The extension package curve2e

Claudio Beccari

Version number v.1.50; last revised 2015/06/19.

Contents

Abstract

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

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

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

1 Package pict2e and this extension curve2e

Package pict2e was announced in issue 15 of latexnews around December 2003; it was declared that the new package would replace the dummy one that has been

accompanying every release of LaTeX 2$_\varepsilon$ 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; 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 adds the following features.

1. Most if not all coordinate pairs and slope pairs are treated as *ordered pairs*, that is *complex numbers*; in practice the user does not notice any difference from what he/she was used to, but all the mathematical treatment to be applied to these entities is coded as complex number operations, since complex numbers may be viewed non only as ordered pairs, but also as vectors or roto-amplification operators.

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

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

4. The `\line` macro is redefined so as to allow integer and fractional direction coefficients, but maintaining the same syntax as in the original `picture` environment (now available also with `pict2e`).

5. A new macro `\Line` was defined so as to avoid the need to specify the horizontal projection of inclined lines (now available also with `pict2e`); this macro name now conflicts with `pict2e` 2009 version; therefore its name is changed to `\LIne` and supposedly it will not be used very often, if ever, by the end user (but it is used within this package macros).

6. A new macro `\LINE` was defined in order to join two points specified with their coordinates; this is now the normal behavior of the `\Line` macro of `pict2e` so that `\LINE` is now renamed `\segment`; there is no need to use the `\put` command with this line specification.

7. A new macro `\DLine` is defined in order to draw dashed lines joining any two given points; the dash length and gap (equal to one another) get specified through one of the macro arguments.

8. A new macro `\Dotline` is defined in order to draw dotted straight lines as a sequence of equally spaced dots, where the gap can be specified by the user; such straight line may have any inclination, as well as the above dashed lines.

9. 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.

10. 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.

11. 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. It must be noticed that the syntax is slightly different, so that it's reasonable that these commands, in spite of producing identical arcs, might be more comfortable with this or that syntax.

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

13. 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 currently specified color.

14. `\Curve` is a recursive macro that can draw an unlimited (reasonably low) number of connecter Bézier spline arcs with continuos tangents except for cusps; these arcs require only the specification of te 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.

15. Last but not least, all these commands accept polar coordinates or cartesian ones at the choice of the user who may use for each object the formalism he/she prefers. Also the `put` and `\multiput` commands have been redefined so as to accept the cartesian or the polar coordinates.

16. The basic macros used within the cumulative `\Curve` macro can be used individually in order to draw any curve, one cubic arc at the time; but they are intended for internal use, even if it is not prohibited to use them; by themselves such arcs are not different form those used by `Curve`, but the final command, `\FillCurve`, should be used in place of `\CurveFinish`, so as to fill up the closed path with the locally specified color; see figure 5. It is much more convenient to use the starred version of `\Curve` macro.

The `pict2e` package already defines macros such as `\moveto`, `\lineto`, `\curveto`, `\closepath`, `\fillpath`, and `\strokepath`; of course these macros can be used by the end user, and sometimes they perform better than the macros defined in this package, because the user has a better control on the position of the Bézier control points, while here the control points are sort of rigid. It would be very useful to resort to the `hobby` package, but its macros are conforming with those of the `tikz` and `pgf` packages, not with `curve2e`; an interface should be created in order to deal with the `hobby` package, but this has not yet been done.

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. The trigonometric functions have also been defined in a way that the author believes to be more efficient than those defined by the `trig` package; in any case the macro names are sufficiently different to accommodate both definition sets in the same LaTeX run.

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

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

2 Summary of modifications and new commands

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

1. This package `curve2e` calls directly the LATEX packages `color` and `pict2e` to which it passes any possible option that the latter can receive; actually the only options that make sense are those concerning the arrow tips, either LATEX or PostScript styled, because it is assumed that if you use this package you are not interested in using the original LATEX commands. See the `pict2e` documentation in order to use the correct options `pict2e` can receive.

2. The user is offered new commands in order to control the line terminators and the line joins; specifically:

   - `\roundcap`: the line is terminated with a semicircle;
   - `\squarecap`: the line is terminated with a half square;
   - `\roundjoin`: two lines are joined with a rounded join;
   - `\beveljoin`: two lines are joined with a bevel join;
   - `\miterjoin`: two lines are joined with a miter join.

   All the above commands should respect the intended range; but since they act at the PostScript or PDF level, not at TEX level, it might be necessary to issue the necessary command in order to restore the previous terminator or join.

