10){\circle*{2}}
\Dashline(0,25)(20,40){4}
\put(0,25){\circle*{2}}
\put(20,40){\circle*{2}}
\Dotline(0,0)(40,40){2}[0.75mm]
\put(40,40){\circle*{2}}
\end{picture}

```

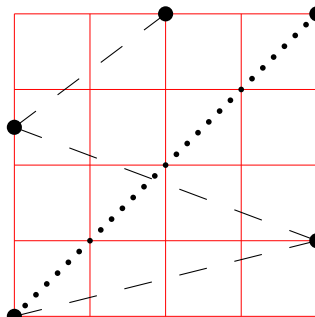


Figure 5: Dashed lines and graph grid

See figures 5 and 7 for examples.

10. `\GraphicGrid` and `\AutoGrid` are commands that draw a red grid under the drawing with lines separated `10\unitlengths` apart; it is described only with a comma separated couple of numbers, representing the base and the height of the grid, see figure 5; it's better to specify multiples of ten and the grid can be placed anywhere in the drawing canvas by means of `\put`, whose cartesian coordinates are multiples of 10; nevertheless the grid line distance is rounded to the nearest multiple of 10, while the point coordinates specified to `\put` are not rounded at all; therefore some care should be used to place the working grid on the drawing canvas.

```

\GraphicGrid(grad base, grid height)
\AutoGrid

```

This grid is intended as an aid while drawing; even if you sketch your drawing on millimetre paper, the drawing grid turns out to be very useful; one must only delete or comment out the command when the drawing is finished. Several examples of usage of such grid are shown in several figures.

`\Autogrid` does not require arguments, but requires the canvas dimensions and offsets to be specified as multiples of 10; if the latter are specified they are simply ignored.

11. New trigonometric function macros have been computed by means of the functionalities of the `xfp` included in the \LaTeX kernel. The difference with the other existing macros is that angles are specified in sexagesimal degrees, so that the users need not transform to radians. The computations are done taking into account that “abnormal” values that can occasionally be avoided, for example $\tan 90^\circ$ must be avoided and replaced with a suitably large number, because the \TeX system does not handle “infinity”. These trigonometric functions are used within the complex number macros; but if the user wants to use them the syntax is the following:

```

\SinOf<angle>to<control sequence>
\CosOf<angle>to<control sequence>
\tanOf<angle>to<control sequence>

```

```

\unitlength =0.025\textwidth
\begin{picture}(40,30)
\AutoGrid(40,30)
\Dashline(0,0)(40,10){2}
\Dashline(0,0)(40,20){2}
\Dashline(0,0)(40,30){2}
\Dashline(0,0)(30,30){2}
\Dashline(0,0)(20,30){2}
\Dashline(0,0)(10,30){2}
{\color{blue}%
\Dashline(40,0)(108:30){2}
\Dashline(40,0)(126:30){2}
\Dashline(40,0)(144:30){2}
\Dashline(40,0)(162:30){2}}
\end{picture}

```

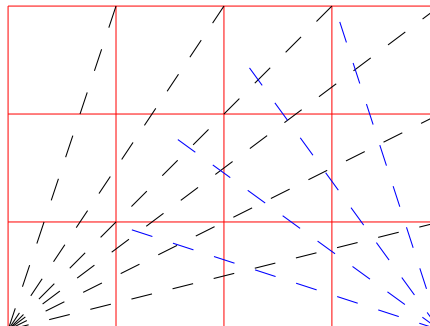


Figure 6: Different length dashed lines with the same nominal dash length; notice the relative polar coordinates used for the dashed lines starting at the grid lower right vertex.

The *control sequence* may then be used, for example, as a multiplying factor of a length.

12. Arcs can be drawn as simple circular arcs, or with one or two arrows at their ends (curved vectors); the syntax is:

```

\Arc(\center)(\starting point){\angle}
\VectorArc(\center)(\starting point){\angle}
\VectorARC(\center)(\starting point){\angle}
\VVectorArc(\center)(\starting point){\angle}

```

If the angle is specified numerically it must be enclosed in braces, while if it is specified with a control sequence the braces (curly brackets) are not necessary. The above macro `\Arc` draws a simple circular arc without arrows; `\VectorArc` draws an arc with an arrow tip at the ending point; `\VectorARC` (alias `\VVectorArc`) draws an arc with arrow tips at both ends; see figure 8.

Notice that the starting point may be specified with polar coordinates; differently to cartesian coordinates, that are absolute with respect with the drawing axes, the polar ones are relative to the center of the arcs with or without vector tips.

13. The available commands allow to create the drawings necessary to prove some geometrical theorems; for example let us prove the Pitagora's theorem. Figure 9 displays a right triangle with its hypotenuse laying horizontally; its vertices are labeled A, B, and C, being A the right angle vertex. The height relative to the hypotenuse intersects this side in point H, and divides the whole triangle ABC, in two similar smaller ones AHC, and ABH. Segments CH, and HB, add to the whole hypotenuse length. Figure 9 displays also the square CBDE, built on the hypotenuse and divided in two rectangles CHFE, and HBDF, where HF is the continuation of the height line.

```

\unitlength =0.025\textwidth
\begin{picture}(40,30)
\AutoGrid(40,30)
\Dotline(0,0)(40,10){1.5}[2pt]
\Dotline(0,0)(40,20){1.5}[2pt]
\Dotline(0,0)(40,30){1.5}[2pt]
\Dotline(0,0)(30,30){1.5}[2pt]
\Dotline(0,0)(20,30){1.5}[2pt]
\Dotline(0,0)(10,30){1.5}[2pt]
{\color{red}\relax
\Dotline(40,0)(108:30){1.5}
\Dotline(40,0)(126:30){1.5}[2pt]
\Dotline(40,0)(144:30){1.5}[2pt]
\Dotline(40,0)(162:30){1.5}[2pt]}%
\end{picture}

```

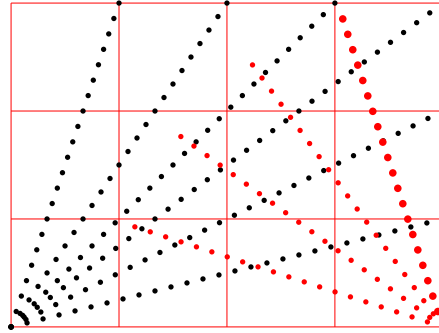


Figure 7: Different length dotted lines with the same nominal dot gap; again notice the relative polar coordinates for the dotted lines starting at the grid lower right vertex.

