The extension package curve2e

Claudio Beccari*

Version v.2.2.6 – Last revised 2020-04-02.

Contents

Abstract

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

Please take notice that on 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 some 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.

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

1 Introduction

Package `pict2e` was announced in issue 15 of `latexnews` around December 2003; it was specified that the new package would replace the dummy one that has been accompanying every release of LaTeX 2$_\varepsilon$ 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 (but see below); 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 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 must pay attention to the meaning of cartesian vs. polar coordinates. Both imply a displacement with respect the actual origin of the axes. So when a circle 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 become the stating 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.

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 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.

3. Commands for setting the line terminations were introduced; the user can chose between square or round caps; the default is set to round caps; now this feature is directly available with `pict2e`.

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

5. 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`, and then introduced directly in `pict2e`.

6. 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 `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 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 line specification.

8. 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 cartesian or polar form; the end point in cartesian format specifies the desired end point; while 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.

9. 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.

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 is 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. The above `\Curve` macro is recursive and it 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 the documentation `curve2e-manual.pdf` file. 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 each Bézier-spline 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 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 any case they are redefined so as to accept symbolic names for the point coordinates in both the cartesian and polar form.

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, the `xfp` has recently become available, by which any sort of calculations can be done with 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 to implement them and to make other calculations.

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.

2 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 versions 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 our 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 `pict2e`, version 0.2x or later, dated 2009/08/05 or later, is being used, such modification is not necessary any more, but it's true that it became imperative when older versions were used.

Some others users spotted other "features" that did not produce the desired results; they have been acknowledged by footnotes in correspondence with the corrections that were made tanks their contribution.

3 Source code

3.1 Some preliminary extensions to the `pict2e` package

The necessary preliminary code has already been introduced. Here we require the `color` package and the `pict2e` one; for the latter one we make sure that a sufficiently recent version is used. If you want to use package `xcolor`, load it *after* `curve2e`.

Here we load also the `xparse` and `xfp` packages because we use their functionalities; but we do load them only if they are not already loaded with or without options; nevertheless we warn the user who wants to load them explicitly, to do this action before loading `curve2e`. The `xfp` package is absolutely required; if this package is not found in the TEX system installation, the loading of this new `curve2e` is aborted, and the previous version 1.61 is loaded in its place; the overall functionalities should non change much, but the functionalities of `xfp` are not available.

```
\IfFileExists{xfp.sty}{%
  \RequirePackage{color}
  \RequirePackageWithOptions{pict2e}[2014/01/01]
  \@ifl@aded{sty}{xparse}{}{\RequirePackage{xparse}}
  \@ifl@aded{sty}{xfp}{}{\RequirePackage{xfp}}%
}{%
    \RequirePackage{curve2e-v161}%
    \PackageWarningNoLine{curve2e}{%
      Package xfp is required, but apparently\MessageBreak%
      such package cannot be found in this \MessageBreak%
      TeX system installation\MessageBreak%
      Either your installation is not complete \MessageBreak%
      or it is older than 2018-10-17.\MessageBreak%
      \MessageBreak%
      ************************************\MessageBreak%
      Version 1.61 of curve2e has been loaded\MessageBreak%
      instead of the current version\MessageBreak%
      ************************************\MessageBreak}%
      \endinput
}
```

Since we already loaded package`xfp` or at least we explicitly load it in our preamble, we add, if not already defined by the package, the two new commands that allow to make floating point tests, and to implement a "while" cycle[1]

```
%
\ExplSyntaxOn
\AtBeginDocument{%
\ProvideExpandableDocumentCommand\fptest{m m m}{%
   \fp_compare:nTF{#1}{#2}{#3}}
\ProvideExpandableDocumentCommand\fpdowhile{m m}{%
   \fp_do_while:nn{#1}{#2}}
```

[1]Thanks to Brian Dunn who spotted a bug in the previous 2.0.x version definitions.

```
}
\ExplSyntaxOff

```

The next macros are just for debugging. With the `trace` package it would probably be better to define other macros, but this is not for the users, but for the developers.

```
\def\TRON{\tracingcommands\tw@ \tracingmacros\tw@}%
\def\TROF{\tracingcommands\z@ \tracingmacros\z@}%
```

Next we define some new dimension registers that will be used by the subsequent macros; should they be already defined, there will not be any redefinition; nevertheless the macros should be sufficiently protected so as to avoid overwriting register values loaded by other macro packages.

```
\ifx\undefined\@tdA \newdimen\@tdA \fi
\ifx\undefined\@tdB \newdimen\@tdB \fi
\ifx\undefined\@tdC \newdimen\@tdC \fi
\ifx\undefined\@tdD \newdimen\@tdD \fi
\ifx\undefined\@tdE \newdimen\@tdE \fi
\ifx\undefined\@tdF \newdimen\@tdF \fi
\ifx\undefined\defaultlinewidth \newdimen\defaultlinewidth \fi
```

3.2 Line thickness macros

It is better to define a macro for setting a different value for the line and curve thicknesses; the '`\defaultlinewidth` should contain the equivalent of `\@wholewidth`, that is the thickness of thick lines; thin lines are half as thick; so when the default line thickness is specified to, say, 1pt, thick lines will be 1pt thick and thin lines will be 0.5pt thick. The default whole width of thick lines is 0,8pt, but this is specified in the kernel of LaTeX and/or in `pict2e`. On the opposite it is necessary to redefine `\linethickness` because the LaTeX kernel global definition does not hide the space after the closed brace when you enter something such as `\linethickness{1mm}` followed by a space or a new line.[2]

```
\gdef\linethickness#1{%
\@wholewidth#1\@halfwidth.5\@wholewidth\ignorespaces}%
\newcommand\defaultlinethickness[1]{\defaultlinewidth=#1\relax
\def\thicklines{\linethickness{\defaultlinewidth}}%
\def\thinlines{\linethickness{.5\defaultlinewidth}}\thinlines
   \ignorespaces}%
```

The `\ignorespaces` at the end of these macros is for avoiding spurious spaces to get into the picture that is being drawn, because these spaces introduce picture deformities often difficult to spot and to eliminate.

[2] Thanks to Daniele Degiorgi `degiorgi@inf.ethz.ch`). This feature should have been eliminated from the LaTeX 2_ε ¡2020.0202¿ patch level 4 update.

3.3 Improved line and vector macros

The macro `\LIne` allows to draw a line with arbitrary inclination as if it was a polygonal with just two vertices; actually it joins the canvas coordinate origin with the specified relative coordinate; therefore this object must be set in place by means of a `\put` command. Since its starting point is always at a relative 0,0 coordinate point inside the box created with `\put`, the two arguments define the horizontal and the vertical component respectively.

```
\def\LIne(#1){{\GetCoord(#1)\@tX\@tY
  \moveto(0,0)
  \pIIe@lineto{\@tX\unitlength}{\@tY\unitlength}\strokepath}\ignorespaces
}%
```

A similar macro `\segment` operates between two explicit points with absolute coordinates, instead of relative to the position specified by a `\put` command; it resorts to the `\polyline` macro that shall be defined in a while. The `\@killglue` command might be unnecessary, but it does not harm; it eliminates any explicit or implicit spacing that might precede this command.

```
\def\segment(#1)(#2){\@killglue\polyline(#1)(#2)}%
```

By passing its ending points coordinates to the `\polyline` macro, both macro arguments are a pair of coordinates, not their components; in other words, if $P_1 = (x_1, y_2)$ and $P_2 = (x_2, y_2)$, then the first argument is the couple x_1, y_1 and likewise the second argument is x_2, y_2. Notice that since `\polyline` accepts also the vertex coordinates in polar form, also`\segment` accepts the polar form. Please remember that the decimal separator is the decimal *point*, while the *comma* acts as cartesian coordinate separator. This recommendation is particularly important for non-English speaking users, since in all other languages the decimal separator is or must by a comma.

The `\line` macro is redefined by making use of a division routine performed in floating point arithmetics; for this reason the LaTeX kernel and the overall TeX system installation must be as recent as the release date of the `xfp` package, i.e. 2018-10-17. The floating point division macro receives in input two fractional numbers and yields on output their fractional ratio. Notice that this command `\line` should follow the same syntax as the original pre 1994 LaTeX version; but the new definition accepts the direction coefficients in polar mode; that is, instead of specifying a slope of 30° with its actual sine and cosine values (or values proportional to such functions), for example, `(0.5,0.866025)`, you may specify it as `(30:1)`, i.e. as a unit vector with the required slope of 30°.

The beginning of the macro definition is the same as that of `pict2e`:

```
\def\line(#1)#2{\begingroup
  \@linelen #2\unitlength
  \ifdim\@linelen<\z@\@badlinearg\else
```

but as soon as it is verified that the line length is not negative, things change remarkably; in facts the machinery for complex numbers is invoked. This makes the code much simpler, not necessarily more efficient; nevertheless `\DirOfVect` takes the only macro argument (that actually contains a comma separated pair

of fractional numbers) and copies it to `\Dir@line` (an arbitrarily named control sequence) after re-normalizing to unit magnitude; this is passed to `GetCoord` that separates the two components into the control sequences `\d@mX` and `\d@mY`; these in turn are the values that are actually operated upon by the subsequent commands.

```
    \expandafter\DirOfVect#1to\Dir@line
    \GetCoord(\Dir@line)\d@mX\d@mY
```

The normalised vector direction is actually formed with the directing cosines of the line direction; since the line length is actually the horizontal component for non vertical lines, it is necessary to compute the actual line length for non vertical lines by dividing the given length by the magnitude of the horizontal cosine `\d@mX`, and the line length is accordingly scaled:

```
    \ifdim\d@mX\p@=\z@\else
      \edef\sc@lelen{\fpeval{1 / abs(\d@mX)}}\relax
      \@linelen=\sc@lelen\@linelen
    \fi
```

Of course, if the line is vertical this division must not take place. Finally the `moveto`, `lineto` and `stroke` language keywords are invoked by means of the internal `pict2e` commands in order to draw the line. Notice that even vertical lines are drawn with the PDF language commands instead of resorting to the DVI low level language that was used in both `pict2e` and the original (pre 1994) `picture` commands; it had a meaning in the old times, but it certainly does not have any nowadays, since lines are drawn by the driver that produces the output in a human visible document form, not by TEX the program.

```
    \moveto(0,0)\pIIe@lineto{\d@mX\@linelen}{\d@mY\@linelen}%
    \strokepath
  \fi
\endgroup\ignorespaces}%
```

The new definition of the command `\line`, besides the ease with which is readable, does not do different things from the definition of `pict2e` 2009, even if it did perform in a better way compared to the 2004 version that was limited to integer direction coefficients up to 999 in magnitude. Moreover this `curve2e` version accepts polar coordinates as slope pairs, making it much simpler to draw lines with specific slopes.

It is necessary to redefine the low level macros `\moveto`, `\lineto`, and `\curveto`, because their original definitions accept only cartesian coordinates. We proceed the same as for the `\put` command.

```
\let\originalmoveto\moveto
\let\originallineto\lineto
\let\originalcurveto\curveto

\def\moveto(#1){\GetCoord(#1)\MTx\MTy
  \originalmoveto(\MTx,\MTy)\ignorespaces}
\def\lineto(#1){\GetCoord(#1)\LTx\LTy
  \originallineto(\LTx,\LTy)\ignorespaces}
\def\curveto(#1)(#2)(#3){\GetCoord(#1)\CTpx\CTpy
  \GetCoord(#2)\CTsx\CTsy\GetCoord(#3)\CTx\CTy
```

```
\originalcurveto(\CTpx,\CTpy)(\CTsx,\CTsy)(\CTx,\CTy)\ignorespaces}
```

3.4 Dashed and dotted lines

Dashed and dotted lines are very useful in technical drawings; here we introduce two macros that help drawing them in the proper way; besides the obvious difference between the use of dashes or dots, they may refer in a different way to the end points that must be specified to the various macros.

The coordinates of the first point P_1, where le line starts, are always referred to the origin of the coordinate axes; the end point P_2 coordinates are referred to the origin of the axes if in cartesian form, while with the polar form they are referred to P_1; both coordinate types have their usefulness: see the documentation `curve2e-manual.pdf` file.

The above mentioned macros create dashed lines between two given points, with a dash length that must be specified, or dotted lines, with a dot gap that must be specified; actually the specified dash length or dot gap is a desired one; the actual length or gap is computed by integer division between the distance of the given points and the desired dash length or dot gap; when dashes are involved, this integer is tested in order to see if it is an odd number; if it's not, it is increased by unity. Then the actual dash length or dot gap is obtained by dividing the above distance by this number.

Another vector $P_2 - P_1$ is created by dividing it by this number; then, when dashes are involved, it is multiplied by two in order to have the increment from one dash to the next; finally the number of patterns is obtained by integer division of this number by 2 and increasing it by 1. Since the whole dashed or dotted line is put in position by an internal `\put` command, is is not necessary to enclose the definitions within groups, because they remain internal to the `\put` argument box.

Figure 6 of the `curve2e-manual.pdf` user manual shows the effect of the slight changing of the dash length in order to maintain *approximately* the same dash-space pattern along the line, irrespective of the line length. The syntax is the following:

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

where ⟨*first point*⟩ contains the coordinates of the starting point and ⟨*second point*⟩ the absolute (cartesian) or relative (polar) coordinates of the ending point; of course the ⟨*dash length*⟩, which equals the dash gap, is mandatory. An optional asterisk is used to be back compatible with previous implementations but its use is now superfluous; with the previous implementation of the code, in facts, if coordinates were specified in polar form, without the optional asterisk the dashed line was misplaced, while if the asterisk was specified, the whole object was put in the proper position. With this new implementation, both the cartesian and polar coordinates always play the role they are supposed to play independently from the asterisk. The `\IsPolar` macro is introduced to analyse the coordinate type used for the second argument, and uses such second argument accordingly.

```
\def\IsPolar#1:#2?{\def\@TempOne{#2}\unless\ifx\@TempOne\empty
   \expandafter\@firstoftwo\else
```

```
   \expandafter\@secondoftwo\fi}

\ifx\Dashline\undefined
  \def\Dashline{\@ifstar{\Dashline@}{\Dashline@}}% bckwd compatibility
  \let\Dline\Dashline

  \def\Dashline@(#1)(#2)#3{\put(#1){%
   \GetCoord(#1)\@tA\@tB \MakeVectorFrom\@tA\@tB to\V@ttA
   \GetCoord(#2)\@tA\@tB \MakeVectorFrom\@tA\@tB to\V@ttB
   \IsPolar#2:?{%                          Polar
      \Dashline@@(0,0)(\V@ttB){#3}}%
   {%                                      Cartesian
     \SubVect\V@ttA from\V@ttB to\V@ttC
     \Dashline@@(0,0)(\V@ttC){#3}%
   }
}}

  \def\Dashline@@(#1)(#2)#3{%
   \countdef\NumA3254\countdef\NumB3252\relax
   \GetCoord(#1)\@tA\@tB \MakeVectorFrom\@tA\@tB to\V@ttA
   \GetCoord(#2)\@tA\@tB \MakeVectorFrom\@tA\@tB to\V@ttB
   \SubVect\V@ttA from\V@ttB to\V@ttC
   \ModOfVect\V@ttC to\DlineMod
   \DivideFN\DlineMod by#3 to\NumD
   \NumA=\fpeval{trunc(\NumD,0)}\relax
   \unless\ifodd\NumA\advance\NumA\@ne\fi
   \NumB=\NumA \divide\NumB\tw@
   \DividE\DlineMod\p@ by\NumA\p@ to\D@shMod
   \DividE\p@ by\NumA\p@ to \@tempa
   \Multvect{\V@ttC}{\@tempa,0}\V@ttB
   \Multvect{\V@ttB}{2,0}\V@ttC
   \advance\NumB\@ne
   \put(\V@ttA){\multiput(0,0)(\V@ttC){\NumB}{\LIne(\V@ttB)}}
   \ignorespaces}
\fi
```

A simpler `\Dotline` macro draws a dotted line between two given points; the dots are rather small, therefore the inter dot distance is computed in such a way as to have the first and the last dot at the exact position of the dotted-line end-points; again the specified dot distance is nominal in the sense that it is recalculated in such a way that the first and last dots coincide with the line end points. Again if the second point coordinates are in polar form they are considered as relative to the first point.Since the dots must emerge from the background of the drawing they should not be too small: they must be seen; therefore their diameter cannot be tied to the unit length of the particular drawing, but must have at visible size; by default it is set to 0.5 mm (about 20 mills, in US units) but through an optional argument to the macro, it may be set to any desired size; remember that 1 pt is about one third of a millimeter; sometimes it might be too small; 1 mm is a very black dot, therefore the user must pay attention when s/he specifies the dot

diameter, so as not to exaggerate in either direction. The syntax is as follows:

`\Dotline`(⟨*start point*⟩)(⟨*end point*⟩){⟨*dot distance*⟩}[⟨*diameter*⟩]

```
\ifx\Dotline\undefined
  \providecommand\Dotline{}
  \RenewDocumentCommand\Dotline{R(){0,0} R(){1,0} m O{1mm}}{%
  \put(#1){\edef\Diam{\fpeval{{#4}/\unitlength}}%
  \IsPolar#2:?{\CopyVect#2to\DirDot}%
              {\SubVect#1from#2to\DirDot}%
  \countdef\NumA=3254\relax
  \ModAndAngleOfVect\DirDot to\ModDirDot and\AngDirDot
  \edef\NumA{\fpeval{trunc(\ModDirDot/{#3},0)}}%
  \edef\ModDirDot{\fpeval{\ModDirDot/\NumA}}%
  \multiput(0,0)(\AngDirDot:\ModDirDot){\inteval{\NumA+1}}%
    {\makebox(0,0){\circle*{\Diam}}}}\ignorespaces}
\fi
```

Notice that vectors as complex numbers in their cartesian and polar forms always represent a point position referred to a local origin of the axes; this is why in figures 6 and 7 of the user manual the dashed and dotted lines that start from the lower right corner of the graph grid, and that use polar coordinates, are put in their correct position thanks to the different behaviour obtained with the `\IsPolar` macro.

3.5 Coordinate handling

The new macro `\GetCoord` splits a vector (or complex number) specification into its components; in particular it distinguishes the polar from the cartesian form of the coordinates. The latter have the usual syntax ⟨*x,y*⟩, while the former have the syntax ⟨*angle:radius*⟩. The `\put` and `\multiput` commands are redefined to accept the same syntax; the whole work is done by `\SplitNod@` and its subsidiaries.

Notice that package `eso-pic` uses `picture` macros in its definitions, but its original macro `\LenToUnit` is incompatible with this `\GetCoord` macro; its function is to translate real lengths into coefficients to be used as multipliers of the current `\unitlength`; in case that the `eso-pic` had been loaded, at the `\begin{document}` execution, the `eso-pic` macro is redefined using the e-TEX commands so as to make it compatible with these local macros.[3]

```
\AtBeginDocument{\@ifpackageloaded{eso-pic}{%
\renewcommand\LenToUnit[1]{\strip@pt\dimexpr#1*\p@/\unitlength}}{}}%
```

The above redefinition is delayed at `\AtBeginDocument` in order to have the possibility to check if the `eso-pic` package had actually been loaded. Nevertheless the code is defined here just because the original `eso-pic` macro was interfering with the algorithms of coordinate handling.

But let us come to the real subject of this section. We define a `\GettCoord` macro that passes control to the service macro with the expanded arguments;

[3] Thanks to Franz-Joseph Berthold who was so kind to spot the bug.

expanding arguments allows to use macros to named points, instead of explicit coordinates; with this version of `curve2e` this facility is not fully exploited, but a creative user can use this feature. Notice the usual trick to use a dummy macro that is defined within a group with expanded arguments, but where the group is closed by the macro itself, so that no traces remain behind after its expansion.

```
\def\GetCoord(#1)#2#3{\bgroup\edef\x{\egroup\noexpand\IsPolar#1:?}\x
{% Polar
  \bgroup\edef\x{\egroup\noexpand\SplitPolar(#1)}\x\SCt@X\SCt@Y}%
{% Cartesian
  \bgroup\edef\x{\egroup\noexpand\SplitCartesian(#1)}\x\SCt@X\SCt@Y}%
  \edef#2{\SCt@X}\edef#3{\SCt@Y}\ignorespaces}

\def\SplitPolar(#1:#2)#3#4{%
  \edef#3{\fpeval{#2 * cosd#1}}\edef#4{\fpeval{#2 * sind#1}}}

\def\SplitCartesian(#1,#2)#3#4{\edef#3{#1}\edef#4{#2}}

```

The macro that detects the form of the coordinates is `\IsPolar`; it examines the parameter syntax in order to see if it contains a colon; it has already been used with the definition of dashed and dotted lines.