3. The commands `\linethickness`, `\thicklines`, `\thinlines` together with `\defaultlinethickness` always redefine the internal `\@wholewidth` and `\@halfwidth` so that the latter always refer to a full width and to a half of it in this way: if you issue the command `\defaultlinewidth{2pt}` all thin lines will be drawn with a thickness of 1 pt while if a drawing command directly refers to the internal value `\@wholewidth`, its line will be

drawn with a thickness of 2 pt. If one issues the declaration `\thinlines` all lines will be drawn with a 1 pt width, but if a command refers to the internal value `\@halfwidth` the line will be drawn with a thickness of 0.5 pt. The command `\linethickness` redefines the above internals but does not change the default width value; all these width specifications apply to all lines, straight ones, curved ones, circles, ovals, vectors, dashed, et cetera. It's better to recall that `thinlines` and `thicklines` are declarations that do not take arguments; on the opposite the other two commands follow the standard syntax:

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

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

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

5. New line and vector macros are defined that avoid the necessity of specifying the horizontal component; `\put(3,4){\LIne(25,15)}` specifies a segment that starts at point $(3, 4)$ and goes to point $(3 + 25, 4 + 15)$; the command `\segment(3,4)(28,19)` achieves the same result without the need of using command `\put`. The same applies to the vector commands `\Vector` and `\VECTOR`. Experience has shown that the commands intended to joint two specified coordinates are particularly useful.

6. The `\polyline` command has been introduced: it accepts an unlimited list of point coordinates enclosed within round parentheses; the command draws a sequence of connected segments that joins in order the specified points; the syntax is:

   `\polyline[`{⟨*optional join style*⟩}`]` `(`⟨P_1⟩`)(`⟨P_2⟩`)...(`⟨P_n⟩`)`

   See figure 1 where a regular pentagon is drawn; usage of polar coordinates is also shown.

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

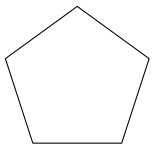

Figure 1: Polygonal line obtained by means of the `\polyline` command; coordinates are in polar form.

Although you can draw polygons with `\polyline`, as it was done in figure 1, do not confuse this command with the command `\polygon` defined in `pict2e`

2009; the latter automatically joins the last specified coordinate to the first one with a straight line, therefore closing the path. `pict2e` defines also the starred command that fills up the inside of the generated polygon.

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

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

   draws a dashed line containing as many dashes as possible, long as specified, and separated by a gap exactly the same size; actually, in order to make an even gap-dash sequence, the desired dash length is used to do some computations in order to find a suitable length, close to the one specified, such that the distance of the end points is evenly divided in equally sized dashes and gaps. The end points may be anywhere in the drawing area, without any constraint on the slope of the joining segment. The desired dash length is specified as a fractional multiple of `\unitlength`; see figure 2.

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

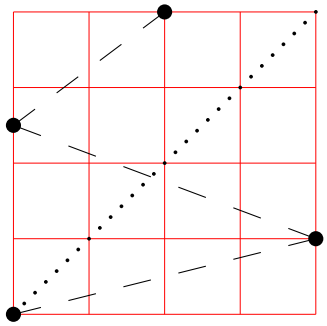

Figure 2: Dashed lines and graph grid

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

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

   See figures 2 and 7 for examples.

9. `\GraphGrid` is a command that draws a red grid under the drawing with lines separated `10\unitlength`s apart; it is described only with a comma separated couple of numbers, representing the base and the height of the grid, see figure 2; it's better to specify multiples of ten and the grid can be placed anywhere in the drawing canvas by means of `\put`, whose cartesian coordinates are multiples of 10; nevertheless the grid line distance is rounded to the nearest multiple of 10, while the point coordinates specified to `\put` are not rounded at all; therefore some care should be used to place the working grid in the drawing canvas. This grid is intended as an aid while

drawing; even if you sketch your drawing on millimetre paper, the drawing grid turns out to be very useful; one must only delete or comment out the command when the drawing is finished.

10. New trigonometric function macros have been implemented; possibly they are not better than the corresponding macros of the `trig` package, but they are supposed to be more accurate at least they were intended to be so. The other difference is that angles are specified in sexagesimal degrees (360° to one revolution), so that reduction to the fundamental quadrant is supposed to be more accurate; the tangent of odd multiples of 90° are approximated with a "TEX infinity", that is the signed value 16383.99999. This will possibly produce computational errors in the subsequent calculations, but at least it does not stop the tangent computation. In order to avoid overflows or underflows in the computation of small angles (reduced to the first quadrant), the sine and the tangent of angles smaller than 1° are approximated by the first term of the McLaurin series, while for the cosine the approximation is given by the first two terms of the McLaurin series. In both cases theoretical errors are smaller than what TEX arithmetics can handle.

    These trigonometric functions are used within the complex number macros; but if the user wants to use them the syntax is the following:

    `\SinOf`⟨*angle*⟩`to`⟨*control sequence*⟩
    `\CosOf`⟨*angle*⟩`to`⟨*control sequence*⟩
    `\TanOf`⟨*angle*⟩`to`⟨*control sequence*⟩

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

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

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

    If the angle is specified numerically it must be enclosed in braces, while if it is specified with a control sequence the braces (curly brackets) are not necessary. The above macro `\Arc` draws a simple circular arc without arrows; `\VectorArc` draws an arc with an arrow tip at the ending point; `\VectorARC` draws an arc with arrow tips at both ends; see figure 3.

