The extension package curve2e

Claudio Beccari*

Version v.2.0.1 – Last revised 2019-03-29.

Contents

1 The configuration file 1

2 Package pict2e and this extension curve2e 2

3 Summary and examples of new commands 5

4 Remark 17

5 Acknowledgements 19

Abstract

This file documents the curve2e extension package to the pict2e bundle implementation that has been described by Lamport himself in the 1994 second edition of his LaTeX handbook.

Please take notice that in April 2011 a new updated version of the package pict2e has been released that incorporates some of the commands defined in early versions of this package; apparently there are no conflicts, but only the advanced features of curve2e remain available for extending the above package.

This extension redefines a couple of commands and introduces some more drawing facilities that allow to draw circular arcs and arbitrary curves with the minimum of user intervention. This version is open to the contribution of other users as well as it may be incorporated in other people's packages. Please cite the original author and the chain of contributors.

1 The configuration file

This package curve2e is distributed with a ltxdoc.cfg configuration file that contains, besides the preamble and the postamble comment lines, the following code line:

```
\AtBeginDocument{\OnlyDescription}
```

If you want to type the whole documentation, comment out that code line in the ltxdoc.cfg file. This is the only modification allowed by the LPPL licence that does not require to change the file name.

For your information the initial part is about 20 pages long; the whole documentation is about 80 pages long.

*E-mail: claudio dot beccari at gmai dot com

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 has been accompanying every release of LaTeX 2_ε 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, that is 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 were also limited in their radii 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 and would not increase proportionally to the oval size.
5. Vector arrows 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.

The 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 my be specified in both cartesian and polar form: internally they are handeld as cartesian coordinates, but the user can specify his/her points also in polar form. In order to avoid confusion with other grapgic packages, `curve2e` uses the usual comma separated couple of integer or fractional numebrs for cartesian coordinates, and the couple $\langle angle\rangle:\langle radius\rangle$ for polar coordinates. All graphic object commands accept polar or cartesian coordinates at the choice of the user who may use for each object the formalism he/she prefers. Also the `put` and `\multiput` commands have been redefined so as to accept cartesian or polar coordinates.

2. Most if not all cartesian 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 he/she 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.

3. Commands for setting the line terminations are introduced; the user can chose between square or rounded caps; the default is set to rounded caps (now this original feature is directly available also with `pict2e`).

4. Commands for specifying the way two lines or curves join to one another.

5. originally the `\line` macro is redefined so as to allow integer and fractional direction coefficients, but maintaining the same syntax as in the original `picture` environment; now this functionality available directly with `pict2e`.

6. A new macro `\Line` was originally defined so as to avoid the need to specify the horizontal projection of inclined lines; now this functionality id available directly with `pict2e`; but this macro name now conflicts with `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).

7. A new macro `\LINE` was defined in order to join two points specified with their coordinates; this is now the normal behavior 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 line specification.

8. A new macro `\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.

9. A 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.

10. Similar macros are redefined for vectors; `\vector` redefines the original macro but with the vector slope limitations removed; `\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.

11. 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

12. The `pict2e polygon` macro to draw closed polylines, in practice general polygons, has been redefined in such a way that it can accept the various vertices specified with 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.

13. 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 the interior of the generated circular arc with the current color. 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.

14. Two new macros `\VectorArc` and `\VectorARC` are defined in order to draw circular arcs with an arrow at one or both ends.

15. 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.

16. `\Curve` is a recursive macro that can draw an unlimited (reasonably limited) number of connected Bézier spline arcs with continuous tangents except for cusps; these arcs require only the 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.

17. 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 form 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 10. 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`; 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 control points, while here the control points are sort of rigid. It would be very useful to resort to the `hobby` package, but its macros are conforming 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 been 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, the `xfp` has recently become available, by which any sort of calculations can be done with floating point 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 I said before, might be redefined so as to use the macros introduced in 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`

to which it passes any possible option that the latter can receive; actually the only options that make sense are those concerning the arrow tips, either LaTeX or PostScript styled, because it is assumed that if you use this package you are not interested in using the original LaTeX commands. See the `pict2e` documentation in order to see the correct options `pict2e` can receive.