In order to accept polar coordinates with `\put` and `\multiput` we resort to using `\GetCoord`; therefore the redefinition of `\put` is very simple because it suffices to save the original meaning of that macro and redefine the new one in terms of the old one.

```
\let\originalput\put
\def\put(#1){\bgroup\GetCoord(#1)\@tX\@tY
\edef\x{\noexpand\egroup\noexpand\originalput(\@tX,\@tY)}\x}
```

For `\multiput` it is more complicated, because the increments from one position to the next cannot be done efficiently because the increments in the original definition are executed within boxes, therefore any macro instruction inside these boxes is lost. It is a good occasion to modify the `\multiput` definition by means of the advanced macro definitions provided by package `xparse`; we can add also some error messages for avoiding doing anything when some mandatory parameters are missing ore are empty, or do not contain anything different from an ordered pair or a polar form. We add also an optional argument to handle the increments outside the boxes. The new macro has the following syntax:

`\multiput[`⟨*shift*⟩`](`⟨*initial*⟩`)(`⟨*increment*⟩`){`⟨*number*⟩`}{`⟨*object*⟩`}[`⟨*handler*⟩`]`

where the optional ⟨*shift*⟩ is used to displace to whole set of ⟨*object*⟩s from their original position; ⟨*initial*⟩ contains the cartesian or polar coordinates of the initial point; ⟨*increment*⟩ contains the cartesian or polar increment for the coordinates to be used from the second position to the last; ⟨*number*⟩ il the total number of ⟨*object*⟩s to be drawn; ⟨*object*⟩ is the object to be put in position at each cycle repetition; the optional ⟨*handler*⟩ may be used to control the current values of the horizontal and vertical increments. The new definition contains two `\put` commands where the second is nested within a while-loop which, in turn, is within

the argument of the first `\put` command. Basically it is the same idea that the original macros, but now the increments are computed within the while loop, but outside the argument of the inner `\put` command. If the optional ⟨*handler*⟩ is specified the increments are computed from the macros specified by the user.

The two increments components inside the optional argument may be set by means of mathematical expressions operated upon by the `\fpeval` function given by the `\xfp` package already loaded by `curve2e`. Of course it is the user responsibility to pay attention to the scales of the two axes and to write meaningful expressions; the figure and code shown in the user manual of this package display some examples: see the documentation `curve2e-manual.pdf` file.

```
\RenewDocumentCommand{\multiput}{O{0,0} d() d() m m o }{%
  \IfNoValueTF{#2}{\PackageError{curve2e}%
      {\string\multiput\space initial point coordinates missing}%
      {Nothing done}
    }%
    {\IfNoValueTF{#3}{\PackageError{curve2e}
      {\string\multiput\space Increment components missing}%
      {Nothing done}
    }%
    {\put(#1){\let\c@multicnt\@multicnt
      \CopyVect #2 to \R
      \CopyVect#3 to\D
        \@multicnt=#4\relax
        \@whilenum \@multicnt > \z@\do{%
          \put(\R){#5}%
          \IfValueTF{#6}{#6}{\AddVect#3 and\R to \R}%
          \advance\@multicnt\m@ne
        }%
      }%
    }%
  }\ignorespaces
}
```

And here it is the new `\xmultiput` command; remember: the internal cycling TEX counter `\@multicnt` is now accessible with the name `multicnt` as if it was a LATEX counter, in particular the user can access its contents with a command such as `\value{multicnt}`. Such counter is *stepped up* at each cycle, instead of being *stepped down* as in the original `\multiput` command. The code is not so different from the one used for the new version of `\multiput`, but it appears more efficient and its code more easily readable.

```
\NewDocumentCommand{\xmultiput}{O{0,0} d() d() m m o }{%
\IfNoValueTF{#2}{\PackageError{curve2e}{%
  \string\Xmultiput\space initial point coordinates missing}%
  {Nothing done}}%
  {\IfNoValueTF{#3}{\PackageError{curve2e}{%
    \string\Xmultiput\space Increment components missing}%
    {Nothing done}}%
  {\put(#1)%
  {\let\c@multicnt\@multicnt
```

```
  \CopyVect #2 to \R
  \CopyVect #3 to \D
    \@multicnt=\@ne
      \fpdowhile{\value{multicnt} < \inteval{#4+1}}% Test
         {%
         \put(\R){#5}
         \IfValueTF{#6}{#6}{%
           \AddVect#3 and\R to \R}
           \advance\@multicnt\@ne
      }
    }
  }}\ignorespaces
}
```

Notice that the internal macros `\R` and `\D`, (respectively the current point coordinates, in form of a complex number, where to put the ⟨*object*⟩, and the current displacement to find the next point) are accessible to the user both in the ⟨*object*⟩ argument field and the ⟨*handler*⟩ argument field. The code used in figure 18 if the user manual shows how to create the hour marks of a clock together with the rotated hour roman numerals.

3.6 Vectors

The redefinitions and the new definitions for vectors are a little more complicated than with segments, because each vector is drawn as a filled contour; the original `pict2e` 2004 macro checked if the slopes are corresponding to the limitations specified by Lamport (integer three digit signed numbers) and sets up a transformation in order to make it possible to draw each vector as an horizontal left-to-right arrow and then to rotate it by its angle about its tail point; with `pict2e` 2009, possibly this redefinition of `\vector` is not necessary, but we do it as well and for the same reasons we had for redefining `\line`; actually there are two macros for tracing the contours that are eventually filled by the principal macro; each contour macro draws the vector with a LaTeX or a PostScript styled arrow tip whose parameters are specified by default or may be taken from the parameters taken from the `PSTricks` package if this one is loaded before `pict2e`; in any case we did not change the contour drawing macros because if they are modified the same modification is passed on to the arrows drawn with the `curve2e` package redefinitions.

Because of these features the new macros are different from those used for straight lines.

We start with the redefinition of `\vector` and we use the machinery for vectors (as complex numbers) we used for `\line`.

```
  \def\vector(#1)#2{%
    \begingroup
      \GetCoord(#1)\d@mX\d@mY
      \@linelen#2\unitlength
```

As in `pict2e` we avoid tracing vectors if the slope parameters are both zero.

```
      \ifdim\d@mX\p@=\z@\ifdim\d@mY\p@=\z@\@badlinearg\fi\fi
```

But we check only for the positive nature of the l_x component; if it is negative, we simply change sign instead of blocking the typesetting process. This is useful also for macros `\Vector`, `\VECTOR`, and `\VVECTOR` to be defined in a while.

```
      \ifdim\@linelen<\z@ \@linelen=-\@linelen\fi
```

We now make a vector with the given slope coefficients even if one or the other is zero and we determine its direction; the real and imaginary parts of the direction vector are also the values we need for the subsequent rotation.

```
      \MakeVectorFrom\d@mX\d@mY to\@Vect
      \DirOfVect\@Vect to\Dir@Vect
```

In order to be compatible with the original `pict2e` we need to transform the components of the vector direction into lengths with the specific names `\@xdim` and `\@ydim`

```
        \YpartOfVect\Dir@Vect to\@ynum \@ydim=\@ynum\p@
        \XpartOfVect\Dir@Vect to\@xnum \@xdim=\@xnum\p@
```

If the vector is really sloping we need to scale the l_x component in order to get the vector total length; we have to divide by the cosine of the vector inclination which is the real part of the vector direction. We use the floating point division function; since it yields a "factor" that we directly use to scale the length of the vector. We finally memorise the true vector length in the internal dimension `@tdB`

```
        \ifdim\d@mX\p@=\z@
        \else\ifdim\d@mY\p@=\z@
          \else
            \edef\sc@lelen{\fpeval{1 / abs(\@xnum)}}\relax
            \@linelen=\sc@lelen\@linelen
          \fi
        \fi
      \@tdB=\@linelen
```

The remaining code is definitely similar to that of `pict2e`; the real difference consists in the fact that the arrow is designed by itself without the stem; but it is placed at the vector end; therefore the first statement is just the transformation matrix used by the output driver to rotate the arrow tip and to displace it the right amount. But in order to draw only the arrow tip we have to set the `\@linelen` length to zero.

```
\pIIe@concat\@xdim\@ydim{-\@ydim}\@xdim{\@xnum\@linelen}{\@ynum\@linelen}%
        \@linelen\z@
        \pIIe@vector
        \fillpath
```

Now we can restore the stem length that must be shortened by the dimension of the arrow; by examining the documentation of `pict2e` we discover that we have to shorten it by an approximate amount of AL (with the notations of `pict2e`, figs 10 and 11); the arrow tip parameters are stored in certain variables with which we can determine the amount of the stem shortening; if the stem was too short and the new length is negative, we avoid designing such a stem.

```
      \@linelen=\@tdB
      \@tdA=\pIIe@FAW\@wholewidth
      \@tdA=\pIIe@FAL\@tdA
      \advance\@linelen-\@tdA
      \ifdim\@linelen>\z@
        \moveto(0,0)
        \pIIe@lineto{\@xnum\@linelen}{\@ynum\@linelen}%
        \strokepath\fi
    \endgroup}
```

We define the macro that does not require the specification of the length or the l_x length component; the way the new `\vector` macro works does not actually require this specification, because TEX can compute the vector length, provided the two direction components are exactly the horizontal and vertical vector components. If the horizontal component is zero, the actual length must be specified as the vertical component. The object defined with `\Vector`, as well as `\vector`, must be put in place by means of a `\put` command.

```
\def\Vector(#1){{%
\GetCoord(#1)\@tX\@tY
\ifdim\@tX\p@=\z@
  \vector(\@tX,\@tY){\@tY}%
\else
  \vector(\@tX,\@tY){\@tX}%
\fi}}
```

On the opposite the next macro specifies a vector by means of the coordinates of its end points; the first point is where the vector starts, and the second point is the arrow tip side. We need the difference of these two coordinates, because it represents the actual vector.

```
\def\VECTOR(#1)(#2){\begingroup
\SubVect#1from#2to\@tempa
\expandafter\put\expandafter(#1){\expandafter\Vector\expandafter(\@tempa)}%
\endgroup\ignorespaces}
```

The double tipped vector is built on the `\VECTOR` macro by simply drawing two vectors from the middle point of the double tipped vector.

```
\def\VVECTOR(#1)(#2){{\SubVect#1from#2to\@tempb
\ScaleVect\@tempb by0.5to\@tempb
\AddVect\@tempb and#1to\@tempb
\VECTOR(\@tempb)(#2)\VECTOR(\@tempb)(#1)}\ignorespaces}
```

The `pict2e` documentation says that if the vector length is zero the macro draws only the arrow tip; this may work with macro `\vector`, certainly not with `\Vector` and `\VECTOR`. This might be useful for adding an arrow tip to a circular arc. See the documentation `curve2e-manual.pdf` file.

3.7 Polylines and polygons

We now define the polygonal line macro; its syntax is very simple:

`\polygonal[`⟨*join*⟩`](`⟨P_0⟩`)(`⟨P_1⟩`)(`⟨P_2⟩`)...(`⟨P_n⟩`)`

Remember: `\polyline` has been incorporated into `pict2e` 2009, but we redefine it so as to allow an optional argument to specify the line join type.

In order to write a recursive macro we need aliases for the parentheses; actually we need only the left parenthesis, but some editors complain about unmatched delimiters, so we define an alias also for the right parenthesis.

```
\let\lp@r( \let\rp@r)
```

The first call to `\polyline`, besides setting the line joins, examines the first point coordinates and moves the drawing position to this point; afterwards it looks for the second point coordinates; they start with a left parenthesis; if this is found the coordinates should be there, but if the left parenthesis is missing (possibly preceded by spaces that are ignored by the `\@ifnextchar` macro) then a warning message is output together with the line number where the missing parenthesis causes the warning: beware, this line number might point to several lines further on along the source file! In any case it's necessary to insert a `\@killglue`command, because `\polyline` refers to absolute coordinates, and not necessarily is put in position through a `\put` command that provides to eliminate any spurious spaces preceding this command.

```
%  \unitlength=0.07\hsize
%  \begin{picture}(8,8)(-4,-4)\color{red}
%  \polygon*(45:4)(135:4)(-135:4)(-45:4)
%  \end{picture}
%
```

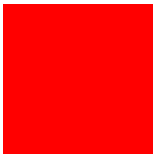

Figure 1: The code and the result of defining a polygon with its vertex polar coordinates

In order to allow a specification for the joints of the various segments of a polyline it is necessary to allow for an optional parameter; the default is the bevel join.

```
\renewcommand*\polyline[1][\beveljoin]{\p@lylin@[#1]}

\def\p@lylin@[#1](#2){\@killglue#1\GetCoord(#2)\d@mX\d@mY
    \pIIe@moveto{\d@mX\unitlength}{\d@mY\unitlength}%
    \@ifnextchar\lp@r{\p@lyline}{%
    \PackageWarning{curve2e}%
    {Polylines require at least two vertices!\MessageBreak
     Control your polyline specification\MessageBreak}%
    \ignorespaces}}

```

But if there is a second or further point coordinate, the recursive macro `\p@lyline` is called; it works on the next point and checks for a further point; if such a point exists the macro calls itself, otherwise it terminates the polygonal line by stroking it.

```
\def\p@lyline(#1){\GetCoord(#1)\d@mX\d@mY
    \pIIe@lineto{\d@mX\unitlength}{\d@mY\unitlength}%
    \@ifnextchar\lp@r{\p@lyline}{\strokepath\ignorespaces}}
```

The same treatment must be done for the `\polygon` macros; we use the defining commands of package `xparse`, in order to use an optional asterisk; as it is usual with `picture` convex lines, the command with asterisk does not trace the contour, but fills the contour with the current color.The asterisk is tested at the beginning and, depending on its presence, a temporary switch is set to `true`; this being the case the contour is filled, otherwise it is simply stroked.

```
\providecommand\polygon{}
\RenewDocumentCommand\polygon{s O{\beveljoin} }{\@killglue\begingroup
\IfBooleanTF{#1}{\@tempswatrue}{\@tempswafalse}%
\@polygon[#2]}

\def\@polygon[#1](#2){\@killglue#1\GetCoord(#2)\d@mX\d@mY
    \pIIe@moveto{\d@mX\unitlength}{\d@mY\unitlength}%
    \@ifnextchar\lp@r{\@@polygon}{%
    \PackageWarning{curve2e}%
    {Polygons require at least two vertices!\MessageBreak
     Control your polygon specification\MessageBreak}%
    \ignorespaces}}

  \def\@@polygon(#1){\GetCoord(#1)\d@mX\d@mY
    \pIIe@lineto{\d@mX\unitlength}{\d@mY\unitlength}%
    \@ifnextchar\lp@r{\@@polygon}{\pIIe@closepath
      \if@tempswa\pIIe@fillGraph\else\pIIe@strokeGraph\fi
      \endgroup
      \ignorespaces}}
```

Now, for example, a filled polygon can be drawn using polar coordinates for its vertices; see figure 1 on page 19.

Remember; the polygon polar coordinates are relative to the origin of the local axes; therefore in order to put a polygon in a different position, it is necessary to do it through a `\put` command.

3.8 The red service grid

The next command is handy for debugging while editing one's drawing; it draws a red grid with square meshes that are ten drawing units apart; there is no graduation along the grid, since it is supposed to be a debugging aid and the user should know what s/he is doing; nevertheless it is advisable to displace the grid by means of a `\put` command so that its grid lines coincide with graph coordinates that are multiples of 10. Missing to do so the readings become cumbersome. The `\RoundUp` macro provides to increase the grid dimensions to integer multiples of ten. Actually the new definition of this command does not need a `put` command (although it is not prohibited to use it) because its syntax now is the following one

`\GraphGrid(`⟨*ll corner offset*⟩`)(`⟨*grid dimensions*⟩`)`

where the first argument is optional: if it is missing, the lower left corner is put at the origin of the canvas coordinates. Of course also the lower left corner offset is recommended to be specified with coordinates that are integer multiples of 10; this is particularly important when the `picture` environment offset is specified with non integer multiple of 10 values. Actually, since both arguments are delimited with round parentheses, a single argument is assumed to contain the grid dimensions, while if both arguments are given, the first one is the lower left corner offset, and the second one the grid dimensions.

In order to render the red grid a little more automatic, a subsidiary service macro of the `picture` environment has been redefined in order to store the coordinates of the lower left and upper right corners of the canvas (the compulsory dimensions and the optional lower left corner shift arguments to the `picture` opening statement) in two new variables, so that when the user specifies the (non vanishing) dimensions of the canvas, the necessary data are already available and there is no need to repeat them to draw the grid. The new argument-less macro is named `\AutoGrid`, while the complete macro is `\GraphGrid` that requires its arguments as specified above. The advantage of the availability of both commands, consists in the fact that `\AutoGrid` covers the whole canvas, while `\GraphGrid` may compose a grid that covers either the whole canvas or just a part of it. In both cases, though, it is necessary the all the canvas coordinates are specified as multiples of 10 (`\unitlength`s). This is simple when `\GraphGrid` is used, while with `\AutoGrid` the specification is in the opening environment statement; and such multiples of 10 might not be the best ones for the final drawing and should be fine tuned after finishing the drawing and the grid is not necessary anymore. The actual `\AutoGrid` command definition accepts two parenthesis delimited arguments, that are not being used in the macro expansion; in this way it is easier to replace `\GraphGrid` with `\AutoGrid` if it is desired to do so. The opposite action, of course is not so simple if the `\AutoGrid` command is not followed by one or two arguments as `\GraphGrid` requires. Approximately `\AutoGrid` may be viewed as a `\GraphGrid` version were both arguments are optional.

```
\def\@picture(#1,#2)(#3,#4){%
  \edef\pict@urcorner{#1,#2}% New statement
  \edef\pict@llcorner{#3,#4}% New statement
  \@picht#2\unitlength
  \setbox\@picbox\hb@xt@#1\unitlength\bgroup
    \hskip -#3\unitlength
    \lower #4\unitlength\hbox\bgroup
      \ignorespaces}
%
\def\Gr@phGrid(#1,#2){\bgroup\textcolor{red}{\linethickness{.1\p@}%
\RoundUp#1modulo10to\@GridWd \RoundUp#2modulo10to\@GridHt
\@tempcnta=\@GridWd \divide\@tempcnta10\relax \advance\@tempcnta\@ne
\multiput(0,0)(10,0){\@tempcnta}{\line(0,1){\@GridHt}}%
\@tempcnta=\@GridHt \divide\@tempcnta10\advance\@tempcnta\@ne
\multiput(0,0)(0,10){\@tempcnta}{\line(1,0){\@GridWd}}\thinlines}%
\egroup\ignorespaces}

```

```
\NewDocumentCommand\AutoGrid{d() d()}{\bgroup%
\put(\pict@llcorner){\expandafter\Gr@phGrid\expandafter(\pict@urcorner)}%
\egroup\ignorespaces}


\NewDocumentCommand\GraphGrid{r() d()}{%
\IfValueTF{#2}{\put(#1){\Gr@phGrid(#2)}}%
              {\put(0,0){\Gr@phGrid(#1)}}}

```

Rounding up is useful because also the grid margins fall on coordinates multiples of 10.

```
\def\RoundUp#1modulo#2to#3{\edef#3{\fpeval{(ceil(#1/#2,0))*#2}}}%
%
```

The `\Integer` macro takes a possibly fractional number whose decimal separator, if present, *must* be the decimal point and uses the point as an argument delimiter. If one has the doubt that the number being passed to `\Integer` might be an integer, he/she should call the macro with a further point; if the argument is truly integer this point works as the delimiter of the integer part; if the argument being passed is fractional this extra point gets discarded as well as the fractional part of the number. This maro used to be used within the definition of `\RoundUp`; with the `xfp` facilities the latter macro does not need it any more, but it continues to be used in several other macros.

```
\def\Integer#1.#2??{#1}%
```

4 Math operations on fractional operands

This is not the place to complain about the fact that all programs of the TEX system use only integer arithmetics; now, with the 2018 distribution of the modern TEX system, package `xfp` is available: this package resorts in the background to language LATEX 3; with this language now it is possible to compute fractional number operations; the numbers are coded in decimal, not in binary, and it is possible also to use numbers written as in computer science, that is as a fractional, possibly signed, number followed by an expression that contains the exponent of 10 necessary to (ideally) move the fractional separator in one or the other direction according to the sign of the exponent of 10; in other words the L3 library for floating point calculations accepts such expressions as `123.456`, `0.12345e3`, and `12345e-3`, and any other equivalent expression. If the first number is integer, it assumes that the decimal separator is to the right of the rightmost digit of the numerical string.

Floating point calculations may be done through the `\fpeval` L3 function with a very simple syntax:

`\fpeval`{⟨*mathematical expression*⟩}

where ⟨*mathematical expression*⟩ can contain the usual algebraic operation sings, `'+' - * / ** ^` and the function names of the most common algebraic, trigono-

metric, and transcendental functions; for direct and inverse trigonometric functions it accepts arguments in radians and in sexagesimal degrees; it accepts the group of rounding/truncating operators; it can perform several kinds of comparisons; as to now (Nov. 2019) the todo list includes the direct and inverse hyperbolic functions. The mantissa length of the floating point operands amounts to 16 decimal digits. Further details may be read in the documentations of the `xfp` and `interface3` packages, just by typing into a command line window the command `texdoc` ⟨*document*⟩, where ⟨*document*⟩ is just the name of the above named files without extension.