12. A multitude of commands have been defined in order to manage complex numbers; actually complex numbers are represented as a comma separated pair of fractional numbers. They are used to address to specific points in the drawing plane, but also as operators so as to scale and rotate other objects. In the following ⟨*vector*⟩ means a comma separated pair of fractional numbers, ⟨*vector macro*⟩ means a macro the contains a comma separated pair of fractional numbers; ⟨*angle macro*⟩ means a macro that contains the

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

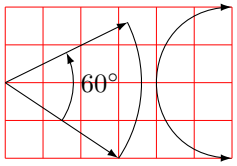

Figure 3: Arcs and curved vectors

angle of a vector in sexagesimal degrees; ⟨*argument*⟩ means a brace delimited numeric value, possibly a macro; *macro* is a valid macro name, that is a backslash followed by letters, or anything else that can receive a definition. A 'direction' of a vector is its versor; the angle of a vector is the angle between the vector and the positive x axis, generally directly used in the Euler formula $\vec{v} = Me^{j\varphi}$.

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

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

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

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

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

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

The `\Curve` macro does not (still) have facilities for cycling the path, that is to close the path from the last specified node-direction to the first specified node-direction. The tangent direction need not be specified with a unit vector, although only its direction is relevant; the scaling of the specified direction vector to a unit vector is performed by the macro itself. Therefore one cannot specify the fine tuning of the curve convexity as it can be done with other programs, as for example with METAFONT or the `pgf/tikz` package and environment. See figure 4 for an example.

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

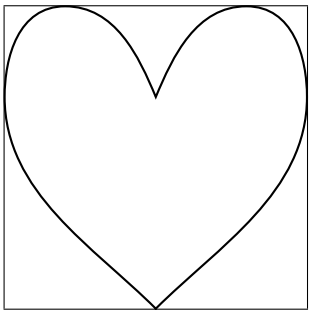

Figure 4: A heart shaped curve with cusps drawn with `\Curve`

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

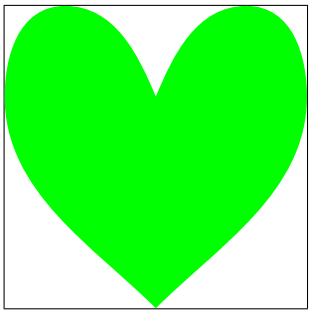

Figure 5: Coloring the inside of a closeded path drawn with `\Curve*`

With the starred version of `\Curve`, instead of stroking the contour, the macro fills up the contour with the selected current color, figure 5.

In spite of the relative simplicity of the macros contained in this package, the described macros, as well as the original ones included in the `pict2e` package, allow to produce fine drawings that were unconceivable with the original LaTeX picture environment. Leslie Lamport himself announced an extension to his environment when LaTeX 2_ε was first issued in 1994; in the `latexnews` news letter of December 2003; the first implementation announced; the first version of this package was issued in 2006. It was time to have a better drawing environment; this package is a simple attempt to follow the initial path while extending the drawing facilities; but Till Tantau's `pgf` package has gone much farther.

3 Remark

There are other packages in the CTAN archives that deal with tracing curves of various kinds. `PSTricks` and `tikz/pgf` are the most powerful ones. But there is also the package `curves` that is intended to draw almost anything by using little dots or other symbols partially superimposed to one another. It uses only quadratic Bézier curves and the curve tracing is eased by specifying only the curve nodes, without specifying the control nodes; with a suitable option to the package call it is possible to reduce the memory usage by using short straight segments drawn with the PostScript facilities offered by the `dvips` driver.

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

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

Package `xpicture` builds over the `picture` LaTeX environment so as to allow to draw the usual curves that are part of an introductory analytic geometry course; lines, circles, parabolas, ellipses, hyperbolas, and polynomials; the syntax is very comfortable; for all these curves it uses the quadratic Bézier splines.

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

4 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 the fractional part and to avoid as much as possible

any numeric overflow; many Josef's ideas are incorporated in the macro that is implemented in this package, although the macro used by Josef is slightly different from this one. Both versions aim at a better accuracy and at widening the operand ranges.

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

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

5 Source code

5.1 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.

```
\RequirePackage{color}
\RequirePackageWithOptions{pict2e}[2014/01/01]
```

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
```

5.2 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. [1]

```
\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 this and the subsequent 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 eliminate.

5.3 Improved line and vector macros

The new macro `\LIne` allows to draw an arbitrary inclination line as if it was a polygonal with just two vertices. This line should be set by means of a `\put` command so that 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. Please remember that the decimal separator is the decimal *point*, while the *comma* acts as coordinate separator. This recommendation is particularly important for non-English speaking users, since in all other languages the comma must be used as the decimal separator.

The `\line` macro is redefined by making use of a new division routine that receives in input two dimensions and yields on output their fractional ratio. 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
```

[1] Thanks to Daniele Degiorgi (`degiorgi@inf.ethz.ch`).

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 normalized 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 horizontal cosine `\d@mX`, and the line length is accordingly scaled:

```
    \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
```

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 "PostScript" commands instead of resorting to the dvi low level language that was used both in `pict2e` and in the original `picture` commands; it had a meaning in the old times, but it certainly does not have any when lines are drawn by the driver that drives the output to a 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, but it did preform in a better way with the 2004 version that was limited to integer direction coefficients up to 999 in magnitude.

5.4 Dashed and dotted lines

Dashed and dotted lines are very useful in technical drawings; here we introduce four 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 with the first

macro type are referred to the origin of the axes, while with the second macro type they are referred to P_1; both macro types have their usefulness and figures 6 and 7 show how to use these macro types.

We distinguish these macro types with an asterisk; the unstarred version is the first macro type, while the starred one refers to the second macro type.

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 can 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 one. 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. A simple `\multiput` completes the job, but in order to use the various vectors and numbers within a group and to throw the result outside the group while restoring all the intermediate counters and registers, a service macro is created with an expanded definition and then this service macro is executed. Figure 6 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 o the line length.