2. 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 command in order to restore the previous terminator or join.

3. The commands `\linethickness`, `\thicklines`, `\thinlines` together with `\defaultlinethickness` always redefine the internal `\@wholewidth` and `\@halfwidth` so that the latter always refer to a full width and to a half of it in this way: if you issue the command `\defaultlinewidth{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, et cetera. It's better to recall that `\thinlines` and `\thicklines` are declarations that do not take arguments; on the opposite the other two commands follow the standard syntax:

   `\linethickness{`⟨*dimensioned value*⟩`}`
   `\defaultlinewidth{`⟨*dimensioned value*⟩`}`

   where ⟨*dimensioned value*⟩ means a length specification complete of its units or a dimensional expression.

4. Straight lines and vectors are redefined in such a way that fractional slope coefficients may be specified; the zero length line does not produce errors and is ignored; the zero length vectors draw only the arrow tips.

```
\unitlength=.5mm
\begin{picture}(60,20)
\put(0,0){\GraphGrid(80,20)}
\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}
```

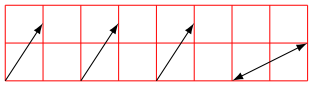

Figure 1: Three (displaced) identical vectors obtained with the three vector macros; a double tipped vector is also shown.

5. 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 command `\put`. 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 this new release of the `curve2e` package. Experience has shown that the commands intended to join two specified points are particularly useful.

6. The `\polyline` command has been introduced: it accepts an unlimited list of point coordinates 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*⟩`](`⟨P_1⟩`)(`⟨P_2⟩`)...(`⟨P_n⟩`)`

   See figure 2 where a regular pentagon is drawn; usage of polar coordinates is also shown.

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

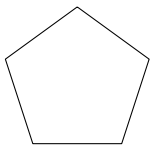

Figure 2: Polygonal line obtained by means of the `\polyline` command; vertex coordinates are in polar form.

   Examples of using polar and cartesian coordinates are shown in figure 3.

   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.

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

```
\begin{picture}(40,30)
\put(0,0){\GraphGrid(40,30)}
\put(40,0){\circle*{1.5}}
   \put(41,0){\makebox(0,0)[bl]{40,0}}
\put(90:30){\circle*{1.5}}
   \put(90:31){\makebox(0,0)[bl]{90:30}}
\put(60:30){\circle*{1.5}}
   \put(60:31){\makebox(0,0)[bl]{60:30}}
\put(30,30){\circle*{1.5}}
   \put(30.7,30.7){\makebox(0,0)[bl]{30,30}}
\multiput(0,0)(30:10){5}%
   {\makebox(0,0){\rule{1.5mm}{1.5mm}}}
\end{picture}
```

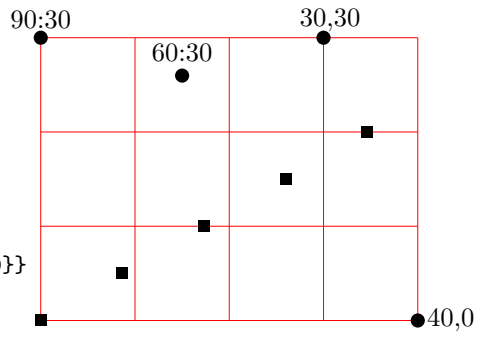

Figure 3: Use of cartesian and polar coordinates

```
\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 polar form.

`\Dashline(`⟨*first point*⟩`)(`⟨*second point*⟩`){`⟨*dash length*⟩`}`

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.

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

   `\Dotline(`⟨*first point*⟩`)(`⟨*end point*⟩`){`⟨*dot gap*⟩`}`

   See figures 5 and 7 for examples.

9. `\GraphGrid` is a command that draws a red grid under the drawing with lines separated `10\unitlength`s 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

```
\unitlength=1mm
\begin{picture}(40,40)
\put(0,0){\GraphGrid(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}
 \put(40,40){\circle*{2}}
\end{picture}
```

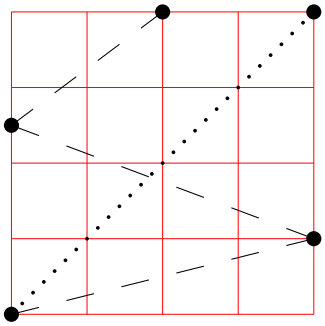

Figure 5: Dashed lines and graph grid