Furthermore we added a couple of interface macros with the internal L3 floating point functions; `\fptest` and `\fpdowhile`. They have the following syntax:

`\fptest`{⟨*logical expression*⟩}{⟨*true code*⟩}{⟨*false code*⟩}
`\fpdowhile`{⟨*logical expression*⟩}{⟨*code*⟩}

The ⟨*logical expression*⟩ compares the values of any kind by means of the usual `>`, `=`, and `<` operators that may be negated with the "not' operator `!`; furthermore the logical results of these comparisons may be acted upon with the "and" operator `&&` and the "or" operator . The ⟨*true code*⟩, and ⟨*code*⟩ are executed if or while the ⟨*logical expression*⟩ is true, while the ⟨*false code*⟩ is executed if the ⟨*logical expression*⟩ is false

Before the availability of the `xfp` package, it was necessary to fake fractional number computations by means of the native e-TEX commands `\dimexpr`, i.e. to multiply each fractional number by the unit `\p@` (1 pt) so as to get a length; operate on such lengths, and then stripping off the 'pt' component from the result; very error prone and with less precision as the one that the modern decimal floating point calculations can do. Of course it is not so important to use fractional numbers with more that 5 or 6 fractional digits, because the other TEX and LATEX macros cannot handle them, but it is very convenient to have simpler and more readable code. We therefore switched to the new floating point functionality, even if this maintains the `curve2e` functionality, but renders this package unusable with older LATEX kernel installations. It has already been explained that the input of this up-to-date version of `curve2e` is aborted if the `xfp` package is not available, but the previous version 1.61 version is loaded in its place; very little functionality is lost, but, evidently, this new version performs in a better way.

4.1 The division macro

The most important macro is the division of two fractional numbers; we seek a macro that gets dividend and divisor as fractional numbers and saves their ratio in a macro; this is done in a simple way with the following code.

```
294 \def\DividE#1by#2to#3{\edef#3{\fpeval{#1 / #2}}}
```

In order to avoid problems with divisions by zero, or with numbers that yield results too large to be used as multipliers of lengths, it would be preferable that the above code be preceded or followed by some tests and possible messages. Actually we decided to avoid such tests and messages, because the internal L3

functions already provide some. This was done in the previous versions of this package, when the `\fpeval` L3 function was not available.

Notice that operands `#1` and `#2` may be integer numbers or fractional, or mixed numbers. They may be also dimensions, but while dimensions in printer points (72.27pt=1in) are handled as assumed, when different units are used, the length must be enclosed in parentheses:

```
%\DividE(1mm)by(3mm) to\result
%
```

yields correctly `\result=0.33333333`. Without parentheses the result is unpredictable.

For backwards compatibility we need an alias.

```
\let\DivideFN\DividE
```

We do the same in order to multiply two integer o fractional numbers held in the first two arguments and the third argument is a definable token that will hold the result of multiplication in the form of a fractional number, possibly with a non null fractional part; a null fractional part is stripped away

```
\def\MultiplY#1by#2to#3{\edef#3{\fpeval{#1 * #2}}}\relax
\let\MultiplyFN\MultiplY
```

but with multiplication it is better to avoid computations with lengths.

The next macro uses the `\fpeval` macro to get the numerical value of a measure in points. One has to call `\Numero` with a control sequence and a dimension, with the following syntax; the dimension value in points is assigned to the control sequence.

`\Numero`⟨*control sequence*⟩⟨*dimension*⟩

```
\unless\ifdefined\Numero
  \def\Numero#1#2{\edef#1{\fpeval{round(#2,6)}}\ignorespaces}%
\fi
```

The numerical value is rounded to 6 fractional digits that are more than sufficient for the graphical actions performed by `curve2e`.

The `\ifdefined` primitive command is provided by the e-TEX extension of the typesetting engine; the test does not create any hash table entry; it is a different way than the `\ifx\csname...\endcsname` test, because the latter first possibly creates a macro meaning `\relax` then executes the test; therefore an undefined macro name is always defined to mean `\relax`.

4.2 Trigonometric functions

We now start with trigonometric functions. In previous versions of this package we defined the macros `\SinOf`, `\CosOf` and `\TanOf` (`\CotOf` did not appear so essential) by means of the parametric formulas that require the knowledge of the tangent of the half angle. We wanted, and still want, to specify the angles

in sexagesimal degrees, not in radians, so that accurate reductions to the main quadrants are possible. The bisection formulas are

$$\begin{aligned}\sin\theta &= \frac{2}{\cot x+\tan x}\\ \cos\theta &= \frac{\cot x-\tan x}{\cot x+\tan x}\\ \tan\theta &= \frac{2}{\cot x-\tan x}\end{aligned}$$

where

$$x = \theta/114.591559$$

is the half angle in degrees converted to radians.

But now, in this new version, the availability of the floating point computations with the specific L3 library makes all the above superfluous; actually the above approach gave good results but it was cumbersome and limited by the fixed radix computations of the TEX system programs.

Matter of facts, we compared the results (with 6 fractional digits) the computations executed with the `sind` function name, in order to use the angles in degrees, and a table of trigonometric functions with the same number of fractional digits, and we did not find any difference, not even one unit on the sixth decimal digit. Probably the `\fpeval` computations, without rounding before the sixteenth significant digit, are much more accurate, but it is useless to have a better accuracy when the other TEX and LATEX macros would not be able to exploit them.

Having available such powerful instrument, even the tangent appears to be of little use for the kind of computations that are supposed to be required in this package.

The codes for the computation of `\SinOf` and `\CosOf` of the angle in degrees is now therefore the following

```
\def\SinOf#1to#2{\edef#2{\fpeval{round(sind#1,6)}}}\relax
\def\CosOf#1to#2{\edef#2{\fpeval{round(cosd#1,6)}}}\relax
```

Sometimes tie argument of a complex number is necessary; therefore with macro `\ArgOfVect` we calculate the four quadrant arctangent (in degrees) of the given vector taking into account the sings of the vector components. We use the `xfp atand` with two arguments, so that it automatically takes into account all the signs for determining the argument of vector x, y by giving the values x and y in the proper order to the function `atan`:

$$\text{if}\quad x+\mathrm{i}y = M\mathrm{e}^{\mathrm{i}\varphi}\quad\text{then}\quad \varphi = \texttt{\textbackslash fpeval}\{\texttt{atand}(y,x)\}$$

The `\ArgOfVect` macro receives on input a vector and determines its four quadrant argument; it only checks if both vector components are zero, because in this case nothing is done, and the argument is assigned the value zero.

```
\def\ArgOfVect#1to#2{\GetCoord(#1){\t@X}{\t@Y}%
\fptest{\t@X=\z@ && \t@Y=\z@}{\edef#2{0}%
\PackageWarning{curve2e}{Null vector}{Check your data\MessageBreak
Computations go on, but the results may be meaningless}}{%
\edef#2{\fpeval{round(atand(\t@Y,\t@X),6)}}}\ignorespaces}
```

Since the argument of a null vector is meaningless, we set it to zero in case that input data refer to such a null vector. Computations go on anyway, but the results may be meaningless; such strange results are an indications that some controls on the code should be done by the user.

It is worth examining the following table, where the angles of nine vectors 45° degrees apart from one another are computed from this macro.

Vector	0,0	1,0	1,1	0,1	-1,1	-1,0	-1,-1	0,-1	1,-1
Angle	0	0	45	90	135	180	-135	-90	-45

Real computations with the `\ArgOfVect` macro produce those very numbers without the need of rounding; `\fpeval` produces all trimming of lagging zeros and rounding by itself.

4.3 Arcs and curves preliminary information

We would like to define now a macro for drawing circular arcs of any radius and any angular aperture; the macro should require the arc center, the arc starting point and the angular aperture. The arc has its reference point in its center, therefore it does not need to be put in place by the command `\put`; nevertheless if `\put` is used, it may displace the arc into another position.

The command should have the following syntax:

`\Arc(`⟨*center*⟩`)(`⟨*starting point*⟩`){`⟨*angle*⟩`}`

which is totally equivalent to:

`\put(`⟨*center*⟩`){\Arc`(0,0)(⟨*starting point*⟩)`{`⟨*angle*⟩`}}`

If the ⟨*angle*⟩, i.e. the arc angular aperture, is positive the arc runs counterclockwise from the starting point; clockwise if it is negative. Notice that since the ⟨*starting point*⟩ is relative to the ⟨*center*⟩ point, its polar coordinates are very convenient, since they become `(`⟨⟨*start angle*⟩`:`⟨*radius*⟩⟩`)`, where the ⟨*start angle*⟩ is relative to the arc center. Therefore you can think about a syntax such as this one:

`\Arc(`⟨⟨*center*⟩⟩`)(`⟨*start angle*`:`*radius*⟩`){`⟨*angle*⟩`}`

The difference between the `pict2e` `\arc` definition consists in a very different syntax:

`\arc[`⟨*start angle*⟩`,`⟨*end angle*⟩`]{`⟨*radius*⟩`}`

and the center is assumed to be at the coordinate established with a required `\put` command; moreover the difference in specifying angles is that ⟨*end angle*⟩ equals the sum of ⟨*start angle*⟩ and ⟨*angle*⟩. With the definition of this `curve2e` package use of a `\put` command is not prohibited, but it may be used for fine tuning the arc position by means of a simple displacement; moreover the ⟨*starting point*⟩ may be specified with polar coordinates (that are relative to the arc center).

It's necessary to determine the end point and the control points of the Bézier spline(s) that make up the circular arc.

The end point is obtained from the rotation of the starting point around the center; but the `pict2e` command `\pIIe@rotate` is such that the pivoting point appears to be non relocatable. It is therefore necessary to resort to low level TEX commands and the defined trigonometric functions and a set of macros that operate on complex numbers used as vector roto-amplification operators.

4.4 Complex number macros

In this package *complex number* is a vague phrase; it may be used in the mathematical sense of an ordered pair of real numbers; it can be viewed as a vector joining the origin of the coordinate axes to the coordinates indicated by the ordered pair; it can be interpreted as a roto-amplification operator that scales its operand and rotates it about a pivot point; besides the usual conventional representation used by the mathematicians where the ordered pair is enclosed in round parentheses (which is in perfect agreement with the standard code used by the `picture` environment) there is the other conventional representation used by the engineers that stresses the roto-amplification nature of a complex number:

$$(x, y) = x + \mathrm{j}y = M\mathrm{e}^{\mathrm{j}\theta}$$

Even the imaginary unit is indicated with i by the mathematicians and with j by the engineers. In spite of these differences, such objects, the *complex numbers*, are used without any problem by both mathematicians and engineers.

The important point is that these objects can be summed, subtracted, multiplied, divided, raised to any power (integer, fractional, positive or negative), be the argument of transcendental functions according to rules that are agreed upon by everybody. We do not need all these properties, but we need some and we must create the suitable macros for doing some of these operations.

In facts we need macros for summing, subtracting, multiplying, dividing complex numbers, for determining their directions (unit vectors or versors); a unit vector is the complex number divided by its magnitude so that the result is the cartesian or polar form of the Euler's formula

$$\mathrm{e}^{\mathrm{j}\phi} = \cos\phi + \mathrm{j}\sin\phi$$

The magnitude of a vector is determined by taking the positive square root of the sum of the squared real and the imaginary parts (often called *Pitagorean sum*); see further on.

It's better to represent each complex number with one control sequence; this implies frequent assembling and disassembling the pair of real numbers that make up a complex number. These real components are assembled into the defining control sequence as a couple of coordinates, i.e. two comma separated integer or fractional signed decimal numbers.

For assembling two real numbers into a complex number we use the following elementary macro:

```
308 \def\MakeVectorFrom#1#2to#3{\edef#3{#1,#2}\ignorespaces}%
```

Another elementary macro copies a complex number into another one:

```
\def\CopyVect#1to#2{\edef#2{#1}\ignorespaces}%
```

The magnitude is determined with the macro `\ModOfVect` with delimited arguments; as usual it is assumed that the results are retrieved by means of control sequences, not used directly.

In the preceding version of package `curve2e` the magnitude M was determined by taking the moduli of the real and imaginary parts, by changing their signs if necessary; the larger component was then taken as the reference one, so that, if a is larger than b, the square root of the sum of their squares is computed as such:

$$M = \sqrt{a^2 + b^2} = |a|\sqrt{1 + (b/a)^2}$$

In this way the radicand never exceeds 2 and it was quite easy to get its square root by means of the Newton iterative process; due to the quadratic convergence, five iterations were more than sufficient. When one of the components was zero, the Newton iterative process was skipped.

With the availability of the `xfp` package and its floating point algorithms it is much easier to compute the magnitude of a complex number; since these algorithms allow to use very large numbers, it is not necessary to normalise the complex number components to the largest one; therefore the code is much simpler than the one used for implementing the Newton method in the previous versions of this package.

```
\def\ModOfVect#1to#2{\GetCoord(#1)\t@X\t@Y
\edef#2{\fpeval{round(sqrt(\t@X*\t@X + \t@Y*\t@Y),6)}}%
\ignorespaces}%
```

Since the macro for determining the magnitude of a vector is available, we can now normalise the vector to its magnitude, therefore getting the Cartesian form of the direction vector. If by any chance the direction of the null vector is requested, the output is again the null vector, without normalisation.

```
\def\DirOfVect#1to#2{\GetCoord(#1)\t@X\t@Y
\ModOfVect#1to\@tempa
\fptest{\@tempa=\z@}{}{%
  \edef\t@X{\fpeval{round(\t@X/\@tempa,6)}}%
  \edef\t@Y{\fpeval{round(\t@Y/\@tempa,6)}}%
}\MakeVectorFrom\t@X\t@Y to#2\ignorespaces}%
```

A cumulative macro uses the above ones to determine with one call both the magnitude and the direction of a complex number. The first argument is the input complex number, the second its magnitude, and the third is again a complex number normalised to unit magnitude (unless the input was the null complex number); remember always that output quantities must be specified with control sequences to be used at a later time.

```
\def\ModAndDirOfVect#1to#2and#3{%
\ModOfVect#1to#2%
\DirOfVect#1to#3\ignorespaces}%
```

The next macro computes the magnitude and the direction of the difference of two complex numbers; the first input argument is the minuend, the second is the

subtrahend; the output quantities are the third argument containing the magnitude of the difference and the fourth is the direction of the difference. The service macro `\SubVect` executes the difference of two complex numbers and is described further on.

```
\def\DistanceAndDirOfVect#1minus#2to#3and#4{%
\SubVect#2from#1to\@tempa
\ModAndDirOfVect\@tempa to#3and#4\ignorespaces}%
```

We now have two macros intended to fetch just the real or, respectively, the imaginary part of the input complex number.

```
\def\XpartOfVect#1to#2{%
\GetCoord(#1)#2\@tempa\ignorespaces}%
%
\def\YpartOfVect#1to#2{%
\GetCoord(#1)\@tempa#2\ignorespaces}%
```

With the next macro we create a direction vector (second argument) from a given angle (first argument, in degrees).

```
\def\DirFromAngle#1to#2{%
\edef\t@X{\fpeval{round(cosd#1,6)}}%
\edef\t@Y{\fpeval{round(sind#1,6)}}%
\MakeVectorFrom\t@X\t@Y to#2\ignorespaces}%
```

Sometimes it is necessary to scale (multiply) a vector by an arbitrary real factor; this implies scaling both the real and imaginary part of the input given vector.

```
\def\ScaleVect#1by#2to#3{\GetCoord(#1)\t@X\t@Y
\edef\t@X{\fpeval{#2 * \t@X}}%
\edef\t@Y{\fpeval{#2 * \t@Y}}%
\MakeVectorFrom\t@X\t@Y to#3\ignorespaces}%
```

Again, sometimes it is necessary to reverse the direction of rotation; this implies changing the sign of the imaginary part of a given complex number; this operation produces the complex conjugate of the given number.

```
\def\ConjVect#1to#2{\GetCoord(#1)\t@X\t@Y
\edef\t@Y{-\t@Y}%
\MakeVectorFrom\t@X\t@Y to#2\ignorespaces}%
```

With all the low level elementary operations we can now proceed to the definitions of the binary operations on complex numbers. We start with the addition:

```
\def\AddVect#1and#2to#3{\GetCoord(#1)\tu@X\tu@Y
\GetCoord(#2)\td@X\td@Y
\edef\t@X{\fpeval{\tu@X + \td@X}}%
\edef\t@Y{\fpeval{\tu@Y + \td@Y}}%
\MakeVectorFrom\t@X\t@Y to#3\ignorespaces}%
```

Then the subtraction:

```
\def\SubVect#1from#2to#3{\GetCoord(#1)\tu@X\tu@Y
\GetCoord(#2)\td@X\td@Y
\edef\t@X{\fpeval{\td@X - \tu@X}}%
\edef\t@Y{\fpeval{\td@Y - \tu@Y}}%
\MakeVectorFrom\t@X\t@Y to#3\ignorespaces}%
```

For the multiplication we need to split the operation according to the fact that we want to multiply by the second operand or by the complex conjugate of the second operand; it would be nice if we could use the usual postfixed asterisk notation for the complex conjugate, but in the previous versions of this package we could not find a simple means for doing so. Therefore the previous version contained a definition of the `\MultVect` macro that followed a simple syntax with an optional asterisk *prefixed* to the second operand. Its syntax, therefore, allowed the following two forms:

`\MultVect`⟨*first factor*⟩ `by` ⟨*second factor*⟩ `to` ⟨*output macro*⟩
`\MultVect`⟨*first factor*⟩ `by` ⋆ ⟨*second factor*⟩ `to` ⟨*output macro*⟩

With the availability of the `xparse` package and its special argument descriptors for the arguments, we were able to define a different macro, `\Multvect`, with both optional positions for the asterisk: *after* and *before*; its syntax allows the following four forms:

`\Multvect`{⟨*first factor*⟩}{⟨*second factor*⟩}⟨*output macro*⟩ `\Multvect`{⟨*first factor*⟩}⋆{⟨*second factor*⟩}⟨*output macro*⟩ `\Multvect`{⟨*first factor*⟩}{⟨*second factor*⟩}⋆⟨*output macro*⟩ `\Multvect`{⟨*first factor*⟩}⋆{⟨*second factor*⟩}⟨*output macro*⟩ `\Multvect`{⟨*first factor*⟩}⋆{⟨*second factor*⟩}⋆⟨*output macro*⟩

Nevertheless we maintain a sort of interface between the old syntax and the new one, so that the two old forms can be mapped to two suitable forms of the new syntax. Old documents are still compilable; users who got used to the old syntax can maintain their habits.

First we define the new macro: it receives the three arguments, the first two as balanced texts; the last one must always be a macro, therefore a single (complex) token that does not require braces, even if it is not forbidden to use them. Asterisks are optional. The input arguments are transformed into couples of argument and modulus; this makes multiplication much simpler as the output modulus is just the product of the input moduli, while the output argument is just the sum of input arguments; eventually it is necessary to transform this polar version of the result into an ordered couple of cartesian values to be assigned to the output macro. In order to maintain the single macros pretty simple we need a couple of service macros and a named counter. We use `\ModOfVect` previously defined, and a new macro `\ModAndAngleOfVect` with the following syntax:

`\ModAndAngleOfVect`⟨*input vector*⟩ `to` ⟨*output modulus*⟩ `and` ⟨*output angle in degrees*⟩

The output quantities are always macros, so they do not need balanced bracing; angles in degrees are always preferred because, in case of necessity, they are easy to reduce to the range $-180° < \alpha \leq +180°$.

```
\def\ModAndAngleOfVect#1to#2and#3{\ModOfVect#1to#2\relax
\ArgOfVect#1to#3\ignorespaces}
```

We name a counter in the upper range accessible with all the modern three typesetting engines, `pdfLaTeX`, `LuaLaTeX` and `XeLaTeX`.

```
\newcount\MV@C
```

This TEX counter definition uses the property of modern typesetting engines that use the ε-TEX extensions, that can define a very large number of counters.