```
\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

```

```
\begin{picture}(40,30)
\put(0,0){\GraphGrid(40,30)}
\Dashline(0,0)(40,10){2}\Dashline(0,0)(40,20){2}
\Dashline(0,0)(40,30){2}\Dashline(0,0)(30,30){2}
\Dashline(0,0)(20,30){2}\Dashline(0,0)(10,30){2}
{\color{red}\Dashline*(40,0)(108:30){2}
\Dashline*(40,0)(126:30){2}
\Dashline*(40,0)(144:30){2}
\Dashline*(40,0)(162:30){2}}
\end{picture}
```

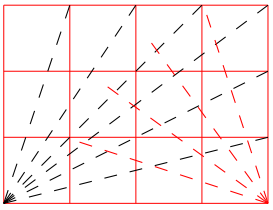

Figure 6: Different length dashed lines with the same nominal dash length

```
\def\Dashline@@(#1)(#2)#3{\put(#1){\Dashline@(0,0)(#2){#3}}}
\fi
```

A simpler `\Dotline` macro can draw a dotted line between to 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. The syntax is as follows:

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

```
\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
```

Notice that vectors as complex numbers in their cartesian and polar forms always represent a point position referred to the origin of the axes; this is why in figures 6 and 7 the dashed and dotted line that depart from the lower right corner of the graph grid, and that use polar coordinates, have to be put at the proper

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

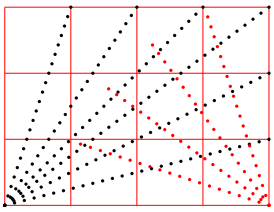

Figure 7: Different length dotted lines with the same nominal dot gap

position with the starred version of the commands that take care of the relative specification made with the polar coordinates.

5.5 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 $\langle x,y\rangle$, while the former have the syntax $\langle angle{:}radius\rangle$. The `\put` command is redefined to accept the same syntax; the whole work is done by `\SplitNod@` and its subsidiaries.

```
\def\GetCoord(#1)#2#3{%
\expandafter\SplitNod@\expandafter(#1)#2#3\ignorespaces}
```

But the macro that detects the form of the coordinates is `\isnot@polar`, that examines the parameter syntax in order to see if it contains a colon; if it does the coordinates are in polar form, otherwise they are in cartesian form:

```
\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(}%)
```

Examples of using polar and cartesian coordinates are shown in figure 8.

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

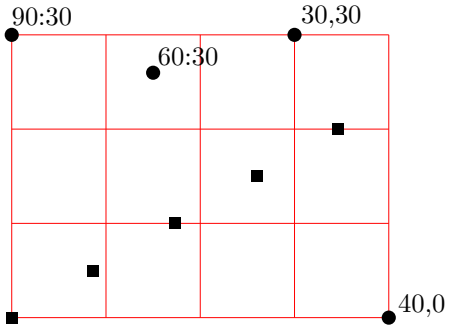

Figure 8: Use of cartesian and polar coordinates

5.6 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 checks 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 arrow 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 redefinitions and 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 and \VECTOR to be defined in a while.

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

We now make a vector with the 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 in 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. I use my division macro; since it yields a "factor" I directly use it to scale the length of the vector. I finally memorize the true vector length in the internal dimension @tdB

```
    \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
```

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 I 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; 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

```
\unitlength=.5mm
\begin{picture}(60,20)
\put(0,0){\GraphGrid(60,20)}
\put(0,0){\vector(1.5,2.3){10}}
\put(20,0){\Vector(10,15.33333)}
\VECTOR(40,0)(50,15.33333)
\end{picture}
```

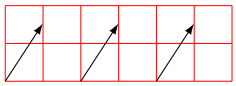

Figure 9: Three (displaced) identical vectors obtained with the three vector macros.

can determine the amount of the stem shortening; if the stem was too short and the new length is negative, we refrain from designing such 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}
```

Now 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.

```
\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 `pict2e` documentation says that if the vector length is zero the macro designs 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 examples in figure 9.

5.7 Polylines

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

`\polygonal`$(P_0)(P_1)(P_2)\ \ldots(P_n)$

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` 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 not necessarily is put in position through a `\put` command that provides to eliminate any spurious spaces preceding this command.

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 allow a specification for the joints of the various segments of a polygonal line it is necessary to allow for an optional parameter; the default join 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 it 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}}
```

5.8 The red service grid

The next command is very useful for debugging while editing one's drawings; 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 he/she is doing; nevertheless it is advisable to displace the grid by means of a `\put` command so that its grid lines coincide with the graph coordinates 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.

```
\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}
```

Rounding up is useful because also the grid margins fall on coordinates multiples of 10. It resorts to the `\Integer` macro that will be described in a while.

```
\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}%
```

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.