```
\begin{picture}(40,30)
\put(0,0){\GraphGrid(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}
```

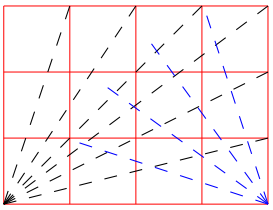

Figure 6: Different length dashed lines with the same nominal dash length

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. 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.

10. New trigonometric function macros have been computed by means of the functionalities of the `xfp` package. The compared to the other existing macros is that angles are specified in sexagesimal degrees, so that the user needs not transform to radians. The computations are done taking into account that abnormal values can occasionally be avoided, for example $\tan 90°$ 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;

```
\begin{picture}(40,30)
\put(0,0){\GraphGrid(40,30)}
\Dotline(0,0)(40,10){1.5}\Dotline(0,0)(40,20){1.5}
\Dotline(0,0)(40,30){1.5}\Dotline(0,0)(30,30){1.5}
\Dotline(0,0)(20,30){1.5}\Dotline(0,0)(10,30){1.5}
{\color{red}\Dotline*(40,0)(108:30){1.5}
\Dotline*(40,0)(126:30){1.5}
\Dotline*(40,0)(144:30){1.5}
\Dotline*(40,0)(162:30){1.5}}%
\end{picture}
```

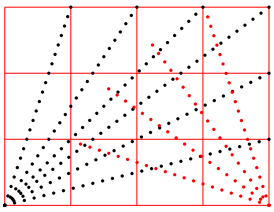

Figure 7: Different length dotted lines with the same nominal dot gap