```

\unitlength =0.5mm
\begin{picture}(60,40)
\GraphGrid(60,40)
\Arc(0,20)(30,0){60}
\VECTOR(0,20)(30,0)\VECTOR(0,20)(32.5,36)
\VectorArc(0,20)(15,10){60}
\put(20,20){\makebox(0,0)[1]{$60^\circ$}}
\VectorARC(60,20)(60,0){-180}
\VVectorArc(60,20)(135:10){90}
\end{picture}

```

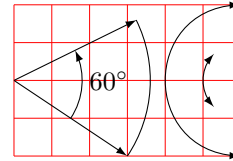


Figure 8: Arcs and curved vectors

The lengths of the original right triangle sides are a, b, c as marked on the figure; Point H divides the hypotenuse of length c in two shorter segments of lengths d, e , respectively, as marked in the figure. Since the three triangles are similar, we can set up the relationships between their sides:

$$d : a = a : c \implies d = a^2/c$$

$$e : b = b : c \implies e = b^2/c$$

Therefore rectangle CHFE area equals $c \cdot d = c \cdot a^2/c = a^2$. Similarly rectangle HBDF area equals $c \cdot e = c \cdot b^2/c = b^2$. The square built on the hypotenuse has an area equal to c^2 ; therefore it is

$$c^2 = a^2 + b^2$$

which proves the Pitagora's theorem.

14. A multitude of commands have been defined in order to manage complex numbers; actually complex numbers are represented as a comma separated

```

\unitlength =1mm
\begin{picture}(70,80)(-10,-50)
\AutoGrid
\thicklines
\polygon(0,0)(50,0)(18,24)
\thinlines
\polyline(0,0)(0,-50)(50,-50)(50,0)
\Pbox(0,0)[r]{C}[0.75ex]
\Pbox(18,0)[tr]{H}[0.75ex]
\Pbox(50,0)[l]{B}[0.75ex]
\Pbox(18,24)[b]{A}[0.75ex]
\Pbox(0,-50)[br]{E}[0.75ex]
\Pbox(50,-50)[bl]{D}[0.75ex]
\Zbox(18,-50)[br]{F}[0.75ex]
\Zbox(10,12)[br]{a}[0]
\Zbox(33,12)[bl]{b}[0]
\Zbox(25,0)[t]{c}[0]
\Zbox(9,0)[b]{d}[0]
\Zbox(30,0)[b]{e}[0]
\Zbox(50,-25)[r]{c}[0]
\Zbox(25,-50)[b]{c}[0]
\Zbox(0,-25)[l]{c}[0]
\Dashline(18,24)(18,-50){1.5}
\end{picture}

```

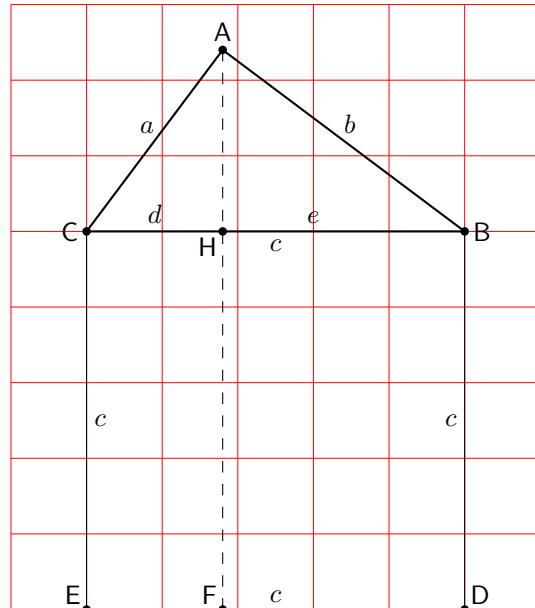


Figure 9: Geometrical construction to prove Pitagora's theorem

pair of fractional numbers; internally these macros use only the cartesian form, and output, unless differently specified is also in cartesian form; but input can be in polar form. They are used to address specific points in the drawing plane, but also as operators so as to scale and rotate other objects. In the following $\langle vector \rangle$ means a comma separated pair of fractional numbers, $\langle vector macro \rangle$ means a macro that contains a comma separated pair of fractional numbers; $\langle angle macro \rangle$ means a macro that contains the angle of a vector in sexagesimal degrees; $\langle argument \rangle$ means a brace delimited numeric value, even a macro; $\langle numeric macro \rangle$ means a macro that contains a fractional number; $macro$ is a valid macro name, i.e. a backslash followed by letters, or anything else that can receive a definition. A *direction* of a vector is its versor; the angle of a vector is the angle between the vector and the positive x axis in counterclockwise direction, as it is used in the Euler formula $\vec{v} = Me^{j\varphi}$.

```

\MakeVectorFrom<numeric macro><numeric macro>to<vector macro>
\CopyVect<first vector>to<second vector macro>
\ModOfVect<vector>to<modulus macro>
\DirOfvect<vector>to<versor macro>
\ModAndDirOfVect<vector>to<modulus macro>and<versor macro>
\ModAndAngleOfVect<vector>to<modulus macro>and<angle macro>
\DistanceAndDirOfVect<1st vector> minus<2nd vector>
to<distance macro> and<versor macro>
\XpartOfVect<vector>to<macro>
\YpartOfVect<vector>to<macro>

```

```

\DirFromAngle<angle>to<vector macro>
\ArgOfVect<vector>to<angle macro>
\ScaleVect<vector>by<scaling factor>to<vector macro>
\ConjVect<vector>to<conjugate vector macro>
\SubVect<subtrahend vector>from<minuend vector>to<vector macro>
\AddVect<first vector>and<second vector>to<vector macro>
\Multvect{<first vector>}*{<second vector>}*{<vector macro>} (the
    asterisks are optional; either one changes the second vector
    into its complex conjugate)
\MultVect<first vector>by<second vector>to<vector macro>
    (discouraged; maintained for backwards compatibility)
\MultVect<first vector>by*{<second vector>}to<vector macro>
    (discouraged; maintained for backwards compatibility)
\Divvect{<dividend vector>}{<divisor vector>}{<vector macro>}
\DivVect<dividend vector>by<divisor vector>to<vector macro>
    (discouraged; maintained for backwards compatibility)

```

15. General curves can be drawn with the `pict2e` macro `\curve` but it requires the specification of the third-order Bézier-spline control points; sometimes it's better to be very specific with the control points and there is no other means to do a decent graph; sometimes the curves to be drawn are not so tricky and a general set of macros can be defined so as to compute the control points, while letting the user specify only the nodes through which the curve must pass, and the tangent direction of the curve in such nodes. Such commands are the following:

- `\Curve` draws a sequence of arcs as explained above, using third order (cubic) Bézier splines. The starred version of this command fills the internal part of the curve with the current color; if the last arc finishes where the first arc starts, it is clear what is the interior; if it does not, the driver (not the code of this package, but the driver between this code and the physical representation on paper or screen) assumes a straight line closure of the whole path. The syntax offers several variants but it is substantially the following:

```

\Curve(<node><direction>>... (<node><direction>>
\Curve*(<node><direction>>... (<node><direction>>
    ...<direction>>[<new direction>](<node>)...

```

See some more explanation below.

- `\Qurve` is similar to `\Curve`, but with second order (quadratic) Bézier splines. The starred version fills the interior with the current color. Its syntax is similar to that of `\Curve`.
- `\CurveBetween` draws a single cubic Bézier spline between two given nodes and with two given direction vectors. This macro is similar to `\Curve`, but it is used to draw a third order Bézier curve between just two nodes and require just two directions
- `\CBezierBetween` draws a single cubic Bézier spline between two given nodes, with two given direction versors along which the control node distances are specified. This is the most general macro (rather difficult to use) with which not only the arc end points are

specified but also the control nodes coordinates are given. It is similar to `\CurveBetween` but the control points of the single arc can be specified; the arc is perfect, but the syntax is more complicated. See below for examples.