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

6 Math operations on fixed radix operands

This is not the place to complain about the fact that all programs of the TEX system use only integer arithmetics; LuaTeX can do floating point arithmetics through the Lua language that it partially incorporates. But this `curve2e` package is supposed to work also with pdfTeX and XeTeX. Therefore the Lua language should not be used.

The only possibility to fake fractional arithmetics is to use fractional numbers as multipliers of the unit length `\p@` that is 1 pt long; calculations are performed on lengths, and eventually their value, extracted from the length registers with the `\the` command is stripped off the "pt" component. The LATEX kernel macro does this in one step. At the same time the dimensional expressions made available by the `e-TeX` extension to all the TEX system engines, allows to perform all operations directly on suitable length registers.

The drawback of working with TEX arithmetics for dimensions is that they are saved in binary form in computer words of 32 bits; the sixteen less significant bits

are reserved for the fractional part; the two more significant bits are reserved for the sign and the type of dimension. There remain in total 30 bits available for the entire number; just to simplify this representation the TEXbook explains that the computer 32-bit word contains the dimension in *scaled points*, where 1 pt equals 2^{16} sp.

Since the number of digits of the fractional part is constant (16) it is said that the number representation is in *fixed radix*. This is much different form the scientific approach to fractional numbers where a 32-bit word reserves 24 bits to the significant digits, one bit for the sign, and a signed exponent of 2 that has 7 significant bits and represents the number of binary digits that is necessary to move the binary fractional sign to the right or to the left in order to remain with a number greater or equal to 1, but lower than 2; this way of coding numbers is called *floating point* representation (of course special numbers, such as zero, require special codes); TEX fixed radix representation may code numbers with absolute value not exceeding $(2^{30} - 1)$ sp $=1073741823$ sp $=16383.99998$ pt; a floating point 32-bit number cannot exceed in magnitude the value of approximately 1.8446744×10^{19}; with fixed radix numbers it is possible to evaluate the absolute value of the imprecision of the results by summing the absolute imprecision of the terms of summation and subtraction; with floating point numbers it is possible to estimate the relative imprecision by summing the relative imprecisions of the terms of multiplication and division.

Working with fixed radix numbers one must keep in mind that 16 fractional binary digits are more or less equivalent to 5 decimal fractional digits; and that 16383,99998 pt are a little less than six meters (5,75832 m). These limits appear completely sufficient to do most computations necessary for typography, but when we pretend to make computations of mathematical functions with such a poor "calculator", we must expect poorly approximated results. Nevertheless using the proper iterative algorithms the results are not too bad, but certainly it is necessary to accept the situation.

Then why not using the `fp` package that allows to do computations in TEX with the floating point representation of numbers? Simply because the results would require a lot of time for their execution; this is a serious problem with package `pgfplots` with which it is possible to draw beautiful 2D and 3D color diagrams, but at the expense of even dozens of seconds of computation time instead of microseconds.

6.1 The new division macro

The most important macro in the whole package is the division macro; it takes two lengths as input values and computes their fractional ratio into a control sequence.

It must take care of the signs, so that it examines the operand signs and determines the result sign separately conserving this computed sign in the macro `\segno`; this done, we are sure that both operands are or are made positive; should the numerator be zero it directly issues the zero quotient; should the denominator be zero it outputs "infinity" (`\maxdimen` in points), that is the maximum allowable length measured in points that TEX can deal with.

Since the result is assigned a value, the calling statement must pass as the third argument either a control sequence or an active character. Of course the first operand is the dividend, the second the divisor and the third the quotient.

Since `curve2e` is supposed to be an extension of `pict2e` and this macro package already contains a division macro, we might not define any other division macro; nevertheless, since the macro in `pict2e` may not be so efficient as it might be if the `e-tex` extensions of the interpreter program were available, here we check and eventually provide a more efficient macro. The latter exploits the scaling mechanism embedded in `pdftex` since 2007, when the extended mode is enabled; it is used to scale a dimension by a fraction: $L \times N/D$, where L is a dimension, and N and D are the numerator an denominator of the scaling factor; these might be integers, but it's better they are both represented by dimension registers, that contain two lengths expressed in the same units, possibly the fractional scaling factor numerator and denominator that 'scale" the unit length `\p@`.

Therefore first we test if the extended mode exists and/or is enabled:

```
\ifdefined\dimexpr
```

then we test if the macro is already defined:

```
    \unless\ifdefined\DividE
```

Notice that `\dimexpr` is the specific extended mode control sequence we are going to use in order to perform our task; if the interpreter program is too old and/or it is a recent version, but it was compiled without activating the extended mode, the macro `\dimexpr` is undefined.

The macro `\DividE`, creates a group where the names of two counters and a dimensional register are defined; the numbers of these integer and dimension registers are expressly above the value 255, because one of the extensions is the possibility of using a virtually unlimited number of registers; moreover even if these registers were used within other macros, their use within a group does not damage the other macros; we just have to use a Knuthian dirty trick to throw the result beyond the end-group command.