Now comes the real macro[4]:

```
\NewDocumentCommand\Multvect{m s m s m}{%
\MV@C=0
\ModAndAngleOfVect#1to\MV@uM and\MV@uA
\ModAndAngleOfVect#3to\MV@dM and\MV@dA
\IfBooleanT{#2}{\MV@C=1}\relax
\IfBooleanT{#4}{\MV@C=1}\relax
\unless\ifnum\MV@C=0\edef\MV@dA{-\MV@dA}\fi
\edef\MV@rM{\fpeval{round((\MV@uM * \MV@dM),6)}}%
\edef\MV@rA{\fpeval{round((\MV@uA + \MV@dA),6)}}%
\GetCoord(\MV@rA:\MV@rM)\t@X\t@Y
\MakeVectorFrom\t@X\t@Y to#5}
```

The macro to remain backward compatible, reduce to two simple macros that take the input delimited arguments and passes them in braced form to the above general macro:

```
\def\MultVect#1by{\@ifstar{\let\MV@c\@ne\@MultVect#1by}%
  {\let\MV@c\empty\@MultVect#1by}}

\def\@MultVect#1by#2to#3{%
  \unless\ifx\MV@c\empty\Multvect{#1}{#2}*{#3}\else
    \Multvect{#1}{#2}{#3}\fi}
```

Testing of both the new and the old macros shows that they behave as expected, although, using real numbers for trigonometric functions, some small rounding unit on the sixth decimal digit still remains; nothing to worry about with a package used for drawing.

The division of two complex numbers implies scaling down the dividend by the magnitude of the divisor and by rotating the dividend scaled vector by the conjugate versor of the divisor:

$$\frac{\vec{N}}{\vec{D}} = \frac{\vec{N}}{M\vec{u}} = \frac{\vec{N}}{M}\vec{u}^{\star}$$

therefore:

```
\def\DivVect#1by#2to#3{\Divvect{#1}{#2}{#3}}

\NewDocumentCommand\Divvect{ m m m }{%
\ModAndDirOfVect#2to\@Mod and\@Dir
\edef\@Mod{\fpeval{1 / \@Mod}}%
\ConjVect\@Dir to\@Dir
\ScaleVect#1by\@Mod to\@tempa
\Multvect{\@tempa}{\@Dir}#3\ignorespaces}%
```

[4] A warm thank-you to Enrico Gregorio, who kindly attracted my attention on the necessity of braces when using this kind of macro; being used to the syntax with delimited arguments I had taken the bad habit of avoiding braces. Braces are very important, but the syntax of the original TEX language, that did not have available the L3 one, spoiled me with the abuse of delimited arguments.

Macros `\DivVect` and `\Divvect` are almost equivalent; the second is possibly slightly more robust. They match the corresponding macros for multiplying two vectors.

4.5 Arcs and curved vectors

We are now in the position of really doing graphic work.

4.5.1 Arcs

We start with tracing a circular arc of arbitrary center, arbitrary starting point and arbitrary aperture; the first macro checks the aperture; if this is not zero it actually proceeds with the necessary computations, otherwise it does nothing.

```
\def\Arc(#1)(#2)#3{\begingroup
\@tdA=#3\p@
\unless\ifdim\@tdA=\z@
  \@Arc(#1)(#2)%
\fi
\endgroup\ignorespaces}%
```

The aperture is already memorised in `\@tdA`; the `\@Arc` macro receives the center coordinates in the first argument and the coordinates of the starting point in the second argument.

```
\def\@Arc(#1)(#2){%
\ifdim\@tdA>\z@
  \let\Segno+%
\else
  \@tdA=-\@tdA \let\Segno-%
\fi
```

The rotation angle sign is memorised in `\Segno` and `\@tdA` now contains the absolute value of the arc aperture.

If the rotation angle is larger than 360° a message is issued that informs the user that the angle will be reduced modulo 360°; this operation is performed by successive subtractions rather than with modular arithmetics on the assumption that in general one subtraction suffices.

```
\Numero\@gradi\@tdA
\ifdim\@tdA>360\p@
 \PackageWarning{curve2e}{The arc aperture is \@gradi\space degrees
      and gets reduced\MessageBreak%
      to the range 0--360 taking the sign into consideration}%
 \@whiledim\@tdA>360\p@\do{\advance\@tdA-360\p@}%
\fi
```

Now the radius is determined and the drawing point is moved to the starting point.

```
\SubVect#2from#1to\@V \ModOfVect\@V to\@Raggio
\CopyVect#2to\@pPun
\CopyVect#1to\@Cent \GetCoord(\@pPun)\@pPunX\@pPunY
```

From now on it's better to define a new macro that will be used also in the subsequent macros that draw arcs; here we already have the starting point coordinates and the angle to draw the arc, therefore we just call the new macro, stroke the line and exit.

```
\@@Arc\strokepath\ignorespaces}%
```

And the new macro `\@@Arc` starts with moving the drawing point to the first point and does everything needed for drawing the requested arc, except stroking it; we leave the `\strokepath` command to the completion of the calling macro and nobody forbids to use the `\@@Arc` macro for other purposes.

```
\def\@@Arc{\pIIe@moveto{\@pPunX\unitlength}{\@pPunY\unitlength}%
```

If the aperture is larger than 180° it traces a semicircle in the right direction and correspondingly reduces the overall aperture.

```
\ifdim\@tdA>180\p@
  \advance\@tdA-180\p@
  \Numero\@gradi\@tdA
  \SubVect\@pPun from\@Cent to\@V
  \AddVect\@V and\@Cent to\@sPun
  \Multvect{\@V}{0,-1.3333333to}\@V
  \if\Segno-\ScaleVect\@V by-1to\@V\fi
  \AddVect\@pPun and\@V to\@pcPun
  \AddVect\@sPun and\@V to\@scPun
  \GetCoord(\@pcPun)\@pcPunX\@pcPunY
  \GetCoord(\@scPun)\@scPunX\@scPunY
  \GetCoord(\@sPun)\@sPunX\@sPunY
  \pIIe@curveto{\@pcPunX\unitlength}{\@pcPunY\unitlength}%
               {\@scPunX\unitlength}{\@scPunY\unitlength}%
               {\@sPunX\unitlength}{\@sPunY\unitlength}%
  \CopyVect\@sPun to\@pPun
\fi
```

If the remaining aperture is not zero it continues tracing the rest of the arc. Here we need the extrema of the arc and the coordinates of the control points of the Bézier cubic spline that traces the arc. The control points lay on the perpendicular to the vectors that join the arc center to the starting and end points respectively.

With reference to figure 11 of the user manual `curve2e-manual.pdf` file, the points P_1 and P_2 are the arc end-points; C_1 and C_2 are the Bézier-spline control-points; P is the arc mid-point, that should be distant from the center of the arc the same as P_1 and P_2. Choosing a convenient orientation of the arc relative to the coordinate axes, the coordinates of these five points are:

$$\begin{aligned} P_1 &= (-R\sin\theta, 0) \\ P_2 &= (R\sin\theta, 0) \\ C_1 &= (-R\sin\theta + K\cos\theta, K\sin\theta) \\ C_2 &= (R\sin\theta - K\cos\theta, K\sin\theta) \\ P &= (0, R(1-\cos\theta)) \end{aligned}$$

The Bézier cubic spline interpolating the end and mid points is given by the parametric equation:

$$P = P_1(1-t)^3 + C_1 3(1-t)^2 t + C_2 3(1-t)t^2 + P_2 t^3$$

where the mid point is obtained for $t = 0.5$; the four coefficients then become $1/8, 3/8, 3/8, 1/8$ and the only unknown remains K. Solving for K we obtain the formula

$$K = \frac{4}{3}\frac{1-\cos\theta}{\sin\theta}R = \frac{4}{3}\frac{1-\cos\theta}{\sin^2\theta}s \tag{1}$$

where θ is half the arc aperture, R is its radius, and s is half the arc chord.

```
\ifdim\@tdA>\z@
  \DirFromAngle\@gradi to\@Dir \if\Segno-\ConjVect\@Dir to\@Dir \fi
  \SubVect\@Cent from\@pPun to\@V
  \Multvect{\@V}{\@Dir}\@V
  \AddVect\@Cent and\@V to\@sPun
  \@tdA=.5\@tdA \Numero\@gradi\@tdA
  \DirFromAngle\@gradi to\@Phimezzi
  \GetCoord(\@Phimezzi)\@cosphimezzi\@sinphimezzi
  \@tdB=1.3333333\p@ \@tdB=\@Raggio\@tdB
  \@tdC=\p@ \advance\@tdC -\@cosphimezzi\p@ \Numero\@tempa\@tdC
  \@tdB=\@tempa\@tdB
  \DividE\@tdB by\@sinphimezzi\p@ to\@cZ
  \ScaleVect\@Phimezzi by\@cZ to\@Phimezzi
  \ConjVect\@Phimezzi to\@mPhimezzi
  \if\Segno-%
    \let\@tempa\@Phimezzi
    \let\@Phimezzi\@mPhimezzi
    \let\@mPhimezzi\@tempa
  \fi
  \SubVect\@sPun from\@pPun to\@V
  \DirOfVect\@V to\@V
  \Multvect{\@Phimezzi}{\@V}\@Phimezzi
  \AddVect\@sPun and\@Phimezzi to\@scPun
  \ScaleVect\@V by-1to\@V
  \Multvect{\@mPhimezzi}{\@V}\@mPhimezzi
  \AddVect\@pPun and\@mPhimezzi to\@pcPun
  \GetCoord(\@pcPun)\@pcPunX\@pcPunY
  \GetCoord(\@scPun)\@scPunX\@scPunY
  \GetCoord(\@sPun)\@sPunX\@sPunY
  \pIIe@curveto{\@pcPunX\unitlength}{\@pcPunY\unitlength}%
               {\@scPunX\unitlength}{\@scPunY\unitlength}%
               {\@sPunX\unitlength}{\@sPunY\unitlength}%
\fi}
```

4.5.2 Arc vectors

We exploit much of the above definitions for the `\Arc` macro for drawing circular arcs with an arrow at one or both ends; the first macro `\VerctorArc` draws an

arrow at the ending point of the arc; the second macro `\VectorARC` draws arrows at both ends; the arrows have the same shape as those for vectors; actually they are drawn by putting a vector of zero length at the proper arc end(s), therefore they are styled as traditional LaTeX or PostScript arrows according to the specific option to the `pict2e` package.

But the arc drawing done here shortens it so as not to overlap on the arrow tip(s); the only arrow tip (or both tips) are also lightly tilted in order to avoid the impression of a corner where the arc enters the arrow tip.

All these operations require a lot of "playing" with vector directions, but even if the operations are numerous, they do not do anything else but: (a) determining the end point and its direction; (b) determining the arrow length as an angular quantity, i.e. the arc amplitude that must be subtracted from the total arc to be drawn; (c) the direction of the arrow should correspond to the tangent to the arc at the point where the arrow tip is attached; (d) tilting the arrow tip by half its angular amplitude; (e) determining the resulting position and direction of the arrow tip so as to draw a zero length vector; (f) possibly repeating the same procedure for the other end of the arc; (g) shortening the total arc angular amplitude by the amount of the arrow tip(s) already set, and finally (h) drawing the circular arc that joins the starting point to the final arrow or one arrow to the other one.

The calling macros are very similar to the `\Arc` macro initial one:

```
\def\VectorArc(#1)(#2)#3{\begingroup
\@tdA=#3\p@ \ifdim\@tdA=\z@\else
  \@VArc(#1)(#2)%
\fi
\endgroup\ignorespaces}%
%
\def\VectorARC(#1)(#2)#3{\begingroup
\@tdA=#3\p@
\ifdim\@tdA=\z@\else
  \@VARC(#1)(#2)%
\fi
\endgroup\ignorespaces}%
```

The single arrow tiped arc is defined with the following long macro where all the described operations are performed more or less in the described succession; probably the macro requires a little cleaning, but since it works fine we did not try to optimise it for time or number of tokens. The final part of the macro is almost identical to that of the plain arc; the beginning also is quite similar. The central part is dedicated to the positioning of the arrow tip and to the necessary calculations for determining the tip tilt and the reduction of the total arc length; pay attention that the arrow length, stored in `\@tdE` is a real length, while the radius stored in `\@Raggio` is just a multiple of the `\unitlength`, so that the division (that yields a good angular approximation to the arrow length as seen from the center of the arc) must be done with real lengths. The already defined `\@@Arc` macro actually draws the curved vector stem without stroking it.

```
\def\@VArc(#1)(#2){%
```

```
\ifdim\@tdA>\z@
  \let\Segno+%
\else
  \@tdA=-\@tdA \let\Segno-%
\fi \Numero\@gradi\@tdA
\ifdim\@tdA>360\p@
 \PackageWarning{curve2e}{The arc aperture is \@gradi\space degrees
      and gets reduced\MessageBreak%
      to the range 0--360 taking the sign into consideration}%
 \@whiledim\@tdA>360\p@\do{\advance\@tdA-360\p@}%
\fi
\SubVect#1from#2to\@V \ModOfVect\@V to\@Raggio \CopyVect#2to\@pPun
\@tdE=\pIIe@FAW\@wholewidth \@tdE=\pIIe@FAL\@tdE
\DividE\@tdE by \@Raggio\unitlength to\DeltaGradi
\@tdD=\DeltaGradi\p@
\@tdD=57.29578\@tdD \Numero\DeltaGradi\@tdD
\@tdD=\ifx\Segno--\fi\@gradi\p@ \Numero\@tempa\@tdD
\DirFromAngle\@tempa to\@Dir
\Multvect{\@V}{\@Dir}\@sPun
\edef\@tempA{\ifx\Segno-\m@ne\else\@ne\fi}%
\Multvect{\@sPun}{0,\@tempA}\@vPun
\DirOfVect\@vPun to\@Dir
\AddVect\@sPun and #1 to \@sPun
\GetCoord(\@sPun)\@tdX\@tdY
\@tdD\ifx\Segno--\fi\DeltaGradi\p@
\@tdD=.5\@tdD \Numero\DeltaGradi\@tdD
\DirFromAngle\DeltaGradi to\@Dird
\Multvect{\@Dir}*{\@Dird}\@Dir%
\GetCoord(\@Dir)\@xnum\@ynum
\put(\@tdX,\@tdY){\vector(\@xnum,\@ynum){0}}%
\@tdE =\ifx\Segno--\fi\DeltaGradi\p@
\advance\@tdA -\@tdE \Numero\@gradi\@tdA
\CopyVect#1to\@Cent \GetCoord(\@pPun)\@pPunX\@pPunY
\@@Arc
\strokepath\ignorespaces}%
```

The macro for the arc terminated with arrow tips at both ends is again very similar, but it is necessary to repeat the arrow tip positioning also at the starting point. The `\@@Arc` macro draws the curved stem.

```
\def\@VARC(#1)(#2){%
\ifdim\@tdA>\z@
  \let\Segno+%
\else
  \@tdA=-\@tdA \let\Segno-%
\fi \Numero\@gradi\@tdA
\ifdim\@tdA>360\p@
 \PackageWarning{curve2e}{The arc aperture is \@gradi\space degrees
      and gets reduced\MessageBreak%
      to the range 0--360 taking the sign into consideration}%
 \@whiledim\@tdA>360\p@\do{\advance\@tdA-360\p@}%
```

```
\fi
\SubVect#1from#2to\@V \ModOfVect\@V to\@Raggio \CopyVect#2to\@pPun
\@tdE=\pIIe@FAW\@wholewidth \@tdE=0.8\@tdE
\DividE\@tdE by \@Raggio\unitlength to\DeltaGradi
\@tdD=\DeltaGradi\p@ \@tdD=57.29578\@tdD \Numero\DeltaGradi\@tdD
\@tdD=\if\Segno--\fi\@gradi\p@ \Numero\@tempa\@tdD
\DirFromAngle\@tempa to\@Dir
\Multvect{\@V}{\@Dir}\@sPun% corrects the end point
\edef\@tempA{\if\Segno--\fi1}%
\Multvect{\@sPun}{0,\@tempA}\@vPun
\DirOfVect\@vPun to\@Dir
\AddVect\@sPun and #1 to \@sPun
\GetCoord(\@sPun)\@tdX\@tdY
\@tdD\if\Segno--\fi\DeltaGradi\p@
\@tdD=.5\@tdD \Numero\@tempB\@tdD
\DirFromAngle\@tempB to\@Dird
\Multvect{\@Dir}*{\@Dird}\@Dir
\GetCoord(\@Dir)\@xnum\@ynum
\put(\@tdX,\@tdY){\vector(\@xnum,\@ynum){0}}% end point arrow tip
\@tdE =\DeltaGradi\p@
\advance\@tdA -2\@tdE \Numero\@gradi\@tdA
\CopyVect#1to\@Cent \GetCoord(\@pPun)\@pPunX\@pPunY
\SubVect\@Cent from\@pPun to \@V
\edef\@tempa{\if\Segno-\else-\fi\@ne}%
\Multvect{\@V}{0,\@tempa}\@vPun
\@tdE\if\Segno--\fi\DeltaGradi\p@
\Numero\@tempB{0.5\@tdE}%
\DirFromAngle\@tempB to\@Dird
\Multvect{\@vPun}{\@Dird}\@vPun% corrects the starting point
\DirOfVect\@vPun to\@Dir\GetCoord(\@Dir)\@xnum\@ynum
\put(\@pPunX,\@pPunY){\vector(\@xnum,\@ynum){0}}% starting point arrow tip
\edef\@tempa{\if\Segno--\fi\DeltaGradi}%
\DirFromAngle\@tempa to \@Dir
\SubVect\@Cent from\@pPun to\@V
\Multvect{\@V}{\@Dir}\@V
\AddVect\@Cent and\@V to\@pPun
\GetCoord(\@pPun)\@pPunX\@pPunY
\@@Arc
\strokepath\ignorespaces}%
```

It must be understood that the curved vectors, the above circular arcs terminated with an arrow tips at one or both ends, have a nice appearance only if the arc radius is not too small, or, said in a different way, if the arrow tip angular width does not exceed a maximum of a dozen degrees (and this is probably already too much); the tip does not get curved as the arc is, therefore there is not a smooth transition from the curved stem and the straight arrow tip if this one is large in comparison to the arc radius.

4.6 General curves

The most used method to draw curved lines with computer programs is to connect several simple curved lines, general "arcs", one to another generally maintaining the same tangent at the junction. If the direction changes we are dealing with a cusp.

The simple general arcs that are directly implemented in every program that displays typeset documents, are those drawn with the parametric curves called *Bézier splines*; given a sequence of points in the x, y plane, say $P_0, P_1, P_2, p_3, \ldots$ (represented as coordinate pairs, i.e. by complex numbers), the most common Bézier splines are the following ones:

$$\mathcal{B}_1 = P_0(1-t) + P_1 t \tag{2}$$

$$\mathcal{B}_2 = P_0(1-t)^2 + P_1 2(1-t)t + P_2 t^2 \tag{3}$$

$$\mathcal{B}_3 = P_0(1-t)^3 + P_1 3(1-t)^2 t + P_2 3(1-t)t^2 + P_3 t^3 \tag{4}$$

All these splines depend on parameter t; they have the property that for $t = 0$ each line starts at the first point, while for $t = 1$ they reach the last point; in each case the generic point P on each curve takes off with a direction that points to the next point, while it lands on the destination point with a direction coming from the penultimate point; moreover, when t varies from 0 to 1, the curve arc is completely contained within the convex hull formed by the polygon that has the spline points as vertices.

Last but not least first order splines implement just straight lines and they are out of question for what concerns maxima, minima, inflection points and the like. Quadratic splines draw just parabolas, therefore they draw arcs that have the concavity just on one side of the path; therefore no inflection points. Cubic splines are extremely versatile and can draw lines with maxima, minima and inflection points. Virtually a multi-arc curve may be drawn by a set of cubic splines as well as a set of quadratic splines (fonts are a good example: Adobe Type 1 fonts have their contours described by cubic splines, while TrueType fonts have their contours described with quadratic splines; at naked eye it is impossible to notice the difference).

Each program that processes the file to be displayed is capable of drawing first order Bézier splines (segments) and third order Bézier splines, for no other reason, at least, because they have to draw vector fonts whose contours are described by Bézier splines; sometimes they have also the program commands to draw second order Bézier splines, but not always these machine code routines are available to the user for general use. For what concerns `pdftex`, `xetex` and `luatex`, they have the user commands for straight lines and cubic arcs. At least with `pdftex`, quadratic arcs must be simulated with a clever use of third order Bézier splines.

Notice that the LaTeX 2_ε environment `picture` by itself is capable of drawing both cubic and quadratic Bézier splines as single arcs; but it resorts to "poor man" solutions. The `pict2e` package removes all the old limitations and implements the interface macros for sending the driver the necessary drawing information, including the transformation from typographical points (72.27 pt/inch) to PostScript big

points (72 bp/inch). But for what concerns the quadratic spline it resorts to the clever use of a cubic spline.

Therefore here we treat first the drawings that can be made with cubic splines; then we describe the approach to quadratic splines.