```
\unitlength=0.5mm
\begin{picture}(60,40)
\put(0,0){\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)[l]{$60^\circ$}}
\VectorARC(60,20)(60,0){-180}
\end{picture}
```

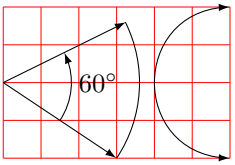

Figure 8: Arcs and curved vectors

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*⟩

The ⟨*control sequence*⟩ may then be used as a multiplying factor of a length.

11. 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*⟩`}`

    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` draws an arc with arrow tips at both ends; see figure 8.

12. A multitude of commands have been defined in order to manage complex numbers; actually complex numbers are represented as a comma separated pair of fractional numbers (here we use only cartesian coordinates). 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 ⟨*vector*⟩ means a comma separated pair of fractional numbers, ⟨*vector macro*⟩ means a macro that contains a comma separated pair of fractional numbers; ⟨*angle macro*⟩ means a macro that contains the angle of a vector in sexagesimal degrees; ⟨*argument*⟩ means a brace delimited numeric value, even a macro; *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 generally directly used in the Euler formula $\vec{v} = Me^{j\varphi}$.

- `\MakeVectorFrom`⟨*two arguments*⟩`to`⟨*vector macro*⟩
- `\CopyVect`⟨*first vector*⟩`to`⟨*second vector macro*⟩
- `\ModOfVect`⟨*vector*⟩`to`⟨*macro*⟩
- `\DirOfvect`⟨*vector*⟩`to`⟨*versor macro*⟩
- `\ModAndDirOfVect`⟨*vector*⟩`to`⟨*1st macro*⟩`and`⟨*2nd macro*⟩
- `\DistanceAndDirOfVect`⟨*1st vector*⟩`minus`⟨*2nd vector*⟩`to`⟨*1st macro*⟩`and`⟨*2nd macro*⟩
- `\XpartOfVect`⟨*vector*⟩`to`⟨*macro*⟩
- `\YpartOfVect`⟨*vector*⟩`to`⟨*macro*⟩
- `\DirFromAngle`⟨*angle*⟩`to`⟨*versor macro*⟩
- `\ArgOfVect`⟨*vector*⟩`to`⟨*angle macro*⟩
- `\ScaleVect`⟨*vector*⟩`by`⟨*scaling factor*⟩`to`⟨*vector macro*⟩
- `\ConjVect`⟨*vector*⟩`to`⟨*conjugate vector macro*⟩
- `\SubVect`⟨*first vector*⟩`from`⟨*second vector*⟩`to`⟨*vector macro*⟩
- `\AddVect`⟨*first vector*⟩`and`⟨*second vector*⟩`to`⟨*vector macro*⟩
- `\MultVect`⟨*first vector*⟩`by`⟨*second vector*⟩`to`⟨*vector macro*⟩
- `\MultVect`⟨*first vector*⟩`by*`⟨*second vector*⟩`to`⟨*vector macro*⟩
- `\DivVect`⟨*first vector*⟩`by`⟨*second vector*⟩`to`⟨*vector macro*⟩

13. 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` to draw 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 fist 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.

- `\Qurve` similar to `\Curve`, but with second order (quadratic) Bézier splines. The starred version fills the interior with the current color
- `\CurveBetween` draws a single cubic Bézier spline between two given nodes and with two given directions vectors.
- `\CBezierBetween` draws a single cubic Bézier spline between two given nodes, with two given directions 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.

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

(⟨*node coordinates*⟩)<⟨*direction vector*⟩>

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

...(⟨...⟩)<⟨...⟩>[⟨*new direction vector*⟩](⟨...⟩)<⟨...⟩>...

Possibly it is necessary to specifiy 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 form 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 syntax[1]:

...(⟨...⟩)<⟨*direction vector*;*start tension*,*end tension*⟩>(⟨...⟩)<⟨...⟩>...

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 stating 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

[1]The 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.

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 9 for an example.

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

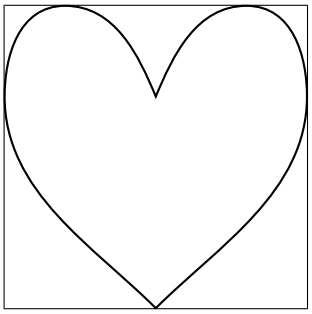

Figure 9: A heart shaped curve with cusps drawn with `\Curve`

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

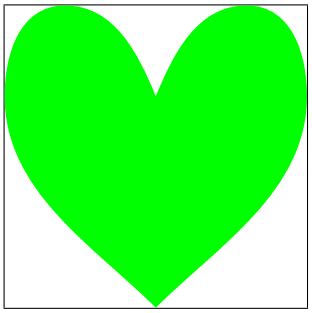

Figure 10: Coloring the inside of a closed path drawn with `\Curve*`

With the starred version of `\Curve`, instead of stroking the contour, the macro fills up the contour with the selected current color, figure 10.

Figure 11 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 containa many of the commands described so far.

To show what you can do with `\CurveBetween` see the code and result shown in figure 12. Notice the effect of changing the directions at both or a the end nodes os a single cubic spline. The directions are conveniently expressed with unit vectors described by polar coordinates.

A little more complicated is the use of the `\CBezierBetween` macro, figure 13. 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