The efficiency of this macro is contained in the extended command `\dimexpr`; both the `\@DimA` and `\Num` registers are program words of 32 bits; the result is stored into an internal register of 64 bits; the final division by a factor stored into a register of 32 bits, so that in terms of scaled points a division by $1\,\mathrm{pt} = 1 \times 2^{16}$, scales down the result by 16 bits, and if the total length of the result is smaller than 2^{30}, the result can be correctly assigned to a dimension register. In any other case the extended features imply suitable error messages and the termination of the program. During the division and a scaling down by 16 bits, the result is not simply truncated, but it is rounded to the nearest integer (in scaled points). The first two operands are lengths and the third is a macro.

```
\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
```

We need a similar macro to divide two fractional or integer numbers, not dimensions, and produce a macro that contains the fractional result.

```
    \unless\ifdefined\DivideFN
         \def\DivideFN#1by#2to#3{\DividE#1\p@ by#2\p@ to{#3}}%
    \fi
```

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 eliminated by `\strip@pt`.

```
    \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}%
    \fi
\fi
```

The next macro uses the `\strip@pt` LaTeX kernel macro to get the numerical value of a measure in points. One has to call `\Numero` with a control sequence and a dimension; the dimension value in points is assigned to the control sequence.

```
\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
```

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`.

6.2 Trigonometric functions

We now start with trigonometric functions. We define the macros `\SinOf`, `\CosOf` and `\TanOf` (we might define also `\CotOf`, but the cotangent does not appear so essential) by means of the parametric formulas that require the knowledge of the

tangent of the half angle. We want to specify the angles in sexagesimal degrees, not in radians, so we can make accurate reductions to the main quadrants. We use the formulas

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

where

$$x = \theta/114.591559$$

is the half angle in degrees converted to radians.

We use this slightly modified set of parametric formulas because the cotangent of x is a by product of the computation of the tangent of x; in this way we avoid computing the squares of numbers that might lead to overflows. For the same reason we avoid computing the value of the trigonometric functions in proximity of the value zero (and the other values that might involve high tangent or cotangent values) and in that case we prefer to approximate the small angle function value with its first or second order truncation of the McLaurin series; in facts for angles whose magnitude is smaller than 1° the magnitude of the independent variable $y = 2x$ (the angle in degrees converted to radians) is so small (about 0.017) that the sine and tangent can be freely approximated with y itself (the error being smaller than approximately 10^{-6}), while the cosine can be freely approximated with the formula $1 - 0.5y^2$ (the error being smaller than about $\cdot 10^{-6}$).

We keep using grouping so that internal variables are local to these groups and do not mess up other things.

The first macro is the service routine that computes the tangent and the cotangent of the half angle in radians; since we have to use always the reciprocal of this value, we call it `\X@` but in spite of the similarity it is the reciprocal of x. Notice that parameter `#1` must be a length.

```
209 \def\g@tTanCotanFrom#1to#2and#3{%
210 \DividE 114.591559\p@ by#1to\X@ \@tdB=\X@\p@
```

Computations are done with the help of counter `\I`, of the length `\@tdB`, and the auxiliary control sequences `\Tan` and `\Cot` whose meaning is transparent. The iterative process controlled by `\@whilenum` implements the (truncated) continued fraction expansion of the tangent function.

$$\tan x = \cfrac{1}{\cfrac{1}{x}-\cfrac{1}{\cfrac{3}{x}-\cfrac{1}{\cfrac{5}{x}-\cfrac{1}{\cfrac{7}{x}-\cfrac{1}{\cfrac{9}{x}-\cfrac{1}{\cfrac{11}{x}-\dots}}}}}}$$

```
\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}%
```

Now that we have the macro for computing the tangent and cotangent of the half angle, we can compute the real trigonometric functions we are interested in. The sine value is computed after reducing the sine argument to the interval $0° < \theta < 180°$; actually special values such as 0°, 90°, 180°, et cetera, are taken care separately, so that CPU time is saved for these special cases. The sine sign is taken care separately according to the quadrant of the sine argument.

Since all computations are done within a group, a trick is necessary in order to extract the sine value from the group; this is done by defining within the group a macro (in this case `\endSinOf`) with the expanded definition of the result, but in charge of of closing the group, so that when the group is closed the auxiliary function is not defined any more, although its expansion keeps getting executed so that the expanded result is thrown beyond the group end.

```
\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}%
```

For the computation of the cosine we behave in a similar way using also the identical trick for throwing the result beyond the group end.

```
\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}%
```

For the tangent computation we behave in a similar way, except that we consider the fundamental interval as $0° < \theta < 90°$; for the odd multiples of 90° we assign the result a TEX infinity value, i.e. `\maxdimen`, the maximum dimension TEX can handle.

```
\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}%
```

As of today the anomaly (angle) of a complex number may not be necessary, but it might become useful in the future; 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. For the principal value of the

arctangent we would like to use the continued fraction:

$$\arctan x = \cfrac{x}{1 + \cfrac{x^2}{3 - x^2 + \cfrac{(3x)^2}{5 - 3x^2 + \cfrac{(5x)^2}{7 - 5x^2 + \cfrac{(7x)^2}{9 - 7x^2 + \ddots}}}}} \tag{1}$$

but after some testing we had to give up due to the slow convergence of continued fraction (1), strictly connected with the slow convergence of the McLaurin series from which it is derived.