```
\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)<-0.1,-1.2>[-0.1,1.2]%
(1,5)<-1,0>(0,3.5)<0,-1>(2.5,0)<0.1,-1>
\end{picture}
```

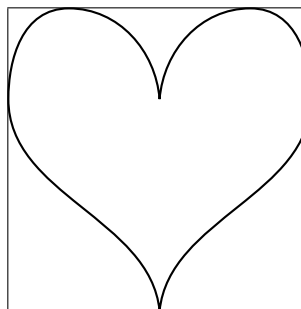


Figure 10: A heart shaped curve with cusps drawn with `\Curve`

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

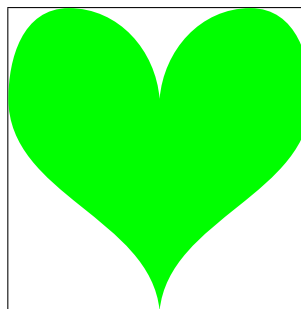


Figure 11: Coloring the inside of a closed path drawn with `\Curve*`

The main macro is `\Curve` and must be followed by an “unlimited” sequence of node-direction coordinates as a quadruple defined as

$$(\langle \text{node coordinates} \rangle) \langle \text{direction vector} \rangle$$

Possibly if a sudden change of direction has to be performed (cusp) another item can be inserted after one of those quadruples in the form

$$\dots (\langle \dots \rangle) \langle \dots \rangle [\langle \text{new direction vector} \rangle] (\langle \dots \rangle) \langle \dots \rangle \dots$$

Sometimes it is necessary to specify the “tension” or the “looseness” of a specific Bézier arc; such tension parameters range from 0 (zero) to 4; the zero value implies a very stiff arc, as if it was a string subject to a high tension (i.e. with zero looseness); a value of 4 implies a very low tension (very high looseness), almost as if the string was not subject to any tension. In METAFONT or METAPOST language such a concept is used very often; in this package, where the Hobby algorithms are not used, the parameter value appears to mean the opposite of tension. A couple of comma separated tension values may be optionally used, they

are separated with a semicolon from the direction vector, and they apply to the arc terminating with the last node; their specification must precede any possible change of tangent according to this syntax¹:

```
... (\node)<direction vector>;<start tension>,<end tension>>(\node)<direction>...
```

The `\Curve` macro does not (still) have facilities for cycling the path, that is to close the path from the last specified node-direction to the first specified node-direction; but, as already mentioned, if the ending node of the last arc does not coincide with the starting node of the first arc, a straight line is assumed to join such nodes; this line does not get drawn, but with starred commands no lines are drawn because only the interior is coloured. The tangent direction need not be specified with a unit vector, although only its direction is relevant; the scaling of the specified direction vector to a unit vector is performed by the macro itself. Therefore one cannot specify the fine tuning of the curve convexity as it can be done with other programs or commands, as, for example, with `METAFONT` or the `pgf/tikz` package and environment. See figure 10 for an example.

With the starred version of `\Curve`, instead of stroking the contour, the macro fills up the contour with the selected current color, see figure 11.

Figure 12 shows a geometric construction that contains the geometric elements and symbols used to determine the parameters of a cubic spline required to draw a quarter circle. This construction contains many of the commands described so far.

To show what you can do with `\CurveBetween` see the code and result shown in figure 13. Notice the effect of changing the directions at both or at the end nodes of a single cubic spline. The directions are conveniently expressed with unit vectors described by polar coordinates. The `\CurveBetween` macro is built on `\CBezierBetween`; this latter command is very complicated to describe and its use is reserved to experienced users; its syntax is described in the code documentation file `cureve2e.pdf`; the reader is encouraged to examine it in case s/he is willing to use it.

A little more complicated is the use of the `\CBezierBetween` macro, figure 14. The directions are specified with unit vectors in polar form; the control points are specified by adding their distances from their neighbouring nodes; actually the right distance is maintained to the value 1, while the left one increases from 4 to 10. The black line corresponds to the standard `\CurveBetween` where the default distance is computed to trace an arc of a circle and is approximately 3.5.

In figure 15 the effect of tension specification is shown. The red line corresponds to the default tension, since the tension values are not specified. The black lines correspond to the various values used in the various commands to the `\Curve` macro. With a tension of zero, the spline is almost coincident with the horizontal base line of the frame. Increasing the parameter value to 4.5, the curved becomes taller and taller, until it wraps itself displaying an evident loop. We would say that the value of 2 is a reasonable maximum one and that increasing that value is just to obtain special effects.

¹The tension may be specified only for cubic splines, because the quadratic ones do not use enough parameters to control the tension; not all commands for drawing cubic splines accept this optional tension specification.