```
\unitlength=0.007\textwidth
\begin{picture}(100,90)(-50,-50)
\put(-50,0){\vector(1,0){100}}\put(50,1){\makebox(0,0)[br]{$x$}}%
\put(20,-1){\makebox(0,0)[t]{$s$}}%
\put(0,0){\circle*{2}}\put(-1,-1){\makebox(0,0)[tr]{$M$}}%
\legenda(12,-45){s=\overline{MP_2}=R\sin\theta}%
\put(0,-50){\vector(0,1){90}}%
\put(1,40){\makebox(0,0)[tl]{$y$}}%
\put(0,-40){\circle*{2}}\put(1,-41){\makebox(0,0)[lt]{$C$}}%
\segment(0,-40)(-40,0)\segment(0,-40)(40,0)%
\put(-41,1){\makebox(0,0)[br]{$P_1$}}\put(-40,0){\circle*{2}}%
\put(41,1){\makebox(0,0)[bl]{$P_2$}}\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]{$C_1$}}%
\segment(40,0)(20,20)\put(20,20){\circle*{2}}%
\put(20,21.5){\makebox(0,0)[b]{$C_2$}}%
\put(0,-40){\put(0,56.5685){\circle*{2}}%
\put(1,58){\makebox(0,0)[bl]{$P$}}}%
\VectorARC(0,-40)(15,-25){45}\put(10,-18){\makebox(0,0)[c]{$\theta$}}%
\VectorARC(40,0)(20,0){-45}\put(19,5){\makebox(0,0)[r]{$\theta$}}%
\VectorARC(-40,0)(-20,0){45}\put(-19,5){\makebox(0,0)[l]{$\theta$}}%
\put(-20,-18){\makebox(0,0)[bl]{$R$}}%
\put(-32,13){\makebox(0,0)[bl]{$K$}}%
\put(32,13){\makebox(0,0)[br]{$K$}}%
\end{picture}
```

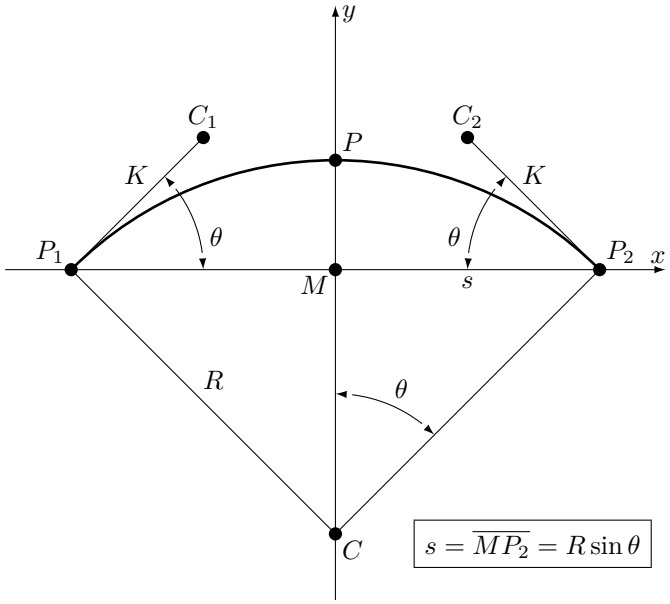

Figure 11: The code to display the Nodes and control points for an arc to be approximated with a cubic Bézier spline

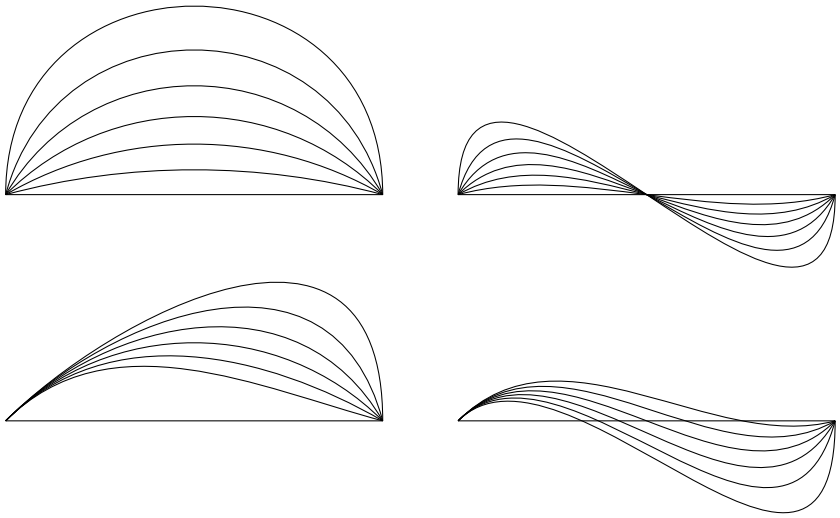

Figure 12: Curves between two points with different start and end slopes

```
\unitlength=0.1\textwidth
\begin{picture}(10,3)
\CurveBetween0,0and10,0WithDirs1,1and{1,-1}
\color{red}%
\CbezierBetween0,0And10,0 WithDirs45:1And-45:1UsingDists4And{1}
\CbezierBetween0,0And10,0 WithDirs45:1And-45:1UsingDists6And{1}
\CbezierBetween0,0And10,0 WithDirs45:1And-45:1UsingDists8And{1}
\CbezierBetween0,0And10,0 WithDirs45:1And-45:1UsingDists10And{1}
\CbezierBetween0,0And10,0 WithDirs45:1And-45:1UsingDists12And{1}
\end{picture}
```

Figure 13: Comparison between similar arcs drawn with `\CurveBetween` (black) and `\CbezierTo` (red)

left one increases from 4 to 10. The black line corresponds to the standard `\CurveBetween` where the default distance is computed by default to trace an arc of a circle and is approximately 3.5.

In figure 14 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 wit the horizontal base line of the frame. Increasing the tension 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 and increasing that tension value is just to obtain special effects.

Figure 15 displays two approximations of a sine wave; Bézier splines can ap-