Waiting for a faster convergence continued fraction, we examined the parametric formula and its inverse:

$$\tan\theta = \frac{2\tan(\theta/2))}{1 - \tan^2(\theta/2)} \tag{2a}$$

$$\tan(\theta/2) = \frac{\sqrt{\tan^2\theta + 1} - 1}{\tan\theta} \tag{2b}$$

If we count the times we use the above formula we can arrive at a point where we have to compute the arctangent of a very small value, where the arctangent and its argument are approximately equal, so that the angle value in radians is equal to its tangent; at that point we multiply by 2^n, where n is the number of bisections, and transform the radians in degrees. The procedure is pretty good, even if is is very rudimental and based on an approximation; the fixed radix computation of the typesetting engine does not help, but we get pretty decent results, although we loose some accuracy that hopefully would not harm further computations.

The results obtainable with equation (2b) are possibly acceptable, but the square that must be computed in it tends to go in underflow if too many iterations are performed and the algorythim crashes; therefore it's virtually impossibile to get more than three correct digits after the decimal separator.

It is probably better to refer to the Newton iterations for solving the equation:

$$\tan\theta - \tan\theta_\infty = 0 \tag{3}$$

in the unknown θ given the value $t = \tan\theta_\infty$; see figure 10.

The iterative algorithm with Newton method implies the recurrence

$$y'_{i-1} = \frac{\mathrm{d}\tan(\theta_{i-1})}{\mathrm{d}\theta} = \frac{1}{\cos^2\theta_{i-1}} \tag{4a}$$

$$\theta_i = \theta_{i-1} - \frac{\tan\theta_{i-1} - t}{y'_{i-1}} = \theta_{i-1} - \cos^2\theta_{i-1}(\tan\theta_{i-1} - t) \tag{4b}$$

The algorithm starts with an initial value θ_0; at each iteration for $i = 1, 2, 3, \ldots$ a new value of θ_i is computed from the data of the previous iteration $i - 1$. When

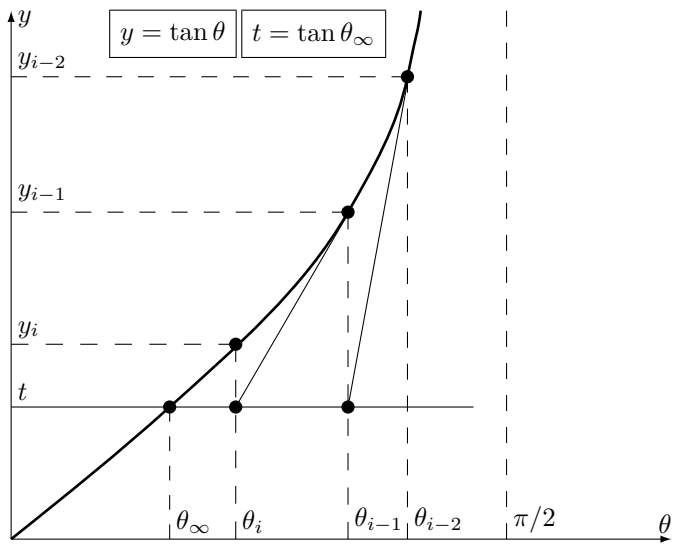

Figure 10: Newton's method of tangents

for a certain i, $\tan\theta_i$ is sufficiently close to t, the iterations may be stopped; since we already have the algorithms for computing both the tangent and the cosine; such Newton iterative method does not set forth any problem, especially if we use the properties of the trigonometric functions and we confine the computations to the first quadrant.

```
\def\ArcTanOf#1to#2{\bgroup
\edef\@tF{#1}\@tdF=\@tF\p@ \@tdE=57.295778\p@
\@tdD=\ifdim\@tdF>\z@ \@tdF\else -\@tdF\fi
\unless\ifdim\@tdD>0.02\p@
    \def\@tX{\strip@pt\dimexpr57.295778\@tdF\relax}%
\else
    \edef\@tX{45}\relax
    \countdef\I 2523 \I=8\relax
    \@whilenum\I>0\do{\TanOf\@tX to\@tG
    \edef\@tG{\strip@pt\dimexpr\@tG\p@-\@tdF\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
\edef\x{\egroup\noexpand\edef\noexpand#2{\@tX}}\x\ignorespaces}%
```

6.3 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's 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.

6.4 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 use by the `picture` environment) there is the other conventional representation used by the engineers that stress 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, these 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 wee need macros for summing, subtracting, multiplying, dividing complex numbers, for determining their directions (unit vectors); 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 a clever square root of a function of the real and the imaginary parts; 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:

```
\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.

The magnitude M is determined by taking the moduli of the real and imaginary parts, changing their signs if necessary; the larger component is 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 is quite easy to get its square root by means of the Newton iterative process; due to the quadratic convergence, five iterations are more than sufficient. When one of the components is zero, the Newton iterative process is skipped. The overall macro is the following:

```
\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}%
```

As a byproduct of the computation the control sequence `\@tempdimc` contains a length the value in points of which is the computed root.

Since the macro for determining the magnitude of a vector is available, we can now normalize 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 normalization.