4.7 Cubic splines

Now we define a macro for tracing a general, not necessarily circular, arc. This macro resorts to a general triplet of macros with which it is possible to draw almost anything. It traces a single Bézier spline from a first point where the tangent direction is specified to a second point where again it is specified the tangent direction. Actually this is a special (possibly useless) case where the general `\curve` macro of `pict2e` could do the same or a better job. In any case...

```
\def\CurveBetween#1and#2WithDirs#3and#4{%
  \StartCurveAt#1WithDir{#3}\relax
  \CurveTo#2WithDir{#4}\CurveFinish\ignorespaces
}%
```

Actually the above macro is a special case of concatenation of the triplet formed by macros `\StartCurve`, `\CurveTo` and`\CurveFinish`; the second macro can be repeated an arbitrary number of times. In any case the directions specified with the direction arguments the angle between the indicated tangent and the arc chord may give raise to some little problems when they are very close to 90° in absolute value. Some control is exercised on these values, but some tests might fail if the angle derives from other calculations; this is a good place to use polar forms for the direction vectors. The same comments apply also to the more general macro `\Curve`,

The first macro initialises the drawing and the third one strokes it; the real work is done by the second macro. The first macro initialises the drawing but also memorises the starting direction; the second macro traces the current Bézier arc reaching the destination point with the specified direction, but memorises this direction as the one with which to start the next arc. The overall curve is then always smooth because the various Bézier arcs join with continuous tangents. If a cusp is desired it is necessary to change the memorised direction at the end of the arc before the cusp and before the start of the next arc; this is better than stroking the curve before the cusp and then starting another curve, because the curve joining point at the cusp is not stroked with the same command, therefore we get two superimposed curve terminations. To avoid this imperfection, we need another small macro `\ChangeDir` to perform this task.

It is necessary to recall that the direction vectors point to the control points, but they do not define the control points themselves; they are just directions, or, even better, they are simply vectors with the desired direction; the macros themselves provide to the normalisation and memorisation.

The next desirable feature would be to design a macro that accepts optional node directions and computes the missing ones according to a suitable strategy. We can think of many such strategies, but none seems to be generally applicable,

in the sense that one strategy might give good results, say, with sinusoids and another one, say, with cardioids, but neither one is suitable for both cases.

For the moment we refrain from automatic direction computation, but we design the general macro as if directions were optional.

Here we begin with the first initialising macro that receives with the first argument the starting point and with the second argument the direction of the tangent (not necessarily normalised to a unit vector)

```
\def\StartCurveAt#1WithDir#2{%
\begingroup
\GetCoord(#1)\@tempa\@tempb
\CopyVect\@tempa,\@tempb to\@Pzero
\pIIe@moveto{\@tempa\unitlength}{\@tempb\unitlength}%
\GetCoord(#2)\@tempa\@tempb
\CopyVect\@tempa,\@tempb to\@Dzero
\DirOfVect\@Dzero to\@Dzero
\ignorespaces}
```

And this re-initialises the direction to create a cusp:

```
\def\ChangeDir<#1>{%
\GetCoord(#1)\@tempa\@tempb
\CopyVect\@tempa,\@tempb to\@Dzero
\DirOfVect\@Dzero to\@Dzero
\ignorespaces}
```

The next macros are the finishing ones; the first strokes the whole curve, while the second fills the (closed) curve with the default color; both close the group that was opened with `\StartCurve`. The third macro is explained in a while; we anticipate it is functional to chose between the first two macros when a star is possibly used to switch between stroking and filling.

```
\def\CurveFinish{\strokepath\endgroup\ignorespaces}%
\def\FillCurve{\fillpath\endgroup\ignorespaces}
\def\CurveEnd{\fillstroke\endgroup\ignorespaces}
```

In order to draw the internal arcs it would be desirable to have a single macro that, given the destination point, computes the control points that produce a cubic Bézier spline that joins the starting point with the destination point in the best possible way. The problem is strongly ill defined and has an infinity of solutions; here we give two solutions: (a) a supposedly smart one that resorts to osculating circles and requires only the direction at the destination point; and (b) a less smart solution that requires the control points to be specified in a certain format.

We start with solution (b), `\CbezierTo`, the code of which is simpler than that of solution (a); then we will produce the solution (a), `\CurveTo`, that will become the main building block for a general path construction macro, `\Curve`.

The "naïve" macro `\CBezierTo` simply uses the previous point direction saved in `\@Dzero` as a unit vector by the starting macro; specifies a destination point, the distance of the first control point from the starting point, the destination point direction that will save also for the next arc-drawing macro as a unit vector, and the distance of the second control point from the destination point along this

last direction. Both distances must be positive possibly fractional numbers. The syntax therefore is the following:

`\CbezierTo`⟨*end point*⟩`WithDir`⟨*direction*⟩`AndDists`⟨K_0⟩`And`⟨K_1⟩

where ⟨*end point*⟩ is a vector macro or a comma separated pair of values; again ⟨*direction*⟩ is another vector macro or a comma separated pair of values, that not necessarily indicate a unit vector, since the macro provides to normalise it to unity; ⟨K_0⟩ and ⟨K_1⟩ are the distances of the control points from their respective node points; they must be positive integers or fractional numbers. If ⟨K_1⟩ is a number, it must be enclosed in curly braces, while if it is a macro name (containing the desired fractional or integer value) there is no need for braces.

This macro uses the input information in order to activate the internal `pict2e` macro `\pIIe@curveto` with the proper arguments, and to save the final direction into the same `\@Dzero` macro for successive use of other arc-drawing macros.

```
\def\CbezierTo#1WithDir#2AndDists#3And#4{%
\GetCoord(#1)\@tX\@tY \MakeVectorFrom\@tX\@tY to\@Puno
\GetCoord(#2)\@tX\@tY \MakeVectorFrom\@tX\@tY to \@Duno
\DirOfVect\@Duno to\@Duno
\ScaleVect\@Dzero by#3to\@Czero \AddVect\@Pzero and\@Czero to\@Czero
\ScaleVect\@Duno by-#4to \@Cuno \AddVect\@Puno and\@Cuno to \@Cuno
\GetCoord(\@Czero)\@XCzero\@YCzero
\GetCoord(\@Cuno)\@XCuno\@YCuno
\GetCoord(\@Puno)\@XPuno\@YPuno
\pIIe@curveto{\@XCzero\unitlength}{\@YCzero\unitlength}%
             {\@XCuno\unitlength}{\@YCuno\unitlength}%
             {\@XPuno\unitlength}{\@YPuno\unitlength}%
\CopyVect\@Puno to\@Pzero
\CopyVect\@Duno to\@Dzero
\ignorespaces}%
```

With this building block it is not difficult to set up a macro that draws a Bézier arc between two given points, similarly to the other macro `\CurveBetween` previously described and defined here:

```
\def\CbezierBetween#1And#2WithDirs#3And#4UsingDists#5And#6{%
\StartCurveAt#1WithDir{#3}\relax
\CbezierTo#2WithDir#4AndDists#5And{#6}\CurveFinish}
```

An example of use is shown in figure 13 of the user manual `curve2e-manual.pdf` file; notice that the tangents at the end points are the same for the black curve drawn with `\CurveBetween` and the five red curves drawn with `\CbezierBetween`; the five red curves differ only for the distance of their control point C_0 from the starting point; the differences are remarkable and the topmost curve even presents a slight inflection close to the end point. These effects cannot be obtained with the "smarter" macro `\CurveBetween`. But certainly this simpler macro is more difficult to use because the distances of the control points are difficult to estimate and require a number of cut-and-try experiments.

The "smarter" curve macro comes next; it is supposed to determine the control points for joining the previous point (initial node) with the specified direction to the next point (final node) with another specified direction.

Since the control points are along the specified directions, it is necessary to determine the distances from the adjacent curve nodes. This must work correctly even if nodes and directions imply an inflection point somewhere along the arc.

The strategy we devised consists in determining each control point as if it were the control point of a circular arc, precisely an arc of an osculating circle, i.e. a circle tangent to the curve at that node. The ambiguity of the stated problem may be solved by establishing that the chord of the osculating circle has the same direction as the chord of the arc being drawn, and that the curve chord is divided into two equal parts each of which should be interpreted as half the chord of the osculating circle.

This makes the algorithm a little rigid; sometimes the path drawn is very pleasant, while in other circumstances the determined curvatures are too large or too small. We therefore add some optional information that lets us have some control over the curvatures; the idea is based on the concept of *tension*, similar but not identical to the one used in the drawing programs METAFONT and METAPOST. We add to the direction information, with which the control nodes of the osculating circle arcs are determined, a scaling factor that should be intuitively related to the tension of the arc (actually, since the tension of the 'rope' is high when this parameter is low, probably a name such as 'looseness' would be better suited): the smaller this number, the closer the arc resembles a straight line as a rope subjected to a high tension; value zero is allowed, while a value of 4 is close to "infinity" and turns a quarter circle into a line with an unusual loop; a value of 2 turns a quarter circle almost into a polygonal line with rounded vertices . Therefore these tension factors should be used only for fine tuning the arcs, not when a path is drawn for the first time.

We devised a syntax for specifying direction and tensions:

⟨*direction*; *tension factors*⟩

where *direction* contains a pair of fractional number that not necessarily refer to the components of a unit vector direction, but simply to a vector with the desired orientation (polar form is OK); the information contained from the semicolon (included) to the rest of the specification is optional; if it is present, the *tension factors* is simply a comma separated pair of fractional or integer numbers that represent respectively the tension at the starting or the ending node of a path arc.

We therefore need a macro to extract the mandatory and optional parts:

```
590 \def\@isTension#1;#2!!{\def\@tempA{#1}%
591 \def\@tempB{#2}\unless\ifx\@tempB\empty\strip@semicolon#2\fi}
592
593 \def\strip@semicolon#1;{\def\@tempB{#1}}
```

By changing the tension values we can achieve different results: see figure 14 in the user manual `curve2e-manual.pdf`.

We use the formula we got for arcs (1), where the half chord is indicated with

s, and we derive the necessary distances:

$$K_0 = \frac{4}{3} s \frac{1 - \cos\theta_0}{\sin^2\theta_0} \tag{5a}$$

$$K_1 = \frac{4}{3} s \frac{1 - \cos\theta_1}{\sin^2\theta_1} \tag{5b}$$

We therefore start with getting the points and directions and calculating the chord and its direction:

```
\def\CurveTo#1WithDir#2{%
\def\@Tuno{1}\def\@Tzero{1}\relax
\edef\@Puno{#1}\@isTension#2;!!%
\expandafter\DirOfVect\@tempA to\@Duno
\bgroup\unless\ifx\@tempB\empty\GetCoord(\@tempB)\@Tzero\@Tuno\fi
\DistanceAndDirOfVect\@Puno minus\@Pzero to\@Chord and\@DirChord
```

Then we rotate everything about the starting point so as to bring the chord on the real axis

```
\Multvect{\@Dzero}*{\@DirChord}\@Dpzero
\Multvect{\@Duno}*{\@DirChord}\@Dpuno
\GetCoord(\@Dpzero)\@DXpzero\@DYpzero
\GetCoord(\@Dpuno)\@DXpuno\@DYpuno
\DivideFN\@Chord by2 to\@semichord
```

The chord needs not be actually rotated because it suffices its length along the real axis; the chord length is memorised in `\@Chord` and its half is saved in `\@semichord`.

We now examine the various degenerate cases, when either tangent is perpendicular or parallel to the chord. Notice that we are calculating the distances of the control points from the adjacent nodes using the half chord length, not the full length. We also distinguish between the computations relative to the arc starting point and those relative to the end point. Notice that if the directions of two successive nodes are identical, it is necessary to draw a line, not a third order spline[5]; therefore it is necessary to make a suitable test that is more comfortable to do after the chord has been rotated to be horizontal; in facts, if the two directions are equal, the vertical componente of the directions are both vanishing values; probably, instead of testing with respect to zero, it might be advisable to test the absolute value with respect to a small number such as, for example, "1.e-6."

```
\fptest{\@DYpuno=0 && \@DYpzero=0}{\GetCoord(\@Puno)\@tX\@tY
  \pIIe@lineto{\@tX\unitlength}{\@tY\unitlength}}%
{\ifdim\@DXpzero\p@=\z@
   \@tdA=1.333333\p@
   \Numero\@KCzero{\@semichord\@tdA}%
\fi
\ifdim\@DYpzero\p@=\z@
  \@tdA=1.333333\p@
```

[5] Many thanks to John Hillas who spotted this bug, that passed unnoticed fora long time, because it is a very unusual situation.

```
  \Numero\@Kpzero{\@semichord\@tdA}%
\fi
```

The distances we are looking for are positive generally fractional numbers; so if the components are negative, we take the absolute values. Eventually we determine the absolute control point coordinates.

```
\unless\ifdim\@DXpzero\p@=\z@
  \unless\ifdim\@DYpzero\p@=\z@
    \edef\@CosDzero{\ifdim\@DXpzero\p@<\z@ -\fi\@DXpzero}%
    \edef\@SinDzero{\ifdim\@DYpzero\p@<\z@ -\fi\@DYpzero}%
    \@tdA=\@semichord\p@ \@tdA=1.333333\@tdA
    \DividE\@tdA by\@SinDzero\p@ to \@KCzero
    \@tdA=\dimexpr(\p@-\@CosDzero\p@)\relax
    \DividE\@KCzero\@tdA by\@SinDzero\p@ to \@KCzero
  \fi
\fi
\MultiplyFN\@KCzero by \@Tzero to \@KCzero
\ScaleVect\@Dzero by\@KCzero to\@CPzero
\AddVect\@Pzero and\@CPzero to\@CPzero
```

We now repeat the calculations for the arc end point, taking into consideration that the end point direction points outwards, so that in computing the end point control point we have to take this fact into consideration by using a negative sign for the distance; in this way the displacement of the control point from the end point takes place in a backwards direction.

```
\ifdim\@DXpuno\p@=\z@
   \@tdA=-1.333333\p@
   \Numero\@KCuno{\@semichord\@tdA}%
\fi
\ifdim\@DYpuno\p@=\z@
  \@tdA=-1.333333\p@
  \Numero\@KCuno{\@semichord\@tdA}%
\fi
\unless\ifdim\@DXpuno\p@=\z@
    \unless\ifdim\@DYpuno\p@=\z@
    \edef\@CosDuno{\ifdim\@DXpuno\p@<\z@ -\fi\@DXpuno}%
    \edef\@SinDuno{\ifdim\@DYpuno\p@<\z@ -\fi\@DYpuno}%
    \@tdA=\@semichord\p@ \@tdA=-1.333333\@tdA
    \DividE\@tdA by \@SinDuno\p@ to \@KCuno
    \@tdA=\dimexpr(\p@-\@CosDuno\p@)\relax
    \DividE\@KCuno\@tdA by\@SinDuno\p@ to \@KCuno
  \fi
\fi
\MultiplyFN\@KCuno by \@Tuno to \@KCuno
\ScaleVect\@Duno by\@KCuno to\@CPuno
\AddVect\@Puno and\@CPuno to\@CPuno
```

Now we have the four points and we can instruct the internal `pict2e` macros to do the path drawing.

```
\GetCoord(\@Puno)\@XPuno\@YPuno
```

```
\GetCoord(\@CPzero)\@XCPzero\@YCPzero
\GetCoord(\@CPuno)\@XCPuno\@YCPuno
\pIIe@curveto{\@XCPzero\unitlength}{\@YCPzero\unitlength}%
             {\@XCPuno\unitlength}{\@YCPuno\unitlength}%
             {\@XPuno\unitlength}{\@YPuno\unitlength}}\egroup
```

It does not have to stroke the curve because other Bézier splines might still be added to the path. On the opposite it memorises the final point to be used as the initial point of the next spline

```
\CopyVect\@Puno to\@Pzero
\CopyVect\@Duno to\@Dzero
\ignorespaces}%
```

We finally define the overall `\Curve` macro that has two flavours: starred and unstarred; the former fills the curve path with the locally selected color, while the latter just strokes the path. Both recursively examine an arbitrary list of nodes and directions; node coordinates are grouped within round parentheses while direction components are grouped within angle brackets. Before testing for a possible star, this initial command kills any space or glue that might precede it[6] The first call of the macro initialises the drawing process and checks for the next node and direction; if a second node is missing, it issues a warning message and does not draw anything. It does not check for a change in direction, because it would be meaningless at the beginning of a curve. The second macro defines the path to the next point and checks for another node; if the next list item is a square bracket delimited argument, it interprets it as a change of direction, while if it is another parenthesis delimited argument it interprets it as a new node-direction specification; if the node and direction list is terminated, it issues the stroking or filling command through `\CurveEnd`, and exits the recursive process. The `\CurveEnd` control sequence has a different meaning depending on the fact that the main macro was starred or unstarred. The `@ChangeDir` macro is just an interface to execute the regular `\ChangeDir` macro, but also for recursing again by recalling `\@Curve`.

```
\def\Curve{\@killglue\@ifstar{\let\fillstroke\fillpath\Curve@}%
{\let\fillstroke\strokepath\Curve@}}

\def\Curve@(#1)<#2>{%
    \StartCurveAt#1WithDir{#2}%
    \@ifnextchar\lp@r\@Curve{%
    \PackageWarning{curve2e}{%
    Curve specifications must contain at least two nodes!\Messagebreak
    Please, control your \string\Curve\space specifications\MessageBreak}}}
\def\@Curve(#1)<#2>{%
    \CurveTo#1WithDir{#2}%
    \@ifnextchar\lp@r\@Curve{%
    \@ifnextchar[\@ChangeDir\CurveEnd}}
\def\@ChangeDir[#1]{\ChangeDir<#1>\@Curve}
```

[6] Thanks to John Hillas who spotted the effects of this missing glue elimination.

As a concluding remark, please notice that the `\Curve` macro is certainly the most comfortable to use, but it is sort of frozen in its possibilities. The user may certainly use the `\StartCurve`, `\CurveTo`, `\ChangeDir`, and `\CurveFinish` or `\FillCurve` for a more versatile set of drawing macros; evidently nobody forbids to exploit the full power of the `\cbezier` original macro for cubic splines; we made available macros `\CbezierTo` and the isolated arc macro `\CbezierBetween` in order to use the general internal cubic Bézier splines in a more comfortable way.

As it can be seen in figure 15 of the `curve2e-manual.pdf` file, the two diagrams should approximately represent a sine wave. With Bézier curves, that resort on polynomials, it is impossible to represent a transcendental function, but it is only possible to approximate it. It is evident that the approximation obtained with full control on the control points requires less arcs and it is more accurate than the approximation obtained with the recursive `\Curve` macro; this macro requires almost two times as many pieces of information in order to minimise the effects of the lack of control on the control points, and even with this added information the macro approaches the sine wave with less accuracy. At the same time for many applications the `\Curve` recursive macro proves to be much easier to use than single arcs drawn with the `\CbezierBetween` macro.

4.8 Quadratic splines

We want to create a recursive macro with the same properties as the above described `\Curve` macro, but that uses quadratic splines; we call it `\Qurve` so that the macro name initial letter reminds us of the nature of the splines being used. For the rest they have an almost identical syntax; with quadratic splines it is not possible to specify the distance of the control points from the extrema, since quadratic splines have just one control point that must lay at the intersection of the two tangent directions; therefore with quadratic splines the tangents at each point cannot have the optional part that starts with a semicolon. The syntax, therefore, is just:

`\Qurve(`⟨*first point*⟩`)<`⟨*direction*⟩`>...(`⟨*any point*⟩`)<`⟨*direction*⟩`>...(`⟨*last point*⟩`)<`⟨*direction*⟩`>`

As with `\Curve`, also with `\Qurve` there is no limitation on the number of points, except for the computer memory size; it is advisable not to use many arcs otherwise it might become very difficult to find errors.

The first macros that set up the recursion are very similar to those we wrote for `\Curve`:

```
672 \def\Qurve{\@ifstar{\let\fillstroke\fillpath\Qurve@}%
673 {\let\fillstroke\strokepath\Qurve@}}
674
675 \def\Qurve@(#1)<#2>{%
676     \StartCurveAt#1WithDir{#2}%
677     \@ifnextchar\lp@r\@Qurve{%
678     \PackageWarning{curve2e}{%
679       Quadratic curve specifications must contain at least
```

```
      two nodes!\Messagebreak
      Please, control your Qurve specifications\MessageBreak}}}%

\def\@Qurve(#1)<#2>{\QurveTo#1WithDir{#2}%
    \@ifnextchar\lp@r\@Qurve{%
    \@ifnextchar[\@ChangeQDir\CurveEnd}}%

\def\@ChangeQDir[#1]{\ChangeDir<#1>\@Qurve}%
```

Notice that in case of long paths it might be better to use the single macros `\StartCurveAt`, `\QurveTo`, `\ChangeDir` and `\CurveFinish` (or `\FillCurve`), with their respective syntax, in such a way that a long list of node-direction specifications passed to `\Qurve` may be split into shorter input lines in order to edit the input data in a more comfortable way.