```

\unitlength=0.007\textwidth
\begin{picture}(100,90)(-50,-50)
\put(-50,0){\vector(1,0){100}}\put(50,1){\makebox(0,0)[br]{\mathit{x}}}%
\put(20,-1){\makebox(0,0)[t]{\mathit{s}}}%
\put(0,0){\circle*{2}}\put(-1,-1){\makebox(0,0)[tr]{\mathit{M}}}%
\legenda(12,-45){s=\overline{MP_2}=R\sin\theta}%
\put(0,-50){\vector(0,1){90}}%
\put(1,40){\makebox(0,0)[t1]{\mathit{y}}}%
\put(0,-40){\circle*{2}}\put(1,-41){\makebox(0,0)[lt]{\mathit{C}}}%
\segment(0,-40)(-40,0)\segment(0,-40)(40,0)%
\put(-41,1){\makebox(0,0)[br]{\mathit{P}_1}}\put(-40,0){\circle*{2}}%
\put(41,1){\makebox(0,0)[bl]{\mathit{P}_2}}\put(40,0){\circle*{2}}%
\put(0,0){\linethickness{1pt}\Arc(0,-40)(40,0){90}}%
\segment(-40,0)(-20,20)\put(-20,20){\circle*{2}}%
\put(-20,21.5){\makebox(0,0)[b]{\mathit{C}_1}}%
\segment(40,0)(20,20)\put(20,20){\circle*{2}}%
\put(20,21.5){\makebox(0,0)[b]{\mathit{C}_2}}%
\put(0,-40){\put(0,56.5685){\circle*{2}}}%
\put(1,58){\makebox(0,0)[bl]{\mathit{P}}}%
\VectorARC(0,-40)(15,-25){45}\put(10,-18){\makebox(0,0)[c]{\mathit{\theta}}}%
\VectorARC(40,0)(20,0){-45}\put(19,5){\makebox(0,0)[r]{\mathit{\theta}}}%
\VectorARC(-40,0)(-20,0){45}\put(-19,5){\makebox(0,0)[l]{\mathit{\theta}}}%
\put(-20,-18){\makebox(0,0)[bl]{\mathit{R}}}%
\put(-32,13){\makebox(0,0)[bl]{\mathit{K}}}%
\put(32,13){\makebox(0,0)[br]{\mathit{K}}}%
\end{picture}

```

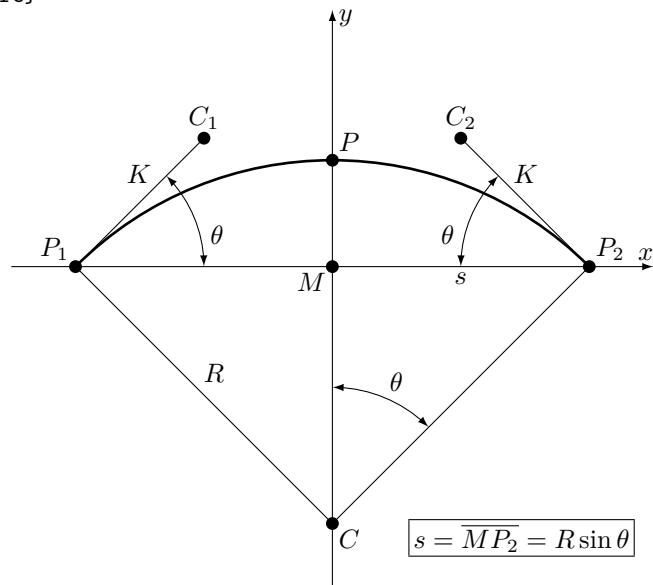


Figure 12: The code to display the nodes and control points for an arc to be approximated with a cubic Bézier spline

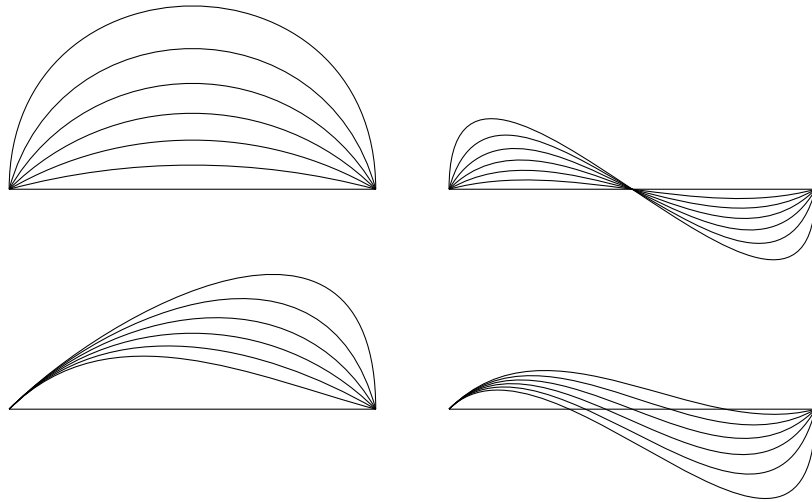


Figure 13: Curves between two points with different start and end slopes

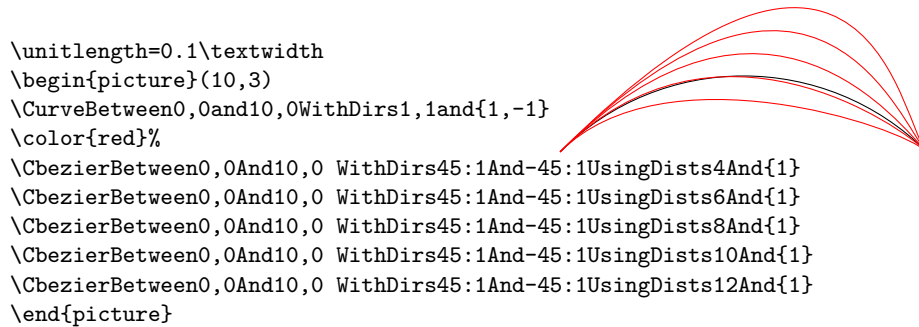


Figure 14: Comparison between similar arcs drawn with `\CurveBetween` with the default tension values (black) and with several tension values specified (red)

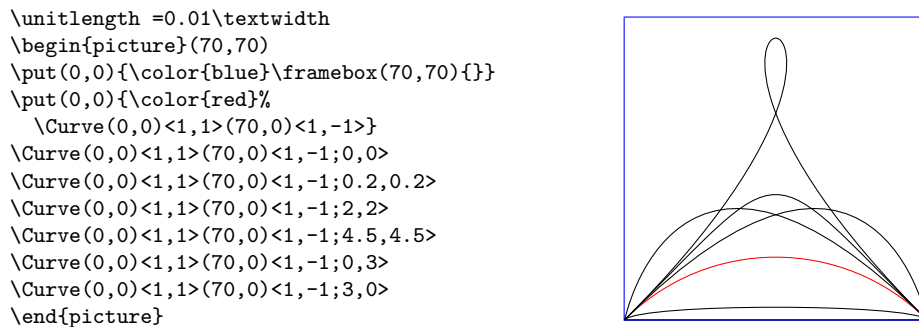


Figure 15: The effects of tension factors

Figure 16 displays two approximations of a sine wave; Bézier splines can approximate transcendental curves, but the approximation may be a poor one, depending on the approximated curve, when few arcs are used to draw it. With arcs specified with more complicated macros the approximation is better even with a lower number of arcs. With many arcs it is

possible to approximate almost anything. On the left side of figure 16 a modest approximation is obtained with just three standard arcs obtained with `\Curve` and four node specifications; on the right we have just two arcs created with `\CBezierBetween` with tension specification and control point distances; this drawing is almost undistinguishable from a real sinusoid.