```
\raggedleft\unitlength=0.01\textwidth
\begin{picture}(70,70)
\put(0,0){\color{blue}\frame(70,70){}}
\put(0,0){\color{red}\Curve(0,0)<1,1>(70,0)<1,-1>}
\Curve(0,0)<1,1>(70,0)<1,-1;0,0>
\Curve(0,0)<1,1>(70,0)<1,-1;0.2,0.2>
\Curve(0,0)<1,1>(70,0)<1,-1;2,2>
\Curve(0,0)<1,1>(70,0)<1,-1;4.5,4.5>
\Curve(0,0)<1,1>(70,0)<1,-1;0,3>
\Curve(0,0)<1,1>(70,0)<1,-1;3,0>
\end{picture}
```

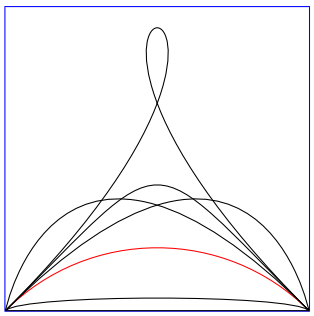

Figure 14: The effects of tension factors

proximate transcendental curves, but the approximation may be a poor one, depending on the approximated curve, if 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 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.

In figure 16 some lines drawn with quadratic splines by means of the `\Qurve` macro are shown. 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 color the outside of the green four-pointed star. The `curve2e-v161`, 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.

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 LaTeX 2_ε was first released in 1994; in the `latexnews` news-letter of December 2003; the first implementation was announced; the first version of this package 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 extending the drawing facilities; but Till Tantau's `pgf` package has gone much farther.

```
\unitlength=0.01\textwidth
\begin{picture}(100,50)(0,-25)
\put(0,0){\VECTOR(0,0)(45,0)\VECTOR(0,-25)(0,25)
\Zbox(45,0)[br]{x}\Zbox(0,26)[tl]{y}
\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)
\Zbox(45,0)[br]{x}\Zbox(0,26)[tl]{y}
\CbezierBetween0,0And20,0WithDirs77:1And-77:1UsingDists28And{28}
\CbezierBetween20,0And40,0WithDirs-77:1And77:1UsingDists28And{28}}
\end{picture}
```

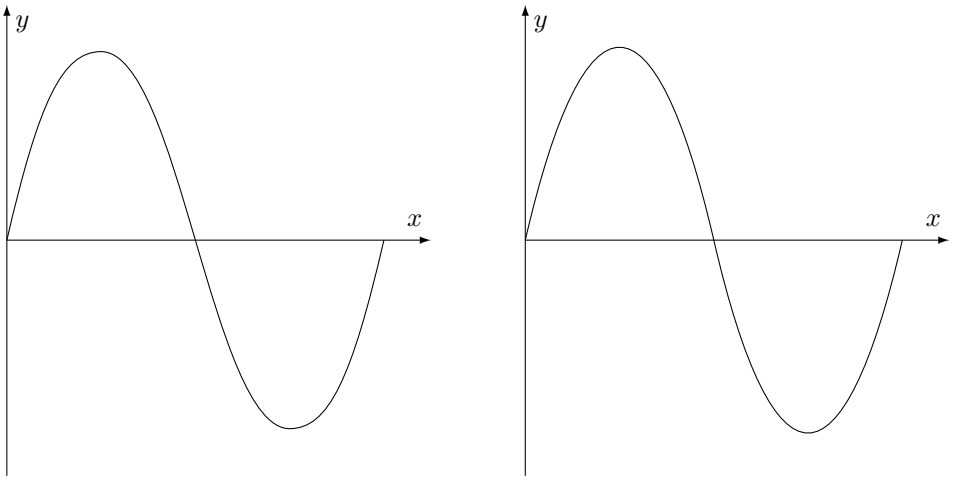

Figure 15: A sequence of arcs; the left figure has been drawn with the `\Curve` command with a sequence of four couples of point-direction arguments; the right figure has been drawn with two commands `\CbezierBetween` that include also the specification of the control points

4 Remark

There are other packages in the CTAN archives that deal with tracing curves of various kinds. `PSTricks` and `tikz/pgf` are the most powerful ones. But there is also the package `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 option to the package call 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 but since it explains very clearly what exactly are the Bézier splines. Apparently `ebezier` should be used only for dvi output