The macro that does everything is `\QurveTo`. It starts by reading its arguments received through the calling macro `\@Qurve`

```
\def\QurveTo#1WithDir#2{%
\edef\@Puno{#1}\DirOfVect#2to\@Duno\bgroup
\DistanceAndDirOfVect\@Puno minus\@Pzero to\@Chord and\@DirChord
```

It verifies if `\@Dpzero` and `\@Dpuno`, the directions at the two extrema of the arc, are parallel or anti-parallel by taking their "scalar" product (`\@Dpzero` times `\@Dpuno*`); if the imaginary component of the scalar product vanishes the two directions are parallel; in this case we produce an error message, but we continue by skipping this arc destination point; evidently the drawing will not be the desired one, but the job should not abort.

```
\Multvect{\@Dzero}*{\@Duno}\@Scalar
\YpartOfVect\@Scalar to \@YScalar
\ifdim\@YScalar\p@=\z@
\PackageWarning{curve2e}%
   {Quadratic Bezier arcs cannot have their starting\MessageBreak
   and ending directions parallel or antiparallel with\MessageBreak
   each other. This arc is skipped and replaced with
   a dotted line.\MessageBreak}%
   \Dotline(\@Pzero)(\@Puno){2}\relax
\else
```

Otherwise we rotate everything about the starting point so as to bring the chord on the real axis; we get also the components of the two directions that, we should remember, are unit vectors, not generic vectors, although the user can use the vector specifications that are more understandable to him/her:

```
\Multvect{\@Dzero}*{\@DirChord}\@Dpzero
\Multvect{\@Duno}*{\@DirChord}\@Dpuno
\GetCoord(\@Dpzero)\@DXpzero\@DYpzero
\GetCoord(\@Dpuno)\@DXpuno\@DYpuno
```

We check if the two directions point to the same half plane; this implies that these rotated directions point to different sides of the chord vector; all this is equivalent to the fact that the two direction Y components have opposite signs, so that their product is strictly negative, while the two X components product is not negative.

```
\MultiplyFN\@DXpzero by\@DXpuno to\@XXD
\MultiplyFN\@DYpzero by\@DYpuno to\@YYD
\unless\ifdim\@YYD\p@<\z@\ifdim\@XXD\p@<\z@
\PackageWarning{curve2e}%
   {Quadratic Bezier arcs cannot have inflection points\MessageBreak
   Therefore the tangents to the starting and ending arc\MessageBreak
   points cannot be directed to the same half plane.\MessageBreak
   This arc is skipped and replaced by a dotted line\MessageBreak}%
   \Dotline(\@Pzero)(\@Puno){2}\fi
\else
```

After these tests we should be in a "normal" situation. We first copy the expanded input information into new macros that have more explicit names: macros stating wit 'S' denote the sine of the direction angle, while those starting with 'C' denote the cosine of that angle. We will use these expanded definitions as we know we are working with the actual values. These directions are those relative to the arc chord.

```
\edef\@CDzero{\@DXpzero}\relax
\edef\@SDzero{\@DYpzero}\relax
\edef\@CDuno{\@DXpuno}\relax
\edef\@SDuno{\@DYpuno}\relax
```

Suppose we write the parametric equations of a straight line that departs from the beginning of the chord with direction angle ϕ_0 and the corresponding equation of the straight line departing from the end of the chord (of length c) with direction angle ϕ_1. We have to find the coordinates of the intersection point of these two straight lines.

$$t\cos\phi_0 - s\cos\phi_1 = c \tag{6a}$$

$$t\sin\phi_0 - s\sin\phi_1 = 0 \tag{6b}$$

The parameters t and s are just the running parameters; we have to solve those simultaneous equations in the unknown variables t and s; these values let us compute the coordinates of the intersection point:

$$X_C = \frac{c\cos\phi_0\sin\phi_1}{\sin\phi_0\cos\phi_1 - \cos\phi_0\sin\phi_1} \tag{7a}$$

$$Y_C = \frac{c\sin\phi_0\sin\phi_1}{\sin\phi_0\cos\phi_1 - \cos\phi_0\sin\phi_1} \tag{7b}$$

Having performed the previous tests we are sure that the denominator is not vanishing (direction are not parallel or anti-parallel) and that it lays at the same side as the direction with angle ϕ_0 with respect to the chord.

The coding then goes on like this:

```
\MultiplyFN\@SDzero by\@CDuno to\@tempA
\MultiplyFN\@SDuno by\@CDzero to\@tempB
\edef\@tempA{\strip@pt\dimexpr\@tempA\p@-\@tempB\p@}\relax
\@tdA=\@SDuno\p@ \@tdB=\@Chord\p@ \@tdC=\@tempA\p@
\edef\@tempC{\strip@pt\dimexpr \@tdA*\@tdB/\@tdC}\relax
```

```
\MultiplyFN\@tempC by\@CDzero to \@XC
\MultiplyFN\@tempC by\@SDzero to \@YC
\ModOfVect\@XC,\@YC to\@KC
```

Now we have the coordinates and the module of the intersection point vector taking into account the rotation of the real axis; getting back to the original coordinates before rotation, we get:

```
\ScaleVect\@Dzero by\@KC to\@CP
\AddVect\@Pzero and\@CP to\@CP
\GetCoord(\@Pzero)\@XPzero\@YPzero
\GetCoord(\@Puno)\@XPuno\@YPuno
\GetCoord(\@CP)\@XCP\@YCP
```

We have now the coordinates of the two end points of the quadratic arc and of the single control point. Keeping in mind that the symbols P_0, P_1 and C denote geometrical points but also their coordinates as ordered pairs of real numbers (i.e. they are complex numbers) we have to determine the parameters of a cubic spline that with suitable values gets simplifications in its parametric equation so that it becomes a second degree function instead of a third degree one. It is possible, even if it appears impossible that e cubic form becomes a quadratic one; we should determine the values of P_a and P_b such that:

$$P_0(1-t)^3 + 3P_a(1-t)^2t + 3P_b(1-t)t^2 + P_1t^3$$

is equivalent to

$$P_0(1-t)^2 + 2C(1-t)t + P_1t^2$$

It turns out that the solution is given by

$$P_a = C + (P_0 - C)/3 \qquad \text{and} \qquad P_b = C + (P_1 - C)/3 \tag{8}$$

The transformations implied by equations (8) are performed by the following macros already available from the `pict2e` package; we use them here with the actual arguments used for this task:

```
\@ovxx=\@XPzero\unitlength \@ovyy=\@YPzero\unitlength
\@ovdx=\@XCP\unitlength \@ovdy=\@YCP\unitlength
\@xdim=\@XPuno\unitlength \@ydim=\@YPuno\unitlength
      \pIIe@bezier@QtoC\@ovxx\@ovdx\@ovro
      \pIIe@bezier@QtoC\@ovyy\@ovdy\@ovri
      \pIIe@bezier@QtoC\@xdim\@ovdx\@clnwd
      \pIIe@bezier@QtoC\@ydim\@ovdy\@clnht
```

We call the basic `pict2e` macro to draw a cubic spline and we finish the conditional statements with which we started these calculations; eventually we close the group we opened at the beginning and we copy the terminal node information (position and direction) into the zero-labelled macros that indicate the starting point of the next arc.

```
      \pIIe@curveto\@ovro\@ovri\@clnwd\@clnht\@xdim\@ydim
\fi\fi\egroup
\CopyVect\@Puno to\@Pzero
\CopyVect\@Duno to\@Dzero
\ignorespaces}
```

An example of usage is shown at the left in figure 16[7] created with the code shown in the same page as the figure.

Notice also that the inflexed line is made with two arcs that meet at the inflection point; the same is true for the line that resembles a sine wave. The cusps of the inner border of the green area are obtained with the usual optional argument already used also with the `\Curve` recursive macro.

The "circle" inside the square frame is visibly different from a real circle, in spite of the fact that the maximum deviation from the true circle is just about 6% relative to the radius; a quarter circle obtained with a single parabola is definitely a poor approximation of a real quarter circle; possibly by splitting each quarter circle in three or four partial arcs the approximation of a real quarter circle would be much better. On the right of figure 16 of the user manual it is possible to compare a "circle" obtained with quadratic arcs with the the internal circle obtained with cubic arcs; the difference is easily seen even without using measuring instruments.

With quadratic arcs we decided to avoid defining specific macros similar to `\CurveBetween` and `\CbezierBetween`; the first macro would not save any typing to the operator; furthermore it may be questionable if it was really useful even with cubic splines; the second macro with quadratic arcs is meaningless, since with quadratic arcs there is just one control point and there is no choice on its position.

5 Conclusion

I believe that the set of new macros provided by this package can really help the user to draw his/her diagrams with more agility; it will be the accumulated experience to decide if this is true.

As a personal experience I found very comfortable to draw ellipses and to define macros to draw not only such shapes or filled elliptical areas, but also to create "legends" with coloured backgrounds and borders. But this is just an application of the functionality implemented in this package. In 2020 I added to CTAN another specialized package, `euclideangeometry.sty` with its manual `euclideangeometry-man.pdf` that uses the facilities of `curve2e` to draw complex diagrams that plot curves and others that solve some geometrical problems dealing with ellipses.

6 The `README.txt` file

The following is the text that forms the contents of the `README.txt` file that accompanies the package. We found it handy to have it in the documented source,

[7] The commands `\legenda`, `\Pall` and `\Zbox` are specifically defined in the preamble of this document; they must be used within a `picture` environment. `\legenda` draws a framed legend made up of a single (short) math formula; `\Pall` is just a shorthand to put a sized dot at a specified position' `\Zbox` puts a symbol in math mode a little displaced in the proper direction relative to a specified position. They are just handy to label certain objects in a `picture` diagram, but they are not part of the `curve2e` package.

because in this way certain pieces of information don't need to be repeated again and again in different files.

```
The package bundle curve2e is composed of the following files

curve2e.dtx
curve2e-manual.tex

The derived files are

curve2e.sty
curve2e-v161.sty
curve2e.pdf
curve2e-manual.pdf
README.txt

Compile curve2e.dtx and curve2e-manual.tex two or three times until
all labels and citation keys are completely resolved.

Move curve2e.dtx and curve2e-manual.tex to ROOT/source/latex/curve2e/
Move curve2e.pdf and curve2e-manual.pdf to ROOT/doc/latex/curve2e/
Move curve2e.sty and curve2e-v161.sty   to ROOT/tex/latex/curve2e/
Move README.txt                         to ROOT/doc/latex/curve2e/

curve2e.dtx is the documented TeX source file of the derived files
curve2e.sty, curve2e-v161.sty and README.txt.

You get curve2e.sty, curve2e.pdf, curve2e-v161.sty, and README.txt
by running pdflatex on curve2e.dtx.

The curve2e-manual files contains the user manual; in
this way the long preliminary descriptive part has been transferred to
a shorter dedicated file, and the \normal" user should have enough
information to use the package. Th curve2e.pdf file, extracted from
the .dtx one, contains the code documentation and is intended for the
developers, or for the curious advanced users. For what concerns
curve2e-v161.sty is a previous version of this package; see below why
the older version might become necessary to the end user.

README.txt, this file, contains general information.
This bundle contains also package curve2e-v161.sty, a roll-back
version needed in certain rare cases.

Curve2e.sty is an extension of the package pict2e.sty which extends the
standard picture LaTeX environment according to what Leslie Lamport
specified in the second edition of his LaTeX manual (1994).

This further extension curve2e.sty to pict2e.sty allows to draw lines
and vectorsith any non integer slope parameters, to draw dashed and dotted
lines of any slope, to draw arcs and curved vectors, to draw curves where
```

just the interpolating nodes are specified together with the slopes at such nodes; closed paths of any shape can be filled with color; all coordinates are treated as ordered pairs, i.e. 'complex numbers'; coordinates may be expressed also in polar form. Coordinates may be specified with macros, so that editing any drawing is rendered much simpler: any point specified with a macro is modified only once in its macro definition.
Some of these features have been incorporated in the 2009 version of pict2e; therefore this package avoids any modification to the original pict2e commands. In any case the version of curve2e is compatible with later versions of pict2e; see below.

Curve2e now accepts polar coordinates in addition to the usual cartesian ones; several macros have been upgraded; a new macro for tracing cubic Bezier splines with their control nodes specified in polar form is available. The same applies to quadratic Bezier splines. The multiput command has been completely modified in a backwards compatible way; the new version allows to manipulate the increment components in a configurable way. A new xmultiput command has been defined that is more configurable than the original one; both commands multiput and xmultiput are backwards compatible with the original picture environment definition.

Curve2e solves a conflict with package eso-pic.

This version of curve2e is almost fully compatible with pict2e dated 2014/01/12 version 0.2z and later.

If you specify

\usepackage[<pict2e options>]{curve2e}

the package pict2e is automatically invoked with the specified options.

The -almost fully compatible- phrase is necessary to explain that this version of curve2e uses some 'functions' of the LaTeX3 language that were made available to the LaTeX developers by mid October 2018. Should the user have an older or a basic/incomplete installation of the TeX system, such L3 functions might not be available. This is why this package checks the presence of the developer interface; in case such interface is not available it rolls back to the previous version renamed curve2e-v161.sty, which is part of this bundle; this roll-back file name must not be modified in any way. The compatibility mentioned above implies that the user macros remain the same, but their implementation requires the L3 interface. Some macros and environments rely totally on the xfp package functionalities, but legacy documents source files should compile correctly.

The package has the LPPL status of maintained.

According to the LPPL licence, you are entitled to modify this package,

```
as long as you fulfil the few conditions set forth by the Licence.

Nevertheless this package is an extension to the standard LaTeX
pict2e (2014) package. Therefore any change must be controlled on the
parent package pict2e, so as to avoid redefining or interfering with
what is already contained in that package.

If you prefer sending me your modifications, as long as I will maintain
this package, I will possibly include every (documented) suggestion or
modification into this package and, of course, I will acknowledge your
contribution.

Claudio Beccari

claudio dot beccari at gmail dot com
```