```

\unitlength=0.01\textwidth
\begin{picture}(100,50)(0,-25)
\put(0,0){%
\VECTOR(0,0)(45,0)\VECTOR(0,-25)(0,25)
\Pbox(45,0)[b]{x}[0]\Pbox(0,26)[t1]{y}[0]}
\Curve(0,0)<77:1>(10,20)<1,0;2,0.4>(30,-20)<1,0;0.4,0.4>(40,0)<77:1;0.4,2>
}
\put(55,0){%
\VECTOR(0,0)(45,0)\VECTOR(0,-25)(0,25)
\Pbox(45,0)[b]{x}[0]\Pbox(0,26)[t1]{y}[0]}
\CBezierBetween0,0And20,0WithDirs77:1And-77:1UsingDists28And{28}
\CBezierBetween20,0And40,0WithDirs-77:1And77:1UsingDists28And{28}}
\end{picture}

```

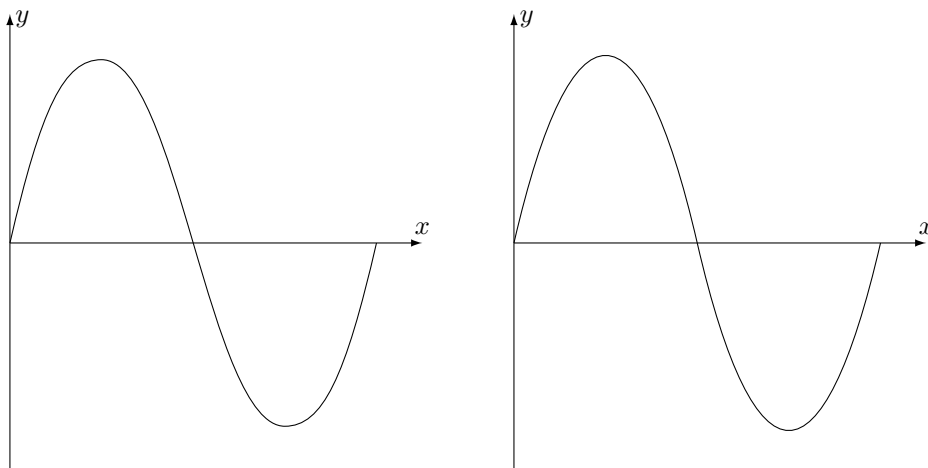


Figure 16: A sequence of arcs; the left figure has been drawn with the `\Curve` command with a sequence of four couples of node-direction arguments; the right figure has been drawn with two commands `\CBezierBetween` that include also the specification of the control points

In figure 17 some lines are shown; they are drawn with quadratic splines by means of the `\Qurve` macro. In the left there are some open and closed curves inscribed within a square. On the right a “real” circle is compared to a quadratic spline circle; the word “real” is emphasised because it actually is an approximation with four quarter-circle cubic splines that, in spite of being drawn with third degree parametric polynomials, approximate very well a real circle; on the opposite the quadratic spline circle is clearly a poor approximation even if the maximum radial error amounts just to about 6% of the radius.

Notice that the previous version of `curve2e` contained an error and would

```

\unitlength=0.0045\textwidth
\begin{picture}(100,100)
\put(0,0){\framebox(100,100){}}
\put(50,50){%
\Curve(0,-50)<1,0>(50,0)<0,1>(0,50)<-1,0>(-50,0)<0,-1>(0,-50)<1,0>
\color{green}
\Curve*(0,-50)<0,1>(50,0)<1,0>[-1,0](0,50)<0,1>[0,-1](-50,0)<-1,0>[1,0](0,-50)<0,-1>
}
\Curve(0,0)<1,4>(50,50)<1,0>(100,100)<1,4>
\put(5,50){\Curve(0,0)<1,1.5>(22.5,20)<1,0>(45,0)<1,-1.5>%
(67.5,-20)<1,0>(90,0)<1,1.5>}
\Zbox(0,0)[tc]{0,0}\Zbox(100,0)[tc]{100,0}
\Zbox(100,100)[bc]{100,100}\Zbox(0,100)[bc]{0,100}
\Pa11[2](0,0)\Pa11[2](100,0)\Pa11[2](100,100)\Pa11[2](0,100)
\end{picture}
\hfill
\begin{picture}(100,100)
\put(0,0){\framebox(100,100){}}
\put(50,50){%
\Curve(0,-50)<1,0>(50,0)<0,1>(0,50)<-1,0>(-50,0)<0,-1>(0,-50)<1,0>
\Curve(0,-50)<1,0>(50,0)<0,1>(0,50)<-1,0>(-50,0)<0,-1>(0,-50)<1,0>}
\Zbox(50,50)[t]{0}\Pa11[2](50,50)\put(50,50){\Vector(45:50)}\Zbox(67,70)[t1]{R}
\end{picture}

```

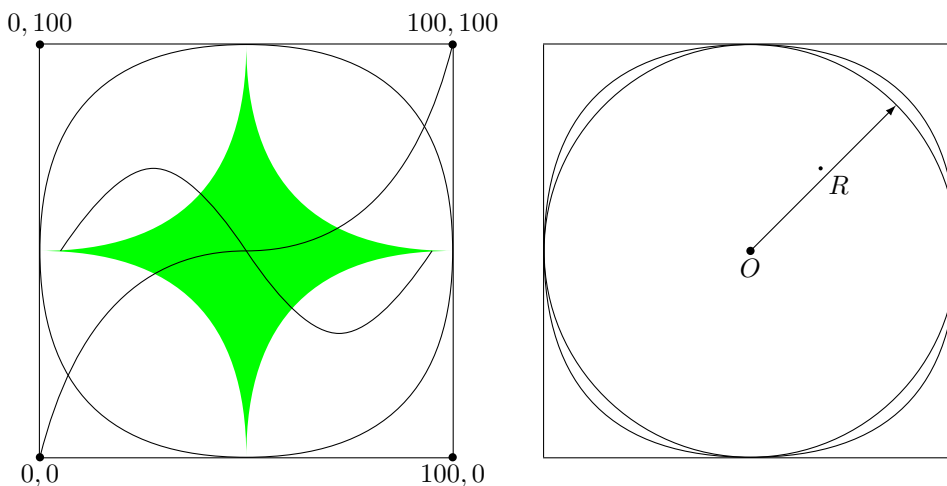


Figure 17: Several graphs drawn with quadratic Bézier splines. On the right a quadratic spline circle is compared with a cubic line circle.

color the outside of the green four-pointed star. The `curve2e-v161` package, attached to this bundle, has been corrected; therefore it is not actually identical to the previous version, although the latter one performed correctly for everything else except for color filled quadratic paths.

16. The new version of `\multiput` is backwards compatible with the original version contained in the \LaTeX kernel. The new macro adds the handling of the coordinate increments from one position to the next for the $\langle object \rangle$ to include in the drawing. The syntax is the following:

```

\multiput( $\langle initial point \rangle$ )( $\langle increment \rangle$ ){ $\langle number of items \rangle$ }{ $\langle object \rangle$ }[ $\langle handler \rangle$ ]

```

The only small change is the addition of the last optional argument that allows to do several interesting actions on the sequence of objects to be

repeated, for example, to set them on a curved line, instead of on a straight line, as shown in figure 18.

In this figure we show the code for the picture shown there. The red grid is nothing new, except that it displays the traditional `\multiput` used in this code, shown in a previous example, produces exactly the same result. But for what concerns the four “graphs” on the grid, it displays an alignment of black dots along the diagonal of the grid (again traditional `\multiput` rendered with the new version); a number of blue dots along a parabola; another number of magenta coloured dots alined along a half sine wave; a number of little green squares aligned along a -15° sloping line starting from the center of the grid; notice the polar values that are used as polar relative coordinate increments.