```
%\unitlength=0.0045\textwidth
%\begin{picture}(100,100)
%\put(0,0){\framebox(100,100){}}
%\put(50,50){%
%  \Qurve(0,-50)<1,0>(50,0)<0,1>(0,50)<-1,0>(-50,0)<0,-1>(0,-50)<1,0>
%\color{green}
%  \Qurve*(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>
%}
%\Qurve(0,0)<1,4>(50,50)<1,0>(100,100)<1,4>
%\put(5,50){\Qurve(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}
%\Pall[2](0,0)\Pall[2](100,0)\Pall[2](100,100)\Pall[2](0,100)
%\end{picture}
%\hfill
%\begin{picture}(100,100)
%\put(0,0){\framebox(100,100){}}
%\put(50,50){%
%\Qurve(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}\Pall[2](50,50)\put(50,50){\Vector(45:50)}\Zbox(67,70)[tl]{R}
%\end{picture}
```

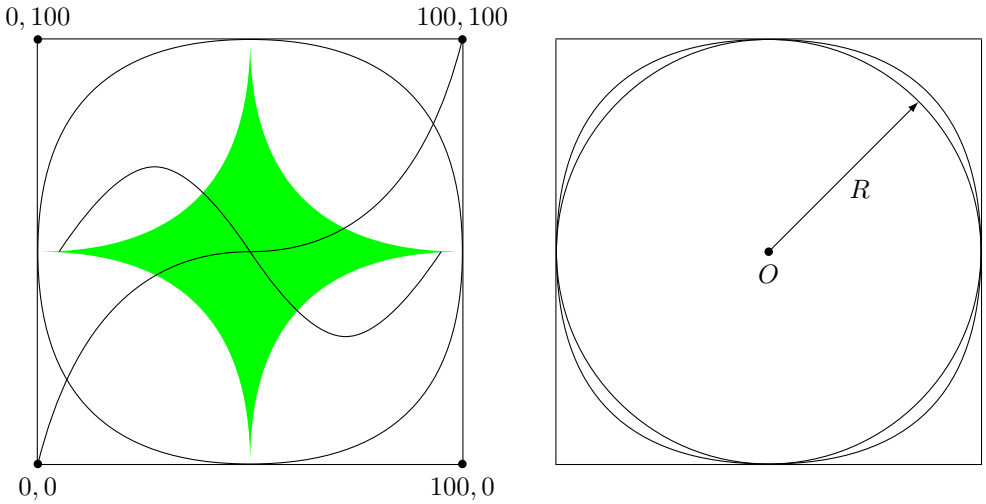

Figure 16: Several graphs drawn with quadratic Bézier splines. On the right a quadratic spline circle is compared with a cubic line circle.

without recourse to PostScript 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 is 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 very comfortable; 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, even extended with `pict2e`, and, why not, with `curve2e`.

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.

Josef Tkadlec and the author collaborated extensively in order to make 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 we abandoned the long division macro, and substituted it with the floating point division provided by the `xfp` package.

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

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 version 0.2x or later, dated 2009/08/05 or later, of `pict2e` is being used, such modification is not necessary, but it's true that it becomes imperative if older versions are used.

References

[1] Gäßlein H., Niepraschk R., and Tkadlec J. *The `pict2e` package*, 2014, PDF documentation of `pict2e`; this package is part of any modern complete distribution of the TeX system. In case of a basic or partial system installation, the package may be installed by means of the specific facilities of the distribution. It may be read by means of the line command `texdoc pict2e`.