7 The roll-back package version curve2e-v161

this is the fall-back version of `curve2e-v161.sty` to which the main file `curve2e.sty` falls back in case the interface package `xfp` is not available.

```
\NeedsTeXFormat{LaTeX2e}[2016/01/01]
\ProvidesPackage{curve2e-v161}%
        [2019/02/07 v.1.61 Extension package for pict2e]

\RequirePackage{color}
\RequirePackageWithOptions{pict2e}[2014/01/01]
\RequirePackage{xparse}
\def\TRON{\tracingcommands\tw@ \tracingmacros\tw@}%
\def\TROF{\tracingcommands\z@ \tracingmacros\z@}%
\ifx\undefined\@tdA \newdimen\@tdA \fi
\ifx\undefined\@tdB \newdimen\@tdB \fi
\ifx\undefined\@tdC \newdimen\@tdC \fi
\ifx\undefined\@tdD \newdimen\@tdD \fi
\ifx\undefined\@tdE \newdimen\@tdE \fi
\ifx\undefined\@tdF \newdimen\@tdF \fi
\ifx\undefined\defaultlinewidth \newdimen\defaultlinewidth \fi
\gdef\linethickness#1{\@wholewidth#1\@halfwidth.5\@wholewidth\ignorespaces}%
\newcommand\defaultlinethickness[1]{\defaultlinewidth=#1\relax
\def\thicklines{\linethickness{\defaultlinewidth}}%
\def\thinlines{\linethickness{.5\defaultlinewidth}}%
\thinlines\ignorespaces}
\def\LIne(#1){{\GetCoord(#1)\@tX\@tY
      \moveto(0,0)
      \pIIe@lineto{\@tX\unitlength}{\@tY\unitlength}\strokepath}\ignorespaces}%
\def\segment(#1)(#2){\@killglue\polyline(#1)(#2)}%
\def\line(#1)#2{\begingroup
  \@linelen #2\unitlength
  \ifdim\@linelen<\z@\@badlinearg\else
```

```
    \expandafter\DirOfVect#1to\Dir@line
    \GetCoord(\Dir@line)\d@mX\d@mY
    \ifdim\d@mX\p@=\z@\else
      \DividE\ifdim\d@mX\p@<\z@-\fi\p@ by\d@mX\p@ to\sc@lelen
      \@linelen=\sc@lelen\@linelen
    \fi
    \moveto(0,0)
    \pIIe@lineto{\d@mX\@linelen}{\d@mY\@linelen}%
    \strokepath
  \fi
\endgroup\ignorespaces}%
\ifx\Dashline\undefined
\def\Dashline{\@ifstar{\Dashline@@}{\Dashline@}}
\def\Dashline@(#1)(#2)#3{%
\bgroup
   \countdef\NumA3254\countdef\NumB3252\relax
   \GetCoord(#1)\@tA\@tB \MakeVectorFrom\@tA\@tB to\V@ttA
   \GetCoord(#2)\@tA\@tB \MakeVectorFrom\@tA\@tB to\V@ttB
   \SubVect\V@ttA from\V@ttB to\V@ttC
   \ModOfVect\V@ttC to\DlineMod
   \DivideFN\DlineMod by#3 to\NumD
   \NumA\expandafter\Integer\NumD.??
   \ifodd\NumA\else\advance\NumA\@ne\fi
   \NumB=\NumA \divide\NumB\tw@
   \DividE\DlineMod\p@ by\NumA\p@  to\D@shMod
   \DividE\p@ by\NumA\p@ to \@tempa
   \MultVect\V@ttC by\@tempa,0 to\V@ttB
   \MultVect\V@ttB by 2,0 to\V@ttC
   \advance\NumB\@ne
   \edef\@mpt{\noexpand\egroup
   \noexpand\multiput(\V@ttA)(\V@ttC){\number\NumB}%
      {\noexpand\LIne(\V@ttB)}}%
   \@mpt\ignorespaces}%
\let\Dline\Dashline

\def\Dashline@@(#1)(#2)#3{\put(#1){\Dashline@(0,0)(#2){#3}}}
\fi
\ifx\Dotline\undefined
\def\Dotline{\@ifstar{\Dotline@@}{\Dotline@}}
\def\Dotline@(#1)(#2)#3{%
\bgroup
   \countdef\NumA 3254\relax \countdef\NumB 3255\relax
   \GetCoord(#1)\@tA\@tB \MakeVectorFrom\@tA\@tB to\V@ttA
   \GetCoord(#2)\@tA\@tB \MakeVectorFrom\@tA\@tB to\V@ttB
   \SubVect\V@ttA from\V@ttB to\V@ttC
   \ModOfVect\V@ttC to\DotlineMod
   \DivideFN\DotlineMod by#3 to\NumD
   \NumA=\expandafter\Integer\NumD.??
   \DivVect\V@ttC by\NumA,0 to\V@ttB
   \advance\NumA\@ne
```

```
   \edef\@mpt{\noexpand\egroup
   \noexpand\multiput(\V@ttA)(\V@ttB){\number\NumA}%
      {\noexpand\makebox(0,0){\noexpand\circle*{0.5}}}}%
   \@mpt\ignorespaces}%

\def\Dotline@@(#1)(#2)#3{\put(#1){\Dotline@(0,0)(#2){#3}}}
\fi
\AtBeginDocument{\@ifpackageloaded{eso-pic}{%
\renewcommand\LenToUnit[1]{\strip@pt\dimexpr#1*\p@/\unitlength}}{}}

\def\GetCoord(#1)#2#3{%
\expandafter\SplitNod@\expandafter(#1)#2#3\ignorespaces}
\def\isnot@polar#1:#2!!{\def\@tempOne{#2}\ifx\@tempOne\empty
\expandafter\@firstoftwo\else
\expandafter\@secondoftwo\fi
{\SplitNod@@}{\SplitPolar@@}}

\def\SplitNod@(#1)#2#3{\isnot@polar#1:!!(#1)#2#3}%
\def\SplitNod@@(#1,#2)#3#4{\edef#3{#1}\edef#4{#2}}%
\def\SplitPolar@@(#1:#2)#3#4{\DirFromAngle#1to\@DirA
\ScaleVect\@DirA by#2to\@DirA
\expandafter\SplitNod@@\expandafter(\@DirA)#3#4}

\let\originalput\put
\def\put(#1){\bgroup\GetCoord(#1)\@tX\@tY
\edef\x{\noexpand\egroup\noexpand\originalput(\@tX,\@tY)}\x}

\let\originalmultiput\multiput
\let\original@multiput\@multiput

\long\def\@multiput(#1)#2#3{\bgroup\GetCoord(#1)\@mptX\@mptY
\edef\x{\noexpand\egroup\noexpand\original@multiput(\@mptX,\@mptY)}%
\x{#2}{#3}\ignorespaces}

\gdef\multiput(#1)#2{\bgroup\GetCoord(#1)\@mptX\@mptY
\edef\x{\noexpand\egroup\noexpand\originalmultiput(\@mptX,\@mptY)}\x(}%)
  \def\vector(#1)#2{%
    \begingroup
      \GetCoord(#1)\d@mX\d@mY
      \@linelen#2\unitlength
      \ifdim\d@mX\p@=\z@\ifdim\d@mY\p@=\z@\@badlinearg\fi\fi
      \ifdim\@linelen<\z@ \@linelen=-\@linelen\fi
      \MakeVectorFrom\d@mX\d@mY to\@Vect
      \DirOfVect\@Vect to\Dir@Vect
        \YpartOfVect\Dir@Vect to\@ynum \@ydim=\@ynum\p@
        \XpartOfVect\Dir@Vect to\@xnum \@xdim=\@xnum\p@
        \ifdim\d@mX\p@=\z@
        \else\ifdim\d@mY\p@=\z@
          \else
            \DividE\ifdim\@xnum\p@<\z@-\fi\p@ by\@xnum\p@ to\sc@lelen
```

```
            \@linelen=\sc@lelen\@linelen
          \fi
        \fi
      \@tdB=\@linelen
\pIIe@concat\@xdim\@ydim{-\@ydim}\@xdim{\@xnum\@linelen}{\@ynum\@linelen}%
        \@linelen\z@
        \pIIe@vector
        \fillpath
      \@linelen=\@tdB
      \@tdA=\pIIe@FAW\@wholewidth
      \@tdA=\pIIe@FAL\@tdA
      \advance\@linelen-\@tdA
      \ifdim\@linelen>\z@
        \moveto(0,0)
        \pIIe@lineto{\@xnum\@linelen}{\@ynum\@linelen}%
        \strokepath\fi
    \endgroup}
\def\Vector(#1){{%
\GetCoord(#1)\@tX\@tY
\ifdim\@tX\p@=\z@\vector(\@tX,\@tY){\@tY}
\else
\vector(\@tX,\@tY){\@tX}\fi}}
\def\VECTOR(#1)(#2){\begingroup
\SubVect#1from#2to\@tempa
\expandafter\put\expandafter(#1){\expandafter\Vector\expandafter(\@tempa)}%
\endgroup\ignorespaces}
\let\lp@r( \let\rp@r)
\renewcommand*\polyline[1][\beveljoin]{\p@lylin@[#1]}

\def\p@lylin@[#1](#2){\@killglue#1\GetCoord(#2)\d@mX\d@mY
    \pIIe@moveto{\d@mX\unitlength}{\d@mY\unitlength}%
    \@ifnextchar\lp@r{\p@lyline}{%
    \PackageWarning{curve2e}%
    {Polylines require at least two vertices!\MessageBreak
     Control your polyline specification\MessageBreak}%
    \ignorespaces}}

\def\p@lyline(#1){\GetCoord(#1)\d@mX\d@mY
    \pIIe@lineto{\d@mX\unitlength}{\d@mY\unitlength}%
    \@ifnextchar\lp@r{\p@lyline}{\strokepath\ignorespaces}}
\providecommand\polygon{}
\RenewDocumentCommand\polygon{s O{\beveljoin} }{\@killglue\begingroup
\IfBooleanTF{#1}{\@tempswatrue}{\@tempswafalse}%
\@polygon[#2]}

\def\@polygon[#1](#2){\@killglue#1\GetCoord(#2)\d@mX\d@mY
    \pIIe@moveto{\d@mX\unitlength}{\d@mY\unitlength}%
    \@ifnextchar\lp@r{\@@polygon}{%
    \PackageWarning{curve2e}%
    {Polygons require at least two vertices!\MessageBreak
```

```
     Control your polygon specification\MessageBreak}%
    \ignorespaces}}

  \def\@@polygon(#1){\GetCoord(#1)\d@mX\d@mY
    \pIIe@lineto{\d@mX\unitlength}{\d@mY\unitlength}%
    \@ifnextchar\lp@r{\@@polygon}{\pIIe@closepath
      \if@tempswa\pIIe@fillGraph\else\pIIe@strokeGraph\fi
      \endgroup
      \ignorespaces}}
\def\GraphGrid(#1,#2){\bgroup\textcolor{red}{\linethickness{.1\p@}%
\RoundUp#1modulo10to\@GridWd \RoundUp#2modulo10to\@GridHt
\@tempcnta=\@GridWd \divide\@tempcnta10\relax \advance\@tempcnta\@ne
\multiput(0,0)(10,0){\@tempcnta}{\line(0,1){\@GridHt}}%
\@tempcnta=\@GridHt \divide\@tempcnta10\advance\@tempcnta\@ne
\multiput(0,0)(0,10){\@tempcnta}{\line(1,0){\@GridWd}}\thinlines}%
\egroup\ignorespaces}
\def\RoundUp#1modulo#2to#3{\expandafter\@tempcnta\Integer#1.??%
\count254\@tempcnta\divide\count254by#2\relax
\multiply\count254by#2\relax
\count252\@tempcnta\advance\count252-\count254
\ifnum\count252>0\advance\count252-#2\relax
\advance\@tempcnta-\count252\fi\edef#3{\number\@tempcnta}\ignorespaces}%
\def\Integer#1.#2??{#1}%
\ifdefined\dimexpr
    \unless\ifdefined\DividE
\def\DividE#1by#2to#3{\bgroup
\dimendef\Num2254\relax \dimendef\Den2252\relax
\dimendef\@DimA 2250
\Num=\p@ \Den=#2\relax
\ifdim\Den=\z@
  \edef\x{\noexpand\endgroup\noexpand\def\noexpand#3{\strip@pt\maxdimen}}%
\else
  \@DimA=#1\relax
  \edef\x{%
    \noexpand\egroup\noexpand\def\noexpand#3{%
       \strip@pt\dimexpr\@DimA*\Num/\Den\relax}}%
\fi
\x\ignorespaces}%
\fi
    \unless\ifdefined\DivideFN
         \def\DivideFN#1by#2to#3{\DividE#1\p@ by#2\p@ to{#3}}%
    \fi
    \unless\ifdefined\MultiplY
        \def\MultiplY#1by#2to#3{\bgroup
        \dimendef\@DimA 2254 \dimendef\@DimB2255
        \@DimA=#1\p@\relax \@DimB=#2\p@\relax
        \edef\x{%
           \noexpand\egroup\noexpand\def\noexpand#3{%
             \strip@pt\dimexpr\@DimA*\@DimB/\p@\relax}}%
        \x\ignorespaces}%
```

```
        \let\MultiplyFN\MultiplY
    \fi
\fi

\unless\ifdefined\Numero
  \def\Numero#1#2{\bgroup\dimen3254=#2\relax
    \edef\x{\noexpand\egroup\noexpand\edef\noexpand#1{%
      \strip@pt\dimen3254}}\x\ignorespaces}%
\fi
\def\g@tTanCotanFrom#1to#2and#3{%
\DividE 114.591559\p@ by#1to\X@ \@tdB=\X@\p@
\countdef\I=2546\def\Tan{0}\I=11\relax
\@whilenum\I>\z@\do{%
   \@tdC=\Tan\p@ \@tdD=\I\@tdB
   \advance\@tdD-\@tdC \DividE\p@ by\@tdD to\Tan
   \advance\I-2\relax}%
\def#2{\Tan}\DividE\p@ by\Tan\p@ to\Cot \def#3{\Cot}\ignorespaces}%
\def\SinOf#1to#2{\bgroup%
\@tdA=#1\p@%
\ifdim\@tdA>\z@%
  \@whiledim\@tdA>180\p@\do{\advance\@tdA -360\p@}%
\else%
  \@whiledim\@tdA<-180\p@\do{\advance\@tdA 360\p@}%
\fi \ifdim\@tdA=\z@
  \def\@tempA{0}%
\else
  \ifdim\@tdA>\z@
    \def\Segno{+}%
  \else
    \def\Segno{-}%
    \@tdA=-\@tdA
  \fi
  \ifdim\@tdA>90\p@
    \@tdA=-\@tdA \advance\@tdA 180\p@
  \fi
  \ifdim\@tdA=90\p@
    \def\@tempA{\Segno1}%
  \else
    \ifdim\@tdA=180\p@
      \def\@tempA{0}%
    \else
      \ifdim\@tdA<\p@
        \@tdA=\Segno0.0174533\@tdA
        \DividE\@tdA by\p@ to \@tempA%
      \else
        \g@tTanCotanFrom\@tdA to\T and\Tp
        \@tdA=\T\p@ \advance\@tdA \Tp\p@
        \DividE \Segno2\p@ by\@tdA to \@tempA%
      \fi
    \fi
```

```
  \fi
\fi
\edef\endSinOf{\noexpand\egroup
  \noexpand\def\noexpand#2{\@tempA}\noexpand\ignorespaces}%
\endSinOf}%
\def\CosOf#1to#2{\bgroup%
\@tdA=#1\p@%
\ifdim\@tdA>\z@%
  \@whiledim\@tdA>360\p@\do{\advance\@tdA -360\p@}%
\else%
  \@whiledim\@tdA<\z@\do{\advance\@tdA 360\p@}%
\fi
\ifdim\@tdA>180\p@
  \@tdA=-\@tdA \advance\@tdA 360\p@
\fi
\ifdim\@tdA<90\p@
  \def\Segno{+}%
\else
  \def\Segno{-}%
  \@tdA=-\@tdA \advance\@tdA 180\p@
\fi
\ifdim\@tdA=\z@
  \def\@tempA{\Segno1}%
\else
  \ifdim\@tdA<\p@
    \@tdA=0.0174533\@tdA \Numero\@tempA\@tdA
    \@tdA=\@tempA\@tdA \@tdA=-.5\@tdA
    \advance\@tdA \p@
    \DividE\@tdA by\p@ to\@tempA%
  \else
    \ifdim\@tdA=90\p@
      \def\@tempA{0}%
    \else
      \g@tTanCotanFrom\@tdA to\T and\Tp
      \@tdA=\Tp\p@ \advance\@tdA-\T\p@
      \@tdB=\Tp\p@ \advance\@tdB\T\p@
      \DividE\Segno\@tdA by\@tdB to\@tempA%
    \fi
  \fi
\fi
\edef\endCosOf{\noexpand\egroup
    \noexpand\def\noexpand#2{\@tempA}\noexpand\ignorespaces}%
\endCosOf}%
\def\TanOf#1to#2{\bgroup%
\@tdA=#1\p@%
\ifdim\@tdA>90\p@%
  \@whiledim\@tdA>90\p@\do{\advance\@tdA -180\p@}%
\else%
  \@whiledim\@tdA<-90\p@\do{\advance\@tdA 180\p@}%
\fi%
```

```
\ifdim\@tdA=\z@%
  \def\@tempA{0}%
\else
  \ifdim\@tdA>\z@
    \def\Segno{+}%
  \else
    \def\Segno{-}%
    \@tdA=-\@tdA
  \fi
  \ifdim\@tdA=90\p@
    \def\@tempA{\Segno16383.99999}%
  \else
    \ifdim\@tdA<\p@
        \@tdA=\Segno0.0174533\@tdA
        \DividE\@tdA by\p@ to\@tempA%
    \else
      \g@tTanCotanFrom\@tdA to\T and\Tp
      \@tdA\Tp\p@ \advance\@tdA -\T\p@
      \DividE\Segno2\p@ by\@tdA to\@tempA%
    \fi
  \fi
\fi
\edef\endTanOf{\noexpand\egroup
   \noexpand\def\noexpand#2{\@tempA}\noexpand\ignorespaces}%
\endTanOf}%
\def\ArcTanOf#1to#2{\bgroup
\countdef\Inverti 4444\Inverti=0
\def\Segno{}
\edef\@tF{#1}\@tdF=\@tF\p@ \@tdE=57.295778\p@
\@tdD=\ifdim\@tdF<\z@ -\@tdF\def\Segno{-}\else\@tdF\fi
\ifdim\@tdD>\p@
\Inverti=\@ne
\@tdD=\dimexpr\p@*\p@/\@tdD\relax
\fi
\unless\ifdim\@tdD>0.02\p@
    \def\@tX{\strip@pt\dimexpr57.295778\@tdD\relax}%
\else
    \edef\@tX{45}\relax
    \countdef\I 2523 \I=9\relax
    \@whilenum\I>0\do{\TanOf\@tX to\@tG
    \edef\@tG{\strip@pt\dimexpr\@tG\p@-\@tdD\relax}\relax
    \MultiplY\@tG by57.295778to\@tG
    \CosOf\@tX to\@tH
    \MultiplY\@tH by\@tH to\@tH
    \MultiplY\@tH by\@tG to \@tH
    \edef\@tX{\strip@pt\dimexpr\@tX\p@ - \@tH\p@\relax}\relax
    \advance\I\m@ne}%
\fi
\ifnum\Inverti=\@ne
\edef\@tX{\strip@pt\dimexpr90\p@-\@tX\p@\relax}
```

```
\fi
\edef\x{\egroup\noexpand\edef\noexpand#2{\Segno\@tX}}\x\ignorespaces}%
\def\MakeVectorFrom#1#2to#3{\edef#3{#1,#2}\ignorespaces}%
\def\CopyVect#1to#2{\edef#2{#1}\ignorespaces}%
\def\ModOfVect#1to#2{\GetCoord(#1)\t@X\t@Y
\@tempdima=\t@X\p@ \ifdim\@tempdima<\z@ \@tempdima=-\@tempdima\fi
\@tempdimb=\t@Y\p@ \ifdim\@tempdimb<\z@ \@tempdimb=-\@tempdimb\fi
\ifdim\@tempdima=\z@
    \ifdim\@tempdimb=\z@
        \def\@T{0}\@tempdimc=\z@
    \else
        \def\@T{0}\@tempdimc=\@tempdimb
    \fi
\else
    \ifdim\@tempdima>\@tempdimb
       \DividE\@tempdimb by\@tempdima to\@T
       \@tempdimc=\@tempdima
    \else
          \DividE\@tempdima by\@tempdimb to\@T
          \@tempdimc=\@tempdimb
    \fi
\fi
\unless\ifdim\@tempdimc=\z@
    \unless\ifdim\@T\p@=\z@
       \@tempdima=\@T\p@ \@tempdima=\@T\@tempdima
       \advance\@tempdima\p@%
       \@tempdimb=\p@%
       \@tempcnta=5\relax
       \@whilenum\@tempcnta>\z@\do{\DividE\@tempdima by\@tempdimb to\@T
       \advance\@tempdimb \@T\p@ \@tempdimb=.5\@tempdimb
       \advance\@tempcnta\m@ne}%
       \@tempdimc=\@T\@tempdimc
    \fi
\fi
\Numero#2\@tempdimc
\ignorespaces}%
\def\DirOfVect#1to#2{\GetCoord(#1)\t@X\t@Y
\ModOfVect#1to\@tempa
\unless\ifdim\@tempdimc=\z@
   \DividE\t@X\p@ by\@tempdimc to\t@X
   \DividE\t@Y\p@ by\@tempdimc to\t@Y
\fi
\MakeVectorFrom\t@X\t@Y to#2\ignorespaces}%
\def\ModAndDirOfVect#1to#2and#3{%
\GetCoord(#1)\t@X\t@Y
\ModOfVect#1to#2%
\ifdim\@tempdimc=\z@\else
  \DividE\t@X\p@ by\@tempdimc to\t@X
  \DividE\t@Y\p@ by\@tempdimc to\t@Y
\fi
```

```
\MakeVectorFrom\t@X\t@Y to#3\ignorespaces}%
\def\DistanceAndDirOfVect#1minus#2to#3and#4{%
\SubVect#2from#1to\@tempa
\ModAndDirOfVect\@tempa to#3and#4\ignorespaces}%
\def\XpartOfVect#1to#2{%
\GetCoord(#1)#2\@tempa\ignorespaces}%
\def\YpartOfVect#1to#2{%
\GetCoord(#1)\@tempa#2\ignorespaces}%
\def\DirFromAngle#1to#2{%
\CosOf#1to\t@X
\SinOf#1to\t@Y
\MakeVectorFrom\t@X\t@Y to#2\ignorespaces}%
\def\ArgOfVect#1to#2{\bgroup\GetCoord(#1){\t@X}{\t@Y}%
\def\s@gno{}\def\addflatt@ngle{0}
\ifdim\t@X\p@=\z@
  \ifdim\t@Y\p@=\z@
    \def\ArcTan{0}%
  \else
    \def\ArcTan{90}%
    \ifdim\t@Y\p@<\z@\def\s@gno{-}\fi
  \fi
\else
  \ifdim\t@Y\p@=\z@
    \ifdim\t@X\p@<\z@
      \def\ArcTan{180}%
    \else
      \def\ArcTan{0}%
    \fi
  \else
    \ifdim\t@X\p@<\z@%
      \def\addflatt@ngle{180}%
      \edef\t@X{\strip@pt\dimexpr-\t@X\p@}%
      \edef\t@Y{\strip@pt\dimexpr-\t@Y\p@}%
      \ifdim\t@Y\p@<\z@
        \def\s@gno{-}%
        \edef\t@Y{-\t@Y}%
      \fi
    \fi
  \DivideFN\t@Y by\t@X to \t@A
  \ArcTanOf\t@A to\ArcTan
  \fi
\fi
\edef\ArcTan{\unless\ifx\s@gno\empty\s@gno\fi\ArcTan}%
\unless\ifnum\addflatt@ngle=0\relax
   \edef\ArcTan{%
   \strip@pt\dimexpr\ArcTan\p@\ifx\s@gno\empty-\else+\fi
   \addflatt@ngle\p@\relax}%
\fi