```

\unitlength=0.01\linewidth
\begin{picture}(100,100)
\GraphGrid(100,100)
\multiput(0,0)(10,10){11}{\circle*{2}}
\color{blue!70!white}
\multiput(0,0)(10,0){11}{\circle*{2}}%
  [\GetCoord(\R)\X\Y
  \edef\X{\fpeval{\X+10}}
  \edef\Y{\fpeval{\X/10**2}}
  \CopyVect\X,\Y to\R]
\color{magenta}
\multiput(0,0)(10,1){11}{\circle*{2}}%
  [\GetCoord(\R)\X\Y
  \edef\X{\fpeval{\X+10}}
  \edef\Y{\fpeval{\sind(\X*1.8)*100}}
  \CopyVect\X\Y to\R]
\color{green!80!black}
\multiput(50,50)(-15:5){11}{%
\polygon*(-1,-1)(1,-1)(1,1)(-1,1)}
\end{picture}

```

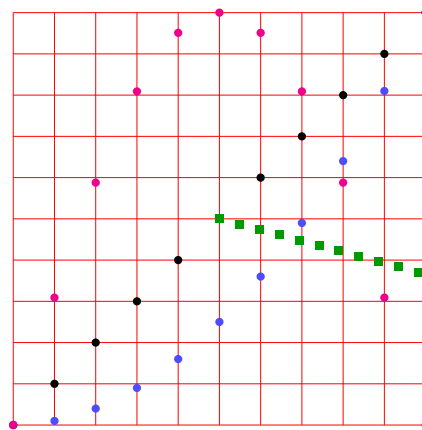


Figure 18: Some examples of the *handler* optional argument

A new command `\xmultiput` (not available with the previous versions of `curve2e`) is extended with respect to the original `\multiput`; it is defined by using some L3 functions; in particular the cycling counter is accessible to the `LATEX` commands and it is stepped up from 1 to the value specified in the proper command argument (in the original command it starts from that value and is stepped down to zero). See the figure on page 19 to inspect its usage. It is important to notice that if the command `\rotatebox` has to be used, as in the example of figure 19, the package `graphicx` should be also loaded, because `curve2e` does not load it.

The `\xmultiput` syntax is similar to that of `\multiput` but besides the stepping up or down of the iteration counter, it can access and modify certain internal with the commands that appear in the *handler* argument. Actually this *handler* is available also with `\multiput`. In both cases the handler can be defined to modify some internals, including the iteration counter only for `\xmultiput`, but also the `\R` and `\D` internals; `\R` contains the coordinates where to put the *object*, while `\D`, if set, contains the angle of rotation of the object. The code and picture examples

in figures 18 and 19 show some examples of usage through tis *handler* code. .

```
\xmultipt(<initial point>)(<increment>){<iterations>}{<object>}[<handler>]
```

```
\unitlength=0.0095\linewidth
\begin{picture}(100,100)
\GraphGrid(100,100)
\put(50,50){\thicklines\circle{100}}
\xmultipt [50,50] (60:40) (-30:1){12}%
  {\makebox(0,0){\circle*{2}}}%
  [\MultVect\R by\D to\R]%
\xmultipt [50,50] (60:46) (-30:1){12}%
  {\ArgOfVect\R to\Ang
  \rotatebox{\fpeval{\Ang-90}}%
  {\makebox(0,0)[b]{\Roman{multicnt}}}}%
  [\Multvect{\R}{\D}\R]
\end{picture}
```

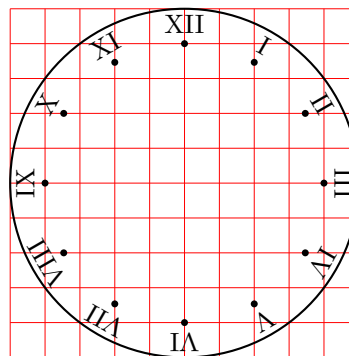


Figure 19: Usage example of the `\xmultipt` command

- This implementation of `curve2e` includes an extension to package `xfp`, in the sense that adds three more L3 commands: `\fpctest`, `\fpdowhile`, `\fpwhiledo` to the two already contained and documented in the latter package. The syntax of such new commands is the following

```
\fpctest{<test>}{<true>}{<false>}
\fpdowhile{<test>}{<operations to be repeated>}
\fpwhiledo{<test>}{<operations to be repeated>}
```

The macro `\fpctest` requires two further arguments that contain what to do if the `<test>` is true, and what to do if the `<test>` is false. The `<test>` is a logical expressions that connects math relation expressions, even floating point ones, by means of *logical operators*; such operators are `||`, `&&`, and `!`, respectively for OR, AND, NOT; math relation expressions contain relation operators, even negated ones: for example `!<` means “not lower than”, which is equivalent to “equal or grater than”, i.e. `=>`. The logical expression is parsed left to right and normal parentheses may be used to alter this sequence. The logical operators work also between logical variables, therefore the `<test>` may contain an interesting mixture of relation and logical operators.

Before using `\fpdowhile` and `\fpwhiledo`, the arguments of `<test>` depends on must be set so that the test is true; during the execution of the `<operations to be repeated>` there must be some setting that eventually renders the `<test>` false. The user should pay attention to set the elements that `<test>` depends on, because the risk is to enter an infinite loop and end up with some error message stating that the working memory of the program is full. Notice that `\fpdowhile` first puts the `<operations to be repeated>` into the work flow then checks the `<test>` and possibly repeats the cycle; on the opposite, `\fpwhiledo` first checks the `<test>` then possibly inserts

the *operations to be repeated* and cycles. Evidently with the same *test* the two while cycles produce different results with a little, but important difference: if the input data are macros defined by previous computations, there is no guarantee that the *test* is initially true; if it is false, `fpwhiledo` does not do anything, while `\fpdowhile` executes one cycle and produces in the output stream something that might be nonsense. Cycles done with `\fpwhiledo` should be safer and should be preferred. Nevertheless such commands are very useful also for drawing graphics; the `xmultiput` command already makes use of such L3 functions. As an example of use, we show how to plot a mathematical function expressed in parametric form:

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

The plot is executed with a piecewise linear approximation of the curve; if the t steps are sufficiently small, the plot turns out to be very nice; here we show an example where we plot a Lissajous curve with two sinusoids of different periods.

We start by defining the Lissajous function with arguments to specify the parameter t , the sinusoid amplitudes A_1, A_2 , the respective “frequencies”, by means of integer multiples of a unit pulsation, N_1, N_2 , and the initial phases ϕ_1, ϕ_2 of such sinusoids. It is better to keep apart the input of the curve coefficients from the actual curve argument/parameter and output point coordinates:

```
\def\LissajousCoefs#1,#2,#3,#4,#5,#6!{%
\edef\LAu{#1}\edef\LNu{#2}\edef\LFu{#3}%
\edef\LAd{#4}\edef\LNd{#5}\edef\LFd{#6}}

\def\LissajousCode#1#2{%
\edef\X{\fpeval{\LAu*cosd(\LNu*#1+\LFu)}}%
\edef\Y{\fpeval{\LAd*cosd(\LNd*#1+\LFd)}}%
\CopyVect\X,\Y to#2\ignorespaces}
```

As it is shown, the coefficients are specified as a comma separated list; the ! list terminator is taken care by the actual drawing command.

Then the curve drawing command requires the coefficient specification only the first time it is used; some messages² are output if the coefficients have been “forgotten”.

```
\NewDocumentCommand\Lissajous{m o m}{%
\IfValueTF{#2}{\LissajousCoefs#2!\relax
\LissajousCode{#1}{#3}}%
{\ifcsname LAu\endcsname
\LissajousCode{#1}{#3}}%
\else\PackageError{curve2e}{%}
```

²Here it would be better to have available a “Macro Error” message; such a macro is not available, but it would be possible to define it by means of the `\GenericError` macro provided by the L^AT_EX₂ε kernel. Here we skip this definition in order to avoid overloading this documentation with such details.

```

{This Lissajous' curve coefficients\MessageBreak
are missing}\Nothing done}\fi}%
\ignorespaces}

```

The syntax is the following:

$$\backslash\text{Lissajous}\{\langle in \rangle\}[\langle A_1 \rangle, \langle N_1 \rangle, \langle \phi_1 \rangle, \langle A_2 \rangle, \langle N_2 \rangle, \langle \phi_2 \rangle][\langle P_{out} \rangle]$$

where $\langle P_{out} \rangle$ is a macro that gets defined with the cartesian coordinates of the computed output point. Arguments $\langle in \rangle$ (the t parameter) and $\langle P_{out} \rangle$ (the computed coordinates) need not be enclosed within braces if they are given as macros; actually the code shown in figure 20 shows such procedure that renders the input code simpler to read.

After this definition the diagram is plotted in figure 20.

```

\unitlength=0.01\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)[tr]{y}[0]
\Pbox(0,0)[tr]{0}[2]
\thicklines
{\countdef\I=2560 \I=0
\fpdowhile{\I !=> 360}{%
\fpctest{\I=0}%
{\Lissajous\I[40,2,90,40,3,0]\Pout
\moveto(\Pout)}%
{\Lissajous\I\Pout
\lineto(\Pout)}%
\advance\I by1}\strokepath}%
\end{picture}

```

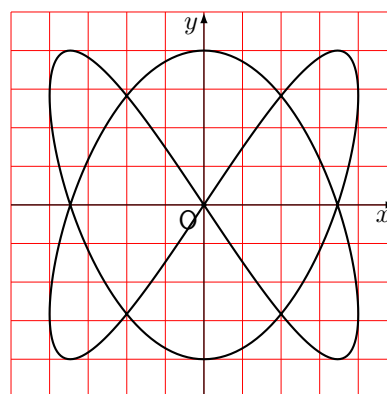


Figure 20: A Lissajous diagram

For the independent variable t , the parameter of the Lissajous parametric equations, it is better to work with degrees instead of radians, and with integer numbers, so that the whole range from 0° to 360° is certainly spanned. Notice the braces that include the code for the Lissajous diagram; they may be useful to render that group suitable to be `\put` somewhere else than with its center at the origin of the canvas axes, and/or to be used as the second argument of a `\rotatebox{\langle angle \rangle}` command so as to rotate the whole diagram.

18. Another useful application of `\fpdowhile` is the following: when making a diagram the axes should get suitable labeled ticks in order to show the graduations; the label of each tick should lay close to the axis on the other side than the tick. It is also necessary to know if the axis to be marked is horizontal or vertical, since in the former case each tick is vertical, while in the latter case it is horizontal. The example in figure ?? shows both the code and its usage. The syntax of the `\Tbox` macro is the following.

$$\backslash\text{Tbox}(\langle coordinates \rangle)[\langle reference \rangle]\{\langle label \rangle\}[\langle size \rangle]\langle direction \rangle$$

where $\langle coordinates \rangle$ indicates the position of the tick base along its axis; $\langle reference \rangle$ is a letter, either `t` (if the axis is on top of the label) or `b` (if

the axis is at the bottom of the label) for vertical ticks or either `r` (if the axis is at the right of the label) or `l` (if the axis is at the left of the label) for horizontal ticks; $\langle label \rangle$ is the value of the scale or a literal label that by default is typeset in math mode, so that a math symbol may be used without the need of entering math mode, while in the unusual circumstance that a textual label is to be used, users should use the `\text` macro; the $\langle size \rangle$ argument is the size of the tick: if such size is zero, just the label is set, but in this case the `\Tbox` macro behaves as the `\Zbox` one, and accepts two reference codes in order to print the label the same as `\Zbox` would do when the dot size is zero; the $\langle direction \rangle$ argument is the letter `V` for vertical ticks, and `H` for horizontal ticks. There is some redundancy because `\Tbox` may behave as `\Zbox`, but experience shows that this is not a problem.

The source preamble should contain the following code.

```

\makeatletter
\NewDocumentCommand\Tbox{D()}{0,0} O{cc} m O{0pt} D<>{Z}}{\bgroup
\edef\TBoxCode{#5}\dimen0=#4\relax %
\edef\tempE{\fpeval{round(#4/\unitlength,3)}}%
\put{#1}{%
\ifdim\dimen0=\z@
  \Zbox(0,0)[#2][#3][\z@]%
\else
  \if\TBoxCode V\relax
  \let\tempC=b\relax
  \if\tempC #2\edef\tempD{0,-\tempE}\else\edef\tempD{0,\tempE}\fi
  \segment(0,0)(\tempD)\Zbox(0,0)[#2][#3][\z@]%
  \else
  \if\TBoxCode H\relax
  \let\tempC=l\relax%
  \if\tempC #2\edef\tempD{-\tempE,0}\else\edef\tempD{\tempE,0}\fi
  \segment(0,0)(\tempD)\Zbox(0,0)[#2][#3][\z@]%
  \else
  \typeout{The specified code\space #5\space is invalid!}%
  \typeout{\string\Tbox\space command ignored}%
  \fi
  \fi
\fi
}\egroup\ignorespaces}%

```

If the code is contained in a personal `.sty` file the `\makeatletter` should be omitted.

As it can be seen, the fifth argument code is preset to to `Z` so that if users forget to specify it, error messages pop up and can be seen in both the console window and in the log file, but no tick and label are typeset. This kind of commands to label in one way or another may be very useful depending on the users' kind of drawings. Figure 21 contains also the iterations to label the axes and are done with the `\fpdowhile` macro; it can be seen that the actual coordinate of each tick is transformed into the actual label that is put in its position by the `\Zbox` command' of course the diagram scale and the actual label are related to one another but do not have the same value.