\edef\x{\noexpand\egroup\noexpand\edef\noexpand#2{\ArcTan}}%
\x\ignorespaces}
```

```
\def\ScaleVect#1by#2to#3{\GetCoord(#1)\t@X\t@Y
\@tempdima=\t@X\p@ \@tempdima=#2\@tempdima\Numero\t@X\@tempdima
\@tempdima=\t@Y\p@ \@tempdima=#2\@tempdima\Numero\t@Y\@tempdima
\MakeVectorFrom\t@X\t@Y to#3\ignorespaces}%
\def\ConjVect#1to#2{\GetCoord(#1)\t@X\t@Y
\@tempdima=-\t@Y\p@\Numero\t@Y\@tempdima
\MakeVectorFrom\t@X\t@Y to#2\ignorespaces}%
\def\AddVect#1and#2to#3{\GetCoord(#1)\tu@X\tu@Y
\GetCoord(#2)\td@X\td@Y
\@tempdima\tu@X\p@\advance\@tempdima\td@X\p@ \Numero\t@X\@tempdima
\@tempdima\tu@Y\p@\advance\@tempdima\td@Y\p@ \Numero\t@Y\@tempdima
\MakeVectorFrom\t@X\t@Y to#3\ignorespaces}%
\def\SubVect#1from#2to#3{\GetCoord(#1)\tu@X\tu@Y
\GetCoord(#2)\td@X\td@Y
\@tempdima\td@X\p@\advance\@tempdima-\tu@X\p@ \Numero\t@X\@tempdima
\@tempdima\td@Y\p@\advance\@tempdima-\tu@Y\p@ \Numero\t@Y\@tempdima
\MakeVectorFrom\t@X\t@Y to#3\ignorespaces}%
\def\MultVect#1by{\@ifstar{\@ConjMultVect#1by}{\@MultVect#1by}}%
\def\@MultVect#1by#2to#3{\GetCoord(#1)\tu@X\tu@Y
\GetCoord(#2)\td@X\td@Y
\@tempdima\tu@X\p@ \@tempdimb\tu@Y\p@
\@tempdimc=\td@X\@tempdima\advance\@tempdimc-\td@Y\@tempdimb
\Numero\t@X\@tempdimc
\@tempdimc=\td@Y\@tempdima\advance\@tempdimc\td@X\@tempdimb
\Numero\t@Y\@tempdimc
\MakeVectorFrom\t@X\t@Y to#3\ignorespaces}%
\def\@ConjMultVect#1by#2to#3{\GetCoord(#1)\tu@X\tu@Y
\GetCoord(#2)\td@X\td@Y \@tempdima\tu@X\p@ \@tempdimb\tu@Y\p@
\@tempdimc=\td@X\@tempdima\advance\@tempdimc+\td@Y\@tempdimb
\Numero\t@X\@tempdimc
\@tempdimc=\td@X\@tempdimb\advance\@tempdimc-\td@Y\@tempdima
\Numero\t@Y\@tempdimc
\MakeVectorFrom\t@X\t@Y to#3\ignorespaces}
\def\DivVect#1by#2to#3{\ModAndDirOfVect#2to\@Mod and\@Dir
\DividE\p@ by\@Mod\p@ to\@Mod \ConjVect\@Dir to\@Dir
\ScaleVect#1by\@Mod to\@tempa
\MultVect\@tempa by\@Dir to#3\ignorespaces}%
\def\Arc(#1)(#2)#3{\begingroup
\@tdA=#3\p@
\unless\ifdim\@tdA=\z@
  \@Arc(#1)(#2)%
\fi
\endgroup\ignorespaces}%
\def\@Arc(#1)(#2){%
\ifdim\@tdA>\z@
  \let\Segno+%
\else
  \@tdA=-\@tdA \let\Segno-%
\fi
\Numero\@gradi\@tdA
```

```
\ifdim\@tdA>360\p@
 \PackageWarning{curve2e}{The arc aperture is \@gradi\space degrees
      and gets reduced\MessageBreak%
      to the range 0--360 taking the sign into consideration}%
 \@whiledim\@tdA>360\p@\do{\advance\@tdA-360\p@}%
\fi
\SubVect#2from#1to\@V \ModOfVect\@V to\@Raggio \CopyVect#2to\@pPun
\CopyVect#1to\@Cent \GetCoord(\@pPun)\@pPunX\@pPunY
\@@Arc
\strokepath\ignorespaces}%
\def\@@Arc{%
\pIIe@moveto{\@pPunX\unitlength}{\@pPunY\unitlength}%
\ifdim\@tdA>180\p@
  \advance\@tdA-180\p@
  \Numero\@gradi\@tdA
  \SubVect\@pPun from\@Cent to\@V
  \AddVect\@V and\@Cent to\@sPun
  \MultVect\@V by0,-1.3333333to\@V \if\Segno-\ScaleVect\@V by-1to\@V\fi
  \AddVect\@pPun and\@V to\@pcPun
  \AddVect\@sPun and\@V to\@scPun
  \GetCoord(\@pcPun)\@pcPunX\@pcPunY
  \GetCoord(\@scPun)\@scPunX\@scPunY
  \GetCoord(\@sPun)\@sPunX\@sPunY
  \pIIe@curveto{\@pcPunX\unitlength}{\@pcPunY\unitlength}%
               {\@scPunX\unitlength}{\@scPunY\unitlength}%
               {\@sPunX\unitlength}{\@sPunY\unitlength}%
  \CopyVect\@sPun to\@pPun
\fi
\ifdim\@tdA>\z@
  \DirFromAngle\@gradi to\@Dir \if\Segno-\ConjVect\@Dir to\@Dir \fi
  \SubVect\@Cent from\@pPun to\@V
  \MultVect\@V by\@Dir to\@V
  \AddVect\@Cent and\@V to\@sPun
  \@tdA=.5\@tdA \Numero\@gradi\@tdA
  \DirFromAngle\@gradi to\@Phimezzi
  \GetCoord(\@Phimezzi)\@cosphimezzi\@sinphimezzi
  \@tdB=1.3333333\p@ \@tdB=\@Raggio\@tdB
  \@tdC=\p@ \advance\@tdC -\@cosphimezzi\p@ \Numero\@tempa\@tdC
  \@tdB=\@tempa\@tdB
  \DividE\@tdB by\@sinphimezzi\p@ to\@cZ
  \ScaleVect\@Phimezzi by\@cZ to\@Phimezzi
  \ConjVect\@Phimezzi to\@mPhimezzi
  \if\Segno-%
    \let\@tempa\@Phimezzi
    \let\@Phimezzi\@mPhimezzi
    \let\@mPhimezzi\@tempa
  \fi
  \SubVect\@sPun from\@pPun to\@V
  \DirOfVect\@V to\@V
  \MultVect\@Phimezzi by\@V to\@Phimezzi
```

```
  \AddVect\@sPun and\@Phimezzi to\@scPun
  \ScaleVect\@V by-1to\@V
  \MultVect\@mPhimezzi by\@V to\@mPhimezzi
  \AddVect\@pPun and\@mPhimezzi to\@pcPun
  \GetCoord(\@pcPun)\@pcPunX\@pcPunY
  \GetCoord(\@scPun)\@scPunX\@scPunY
  \GetCoord(\@sPun)\@sPunX\@sPunY
  \pIIe@curveto{\@pcPunX\unitlength}{\@pcPunY\unitlength}%
               {\@scPunX\unitlength}{\@scPunY\unitlength}%
               {\@sPunX\unitlength}{\@sPunY\unitlength}%
\fi}
\def\VectorArc(#1)(#2)#3{\begingroup
\@tdA=#3\p@ \ifdim\@tdA=\z@\else
  \@VArc(#1)(#2)%
\fi
\endgroup\ignorespaces}%
\def\VectorARC(#1)(#2)#3{\begingroup
\@tdA=#3\p@
\ifdim\@tdA=\z@\else
  \@VARC(#1)(#2)%
\fi
\endgroup\ignorespaces}%
\def\@VArc(#1)(#2){%
\ifdim\@tdA>\z@
  \let\Segno+%
\else
  \@tdA=-\@tdA \let\Segno-%
\fi \Numero\@gradi\@tdA
\ifdim\@tdA>360\p@
 \PackageWarning{curve2e}{The arc aperture is \@gradi\space degrees
      and gets reduced\MessageBreak%
      to the range 0--360 taking the sign into consideration}%
 \@whiledim\@tdA>360\p@\do{\advance\@tdA-360\p@}%
\fi
\SubVect#1from#2to\@V \ModOfVect\@V to\@Raggio \CopyVect#2to\@pPun
\@tdE=\pIIe@FAW\@wholewidth \@tdE=\pIIe@FAL\@tdE
\DividE\@tdE by \@Raggio\unitlength to\DeltaGradi
\@tdD=\DeltaGradi\p@
\@tdD=57.29578\@tdD \Numero\DeltaGradi\@tdD
\@tdD=\ifx\Segno--\fi\@gradi\p@ \Numero\@tempa\@tdD
\DirFromAngle\@tempa to\@Dir
\MultVect\@V by\@Dir to\@sPun
\edef\@tempA{\ifx\Segno-\m@ne\else\@ne\fi}%
\MultVect\@sPun by 0,\@tempA to\@vPun
\DirOfVect\@vPun to\@Dir
\AddVect\@sPun and #1 to \@sPun
\GetCoord(\@sPun)\@tdX\@tdY
\@tdD\ifx\Segno--\fi\DeltaGradi\p@
\@tdD=.5\@tdD \Numero\DeltaGradi\@tdD
\DirFromAngle\DeltaGradi to\@Dird
```

```
\MultVect\@Dir by*\@Dird to\@Dir
\GetCoord(\@Dir)\@xnum\@ynum
\put(\@tdX,\@tdY){\vector(\@xnum,\@ynum){0}}%
\@tdE =\ifx\Segno--\fi\DeltaGradi\p@
\advance\@tdA -\@tdE \Numero\@gradi\@tdA
\CopyVect#1to\@Cent \GetCoord(\@pPun)\@pPunX\@pPunY
\@@Arc
\strokepath\ignorespaces}%
\def\@VARC(#1)(#2){%
\ifdim\@tdA>\z@
  \let\Segno+%
\else
  \@tdA=-\@tdA \let\Segno-%
\fi \Numero\@gradi\@tdA
\ifdim\@tdA>360\p@
 \PackageWarning{curve2e}{The arc aperture is \@gradi\space degrees
      and gets reduced\MessageBreak%
      to the range 0--360 taking the sign into consideration}%
 \@whiledim\@tdA>360\p@\do{\advance\@tdA-360\p@}%
\fi
\SubVect#1from#2to\@V \ModOfVect\@V to\@Raggio \CopyVect#2to\@pPun
\@tdE=\pIIe@FAW\@wholewidth \@tdE=0.8\@tdE
\DividE\@tdE by \@Raggio\unitlength to\DeltaGradi
\@tdD=\DeltaGradi\p@ \@tdD=57.29578\@tdD \Numero\DeltaGradi\@tdD
\@tdD=\if\Segno--\fi\@gradi\p@ \Numero\@tempa\@tdD
\DirFromAngle\@tempa to\@Dir
\MultVect\@V by\@Dir to\@sPun% corrects the end point
\edef\@tempA{\if\Segno--\fi1}%
\MultVect\@sPun by 0,\@tempA to\@vPun
\DirOfVect\@vPun to\@Dir
\AddVect\@sPun and #1 to \@sPun
\GetCoord(\@sPun)\@tdX\@tdY
\@tdD\if\Segno--\fi\DeltaGradi\p@
\@tdD=.5\@tdD \Numero\@tempB\@tdD
\DirFromAngle\@tempB to\@Dird
\MultVect\@Dir by*\@Dird to\@Dir
\GetCoord(\@Dir)\@xnum\@ynum
\put(\@tdX,\@tdY){\vector(\@xnum,\@ynum){0}}% end point arrowt ip
\@tdE =\DeltaGradi\p@
\advance\@tdA -2\@tdE \Numero\@gradi\@tdA
\CopyVect#1to\@Cent \GetCoord(\@pPun)\@pPunX\@pPunY
\SubVect\@Cent from\@pPun to \@V
\edef\@tempa{\if\Segno-\else-\fi\@ne}%
\MultVect\@V by0,\@tempa to\@vPun
\@tdE\if\Segno--\fi\DeltaGradi\p@
\Numero\@tempB{0.5\@tdE}%
\DirFromAngle\@tempB to\@Dird
\MultVect\@vPun by\@Dird to\@vPun% corrects the starting point
\DirOfVect\@vPun to\@Dir\GetCoord(\@Dir)\@xnum\@ynum
\put(\@pPunX,\@pPunY){\vector(\@xnum,\@ynum){0}}% starting point arrow tip
```

```
\edef\@tempa{\if\Segno--\fi\DeltaGradi}%
\DirFromAngle\@tempa to \@Dir
\SubVect\@Cent from\@pPun to\@V
\MultVect\@V by\@Dir to\@V
\AddVect\@Cent and\@V to\@pPun
\GetCoord(\@pPun)\@pPunX\@pPunY
\@@Arc
\strokepath\ignorespaces}%
\def\CurveBetween#1and#2WithDirs#3and#4{%
\StartCurveAt#1WithDir{#3}\relax
\CurveTo#2WithDir{#4}\CurveFinish\ignorespaces}%
\def\StartCurveAt#1WithDir#2{%
\begingroup
\GetCoord(#1)\@tempa\@tempb
\CopyVect\@tempa,\@tempb to\@Pzero
\pIIe@moveto{\@tempa\unitlength}{\@tempb\unitlength}%
\GetCoord(#2)\@tempa\@tempb
\CopyVect\@tempa,\@tempb to\@Dzero
\DirOfVect\@Dzero to\@Dzero
\ignorespaces}
\def\ChangeDir<#1>{%
\GetCoord(#1)\@tempa\@tempb
\CopyVect\@tempa,\@tempb to\@Dzero
\DirOfVect\@Dzero to\@Dzero
\ignorespaces}
\def\CurveFinish{\strokepath\endgroup\ignorespaces}%
\def\FillCurve{\fillpath\endgroup\ignorespaces}
\def\CurveEnd{\fillstroke\endgroup\ignorespaces}
\def\CbezierTo#1WithDir#2AndDists#3And#4{%
\GetCoord(#1)\@tX\@tY \MakeVectorFrom\@tX\@tY to\@Puno
\GetCoord(#2)\@tX\@tY \MakeVectorFrom\@tX\@tY to \@Duno
\DirOfVect\@Duno to\@Duno
\ScaleVect\@Dzero by#3to\@Czero \AddVect\@Pzero and\@Czero to\@Czero
\ScaleVect\@Duno by-#4to \@Cuno \AddVect\@Puno and\@Cuno to \@Cuno
\GetCoord(\@Czero)\@XCzero\@YCzero
\GetCoord(\@Cuno)\@XCuno\@YCuno
\GetCoord(\@Puno)\@XPuno\@YPuno
\pIIe@curveto{\@XCzero\unitlength}{\@YCzero\unitlength}%
             {\@XCuno\unitlength}{\@YCuno\unitlength}%
             {\@XPuno\unitlength}{\@YPuno\unitlength}%
\CopyVect\@Puno to\@Pzero
\CopyVect\@Duno to\@Dzero
\ignorespaces}%
\def\CbezierBetween#1And#2WithDirs#3And#4UsingDists#5And#6{%
\StartCurveAt#1WithDir{#3}\relax
\CbezierTo#2WithDir#4AndDists#5And{#6}\CurveFinish}

\def\@isTension#1;#2!!{\def\@tempA{#1}%
\def\@tempB{#2}\unless\ifx\@tempB\empty\strip@semicolon#2\fi}
\def\strip@semicolon#1;{\def\@tempB{#1}}
```

```
\def\CurveTo#1WithDir#2{%
\def\@Tuno{1}\def\@Tzero{1}\relax
\edef\@Puno{#1}\@isTension#2;!!%
\expandafter\DirOfVect\@tempA to\@Duno
\bgroup\unless\ifx\@tempB\empty\GetCoord(\@tempB)\@Tzero\@Tuno\fi
\DistanceAndDirOfVect\@Puno minus\@Pzero to\@Chord and\@DirChord
\MultVect\@Dzero by*\@DirChord to \@Dpzero
\MultVect\@Duno by*\@DirChord to \@Dpuno
\GetCoord(\@Dpzero)\@DXpzero\@DYpzero
\GetCoord(\@Dpuno)\@DXpuno\@DYpuno
\DivideFN\@Chord by2 to\@semichord
\ifdim\@DXpzero\p@=\z@
   \@tdA=1.333333\p@
   \Numero\@KCzero{\@semichord\@tdA}%
\fi
\ifdim\@DYpzero\p@=\z@
  \@tdA=1.333333\p@
  \Numero\@Kpzero{\@semichord\@tdA}%
\fi
\unless\ifdim\@DXpzero\p@=\z@
  \unless\ifdim\@DYpzero\p@=\z@
    \edef\@CosDzero{\ifdim\@DXpzero\p@<\z@ -\fi\@DXpzero}%
    \edef\@SinDzero{\ifdim\@DYpzero\p@<\z@ -\fi\@DYpzero}%
    \@tdA=\@semichord\p@ \@tdA=1.333333\@tdA
    \DividE\@tdA by\@SinDzero\p@ to \@KCzero
    \@tdA=\dimexpr(\p@-\@CosDzero\p@)\relax
    \DividE\@KCzero\@tdA by\@SinDzero\p@ to \@KCzero
  \fi
\fi
\MultiplyFN\@KCzero by \@Tzero to \@KCzero
\ScaleVect\@Dzero by\@KCzero to\@CPzero
\AddVect\@Pzero and\@CPzero to\@CPzero
\ifdim\@DXpuno\p@=\z@
   \@tdA=-1.333333\p@
   \Numero\@KCuno{\@semichord\@tdA}%
\fi
\ifdim\@DYpuno\p@=\z@
  \@tdA=-1.333333\p@
  \Numero\@KCuno{\@semichord\@tdA}%
\fi
\unless\ifdim\@DXpuno\p@=\z@
    \unless\ifdim\@DYpuno\p@=\z@
    \edef\@CosDuno{\ifdim\@DXpuno\p@<\z@ -\fi\@DXpuno}%
    \edef\@SinDuno{\ifdim\@DYpuno\p@<\z@ -\fi\@DYpuno}%
    \@tdA=\@semichord\p@ \@tdA=-1.333333\@tdA
    \DividE\@tdA by \@SinDuno\p@ to \@KCuno
    \@tdA=\dimexpr(\p@-\@CosDuno\p@)\relax
    \DividE\@KCuno\@tdA by\@SinDuno\p@ to \@KCuno
  \fi
\fi
```

```
\MultiplyFN\@KCuno by \@Tuno to \@KCuno
\ScaleVect\@Duno by\@KCuno to\@CPuno
\AddVect\@Puno and\@CPuno to\@CPuno
\GetCoord(\@Puno)\@XPuno\@YPuno
\GetCoord(\@CPzero)\@XCPzero\@YCPzero
\GetCoord(\@CPuno)\@XCPuno\@YCPuno
\pIIe@curveto{\@XCPzero\unitlength}{\@YCPzero\unitlength}%
             {\@XCPuno\unitlength}{\@YCPuno\unitlength}%
             {\@XPuno\unitlength}{\@YPuno\unitlength}\egroup
\CopyVect\@Puno to\@Pzero
\CopyVect\@Duno to\@Dzero
\ignorespaces}%
\def\Curve{\@ifstar{\let\fillstroke\fillpath\Curve@}%
{\let\fillstroke\strokepath\Curve@}}
\def\Curve@(#1)<#2>{%
    \StartCurveAt#1WithDir{#2}%
    \@ifnextchar\lp@r\@Curve{%
    \PackageWarning{curve2e}{%
    Curve specifications must contain at least two nodes!\Messagebreak
    Please, control your Curve specifications\MessageBreak}}}
\def\@Curve(#1)<#2>{%
    \CurveTo#1WithDir{#2}%
    \@ifnextchar\lp@r\@Curve{%
    \@ifnextchar[\@ChangeDir\CurveEnd}}
\def\@ChangeDir[#1]{\ChangeDir<#1>\@Curve}
\def\Qurve{\@ifstar{\let\fillstroke\fillpath\Qurve@}%
{\let\fillstroke\strokepath\Qurve@}}

\def\Qurve@(#1)<#2>{%
    \StartCurveAt#1WithDir{#2}%
    \@ifnextchar\lp@r\@Qurve{%
    \PackageWarning{curve2e}{%
      Quadratic curve specifications must contain at least
      two nodes!\Messagebreak
      Please, control your Qurve specifications\MessageBreak}}}%
\def\@Qurve(#1)<#2>{\QurveTo#1WithDir{#2}%
    \@ifnextchar\lp@r\@Qurve{%
    \@ifnextchar[\@ChangeQDir\CurveEnd}}%
\def\@ChangeQDir[#1]{\ChangeDir<#1>\@Qurve}%
\def\QurveTo#1WithDir#2{%
\edef\@Puno{#1}\DirOfVect#2to\@Duno\bgroup
\DistanceAndDirOfVect\@Puno minus\@Pzero to\@Chord and\@DirChord
\MultVect\@Dzero by*\@Duno to \@Scalar
\YpartOfVect\@Scalar to \@YScalar
\ifdim\@YScalar\p@=\z@
\PackageWarning{curve2e}%
   {Quadratic Bezier arcs cannot have their starting\MessageBreak
   and ending directions parallel or antiparallel with\MessageBreak
   each other. This arc is skipped and replaced with
   a dotted line.\MessageBreak}%
```

```
   \Dotline(\@Pzero)(\@Puno){2}\relax
\else
\MultVect\@Dzero by*\@DirChord to \@Dpzero
\MultVect\@Duno by*\@DirChord to \@Dpuno
\GetCoord(\@Dpzero)\@DXpzero\@DYpzero
\GetCoord(\@Dpuno)\@DXpuno\@DYpuno
\MultiplyFN\@DXpzero by\@DXpuno to\@XXD
\MultiplyFN\@DYpzero by\@DYpuno to\@YYD
\unless\ifdim\@YYD\p@<\z@\ifdim\@XXD\p@<\z@
\PackageWarning{curve2e}%
   {Quadratic Bezier arcs cannot have inflection points\MessageBreak
   Therefore the tangents to the starting and ending arc\MessageBreak
   points cannot be directed to the same half plane.\MessageBreak
   This arc is skipped and replaced by a dotted line\MessageBreak}%
   \Dotline(\@Pzero)(\@Puno){2}\fi
\else
\edef\@CDzero{\@DXpzero}\relax
\edef\@SDzero{\@DYpzero}\relax
\edef\@CDuno{\@DXpuno}\relax
\edef\@SDuno{\@DYpuno}\relax
\MultiplyFN\@SDzero by\@CDuno to\@tempA
\MultiplyFN\@SDuno by\@CDzero to\@tempB
\edef\@tempA{\strip@pt\dimexpr\@tempA\p@-\@tempB\p@}\relax
\@tdA=\@SDuno\p@ \@tdB=\@Chord\p@ \@tdC=\@tempA\p@
\edef\@tempC{\strip@pt\dimexpr \@tdA*\@tdB/\@tdC}\relax
\MultiplyFN\@tempC by\@CDzero to \@XC
\MultiplyFN\@tempC by\@SDzero to \@YC
\ModOfVect\@XC,\@YC to\@KC
\ScaleVect\@Dzero by\@KC to\@CP
\AddVect\@Pzero and\@CP to\@CP
\GetCoord(\@Pzero)\@XPzero\@YPzero
\GetCoord(\@Puno)\@XPuno\@YPuno
\GetCoord(\@CP)\@XCP\@YCP
\@ovxx=\@XPzero\unitlength \@ovyy=\@YPzero\unitlength
\@ovdx=\@XCP\unitlength \@ovdy=\@YCP\unitlength
\@xdim=\@XPuno\unitlength \@ydim=\@YPuno\unitlength
      \pIIe@bezier@QtoC\@ovxx\@ovdx\@ovro
      \pIIe@bezier@QtoC\@ovyy\@ovdy\@ovri
      \pIIe@bezier@QtoC\@xdim\@ovdx\@clnwd
      \pIIe@bezier@QtoC\@ydim\@ovdy\@clnht
      \pIIe@moveto\@ovxx\@ovyy
      \pIIe@curveto\@ovro\@ovri\@clnwd\@clnht\@xdim\@ydim
\fi\fi\egroup
\CopyVect\@Puno to\@Pzero
\CopyVect\@Duno to\@Dzero
\ignorespaces}
```

References

[1] Gäßlein H., Niepraschk R., and Tkadlec J. *The `pict2e` package*, 2019, PDF documentation of `pict2e`; this package is part of any modern complete distribution of the TEX system; it may be read by means of the line command `texdoc pict2e`. In case of a basic or partial system installation, the package may be installed by means of the specific facilities of the distribution.