```

\centering\unitlength=0.007\linewidth
\begin{picture}(100,70)(-50,-10)%
\AutoGrid
\thicklines
\VECTOR(-50,0)(50,0)\Zbox(50,0)[br]{x}[0pt]%
\VECTOR(0,0)(0,60)\Zbox(0,60)[t1]{y}[0pt]%
%
{\def\Coord{-40}%
\fpdowhile{\Coord<=40}{\edef\X{\fpeval{round(\Coord/20,1)}}%
\Tbox(\Coord,0)[t]{\X}[1.5mm]<V>\edef\Coord{\fpeval{\Coord+10}}}%
%
\def\Coord{10}%
\fpdowhile{\Coord<=50}{\edef\Y{\fpeval{round(\Coord/20,1)}}%
\Tbox(0,\Coord)[r]{\Y}[1.5mm]<H>\edef\Coord{\fpeval{\Coord+10}}}%
}%
%
{\color{blue}\linethickness{1.5pt}%
\Curve(-40,0)<1,0>(0,20)<45:1>% Parabola:  $y=0.25(x_2)^2$ 
\Curve(0,20)<1,0>(20,0)<0,-1>% Quarter circle:  $x^2+y^2=1$ 
\Curve(20,0)<0,1>(40,34.64)<49:1>% Hyperbola:  $x^2-y^2=1$ 
\thinlines
\Dashline(0,0)(45,45){2}% asymptote
\end{picture}

```

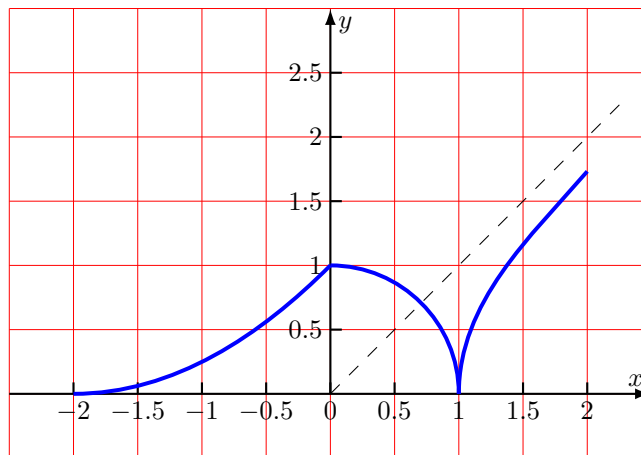


Figure 21: The code to draw a diagram with labeled axes

4 Remarks

In spite of the relative simplicity of the macros contained in this package, the described macros, as well as the original ones included in the `pict2e` package, allow to produce fine drawings that were unconceivable with the original \LaTeX picture environment. Leslie Lamport himself announced an extension to his environment when $\text{\LaTeX} 2_{\epsilon}$ was first released in 1994; in the `ltnews` newsletter of December 2003, the first implementation of Lamport's extension was announced; the first version of this package `curve2e` was issued in 2006. It was time to have a better drawing environment; this package is a simple attempt to follow the initial path while further extending the drawing facilities.

There are other packages in the CTAN archives that deal with tracing curves of various kinds. `PSTricks` and `pgf/tikz` are the most powerful ones. And they are becoming the standard for computer drawing. Their documentation is huge and the multitude of extra modules to perform special tasks is countless. Therefore they are difficult to use; when the user gets used to their particular syntax and got sufficient familiarity with several modules, s/he can use these bundles very comfortably.

This difficulty in becoming a TikZ or PS expert is why I think a simpler drawing machinery should be appreciated. I admit it: I like the `picture` environment; and I like to deal with simple codes so as to create my own macros.

But there is also `curves` that is intended to draw almost anything by using little dots or other symbols partially superimposed to one another. It uses only quadratic Bézier curves and the curve tracing is eased by specifying only the curve nodes, without specifying the control nodes; with a suitable package option, it is possible to reduce the memory usage by using short straight segments drawn with the PostScript facilities offered by the `dvips` driver.

Another package, `ebezier` performs about the same as `curve2e` but draws its Bézier curves by using little dots partially superimposed to one another. The documentation is quite interesting since it explains very clearly what exactly are the Bézier splines. Apparently `ebezier` should be used only for DVI output without recourse to PostScript or PDF machinery.

The `picture` package extends the performance of the `picture` environment (extended with `pict2e`) by accepting coordinates and lengths in real absolute dimensions, not only as multiples of `\unitlength`; it provides commands to extend that functionality to other packages. In certain circumstances it may be very useful.

Package `xpicture` builds over the `picture` \LaTeX environment so as to allow to draw the usual curves that are part of an introductory analytic geometry course; lines, circles, parabolas, ellipses, hyperbolas, and polynomials; the syntax is rather comfortable, although it is not a simple extension of the `picture` own syntax; for all these curves it uses the quadratic Bézier splines.

Package `hobby` extends the cubic Bézier spline handling with the algorithms John Hobby created for `METAFONT` and `METAPOST`. But by now this package interfaces very well with `tikz`; it has not (yet) been adapted to the common `picture` environment extended with `pict2e`, and, why not, with `curve2e`.

If you are interested in further extensions of `curve2e`, besides creating yourself the macros you need, examine the `euclideangeometry` package; I created it in order to have available further functionalities useful to deal with more complicated geometrical constructions. Another small extension is `graphpaper` by

which users can produce themselves drawing paper with bilinear coordinates, semi-logarithmic and bi-logarithmic coordinates, polar linear or semilogarithmic coordinates, and Smith charts. They are an extension chain of the packages that extend the `picture` environment: `graphpaper` calls `euclideangeometry` which calls `curve2e` which calls `pict2e`. I find all this very useful. Of course these packages and their documentation are all contained in T_EX Live.

I am about to complete a book on electromagnetism and electronic circuit theory: it contains some 300 drawings, diagrams, electronic circuits. I created them with `curve2e` and some macros based on this `picture` extension package. The graphics of my book are pretty nice, and I am sure that a professional technical artist can draw better ones, but such an artist would be very expensive and the process would require a lot of time to correct, modify, add details, and so on, while doing things by oneself is certainly a better solution. Packages `tikz/pgfplots` and `PStricks`? Certainly they are very good and much more powerful, but, besides being sort of difficult to learn to use, they require a lot of working memory and more often than not I ran out of computer memory.

5 Acknowledgements

I wish to express my deepest thanks to Michel Goosens who spotted some errors and very kindly submitted them to me so that I was able to correct them.

I collaborated extensively with Josef Tkadlec in order to implement a better real long division so as to get correctly the quotient fractional part and to avoid as much as possible any numeric overflow; many Josef's ideas are incorporated in the macro that was implemented in the previous version of this package, although the macro used by Josef was slightly different. Both versions aim/aimed at a better accuracy and at widening the operand ranges. In this version of `curve2e` I abandoned our long division macro, and substituted it with the floating point division provided by the `xfp` package functionalities that are now part of the L^AT_EX kernel.

Daniele Degiorgi spotted a fault in the kernel definition of `\linethickness` that heavily influenced also `curve2e`; see in the code documentation `curve2e.pdf` file.

Many thanks to Domenicus van der Wijst who spotted a sneaky bug in the updated `\Arc` macro; for arcs not wider than 180° it worked properly, while for larger angle apertures it produced syntax errors. It was just a typo, but very sneaky.

Thanks also to Jin-Hwan Cho and Juho Lee who suggested a small but crucial modification in order to have `curve2e` work smoothly also with XeTeX (XeLaTeX). Actually if `pict2e`, version 0.2x or later, dated 2009/08/05 or later, is being used, such modification is not necessary any more, but it is true that it becomes imperative when legacy versions were used.

Many thanks also to Ashish Kumar Das who spotted an inconsistency in the design of vectors with PostScript style arrow tips with large line width settings, that did not show up with L^AT_EX styled ones.