```
\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}%
```

A cumulative macro uses the above ones for determining 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 normalized 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{%
\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}%
```

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).

```
\def\DirFromAngle#1to#2{%
\CosOf#1to\t@X
\SinOf#1to\t@Y
\MakeVectorFrom\t@X\t@Y to#2\ignorespaces}%
```

Since we have the algorithm to compute the arctangent of a number, it should be relatively easy to compute the angle of a complex number. We just have to pay attention that the algorithm to compute the arctangent does not care about the quadrant where the complex number lays in, and it yields the principal value of the arctan in the domain $-\pi/2 < \theta \leq \pi/2$. With complex numbers we have just a sign change in their angle when they lay in the first or the fourth quadrants; while for the third and second quadrants we have to reflect the complex number to its opposite and in the result we have to add a "flat angle", that is 180° since we are working in degrees. Even if mathematically it is undefined we decided to assign a null angle to a null complex number; possibly a warning message would be helpful, but for drawing purposes we think that the problem is irrelevant.

```
\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}
```

It is worth noting that the absolute error in these computations is lower than 0.0001°; pretty satisfactory since the typesetting engines work in fixed radix notation with 16 fractional binary digits, and an error on the fifth fractional decimal digit is almost the best it can be expected from this kind of arithmetics.

Sometimes it is necessary to scale 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
\@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}%
```

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
\@tempdima=-\t@Y\p@\Numero\t@Y\@tempdima
\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
\@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}%
```

Then the subtraction:

```
\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}%
```

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 I could not find a simple means for doing so; therefore I use the prefixed notation, that is I put the asterisk before the second operand. The first part of the multiplication macro just takes care of the multiplicand and then checks for the asterisk; if there is no asterisk it calls a second service macro that performs a regular complex multiplication, otherwise it calls a third service macro that executes the conjugate multiplication.

```
\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}
```

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 opposite direction of the divisor; therefore:

```
\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}%
```

6.5 Arcs and curved vectors

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

6.5.1 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 memorized 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 memorized 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 stating 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 trace 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 tracing the requested arc, except stroking it; I leave the `stroke` 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 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
```

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, 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

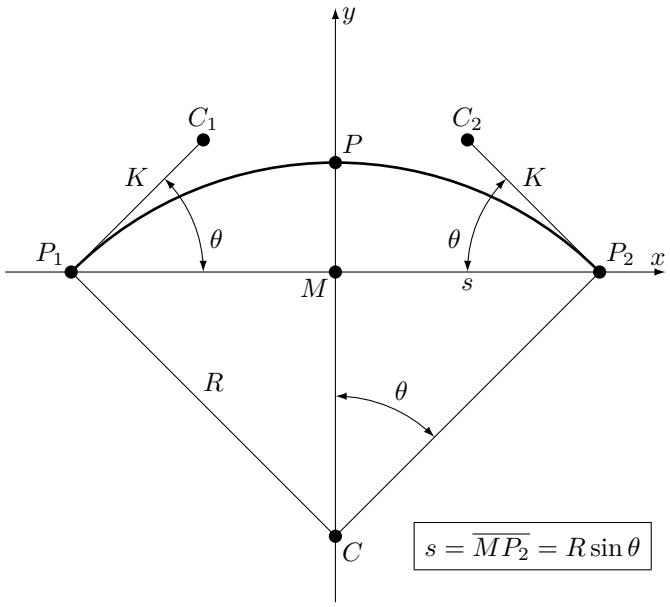

Figure 11: Nodes and control points for an arc to be approximated with a cubic Bézier spline

formula

$$K = \frac{4}{3}\frac{1-\cos\theta}{\sin\theta}R = \frac{4}{3}\frac{1-\cos\theta}{\sin^2\theta}s \tag{5}$$

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 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}
```

6.5.2 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(s); the only arrow (or both ones) 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 be corresponding 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 arrowed 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 I did not try to optimize 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 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}%
```

The macro for the arc terminated with arrow tips at both ends is again very similar, except 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 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}%
```

It must be understood that the curved vectors, the above circular arcs terminated with an arrow tip 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.

6.6 General curves

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 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}%
```

For backwards compatibility the old command with lower case `and` is made to do the same as this macro `\CurveBetween` with capitalised `And`.

Actually the above macro is a special case of concatenation of the triplet formed by macros `\StartCurve`, `\CurveTo` and`\CurveFinish`; the second of which can be repeated an arbitrary number of times. In any case the directions specified with the direction arguments, both here and with the more general macro`\Curve`, the angle between the indicated tangent and the arc chord should never exceed 90° in absolute value; strange error messages may be issued by the interpreter. Some control is exercised on these values, but some tests might fail if the angle derives from computations; this is a good place to use polar forms for the direction vectors.

The first macro initializes the drawing and the third one strokes it; the real work is done by the second macro. The first macro initializes the drawing but also memorizes the starting direction; the second macro traces the current Bézier arc reaching the destination point with the specified direction, but memorizes 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 memorized 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

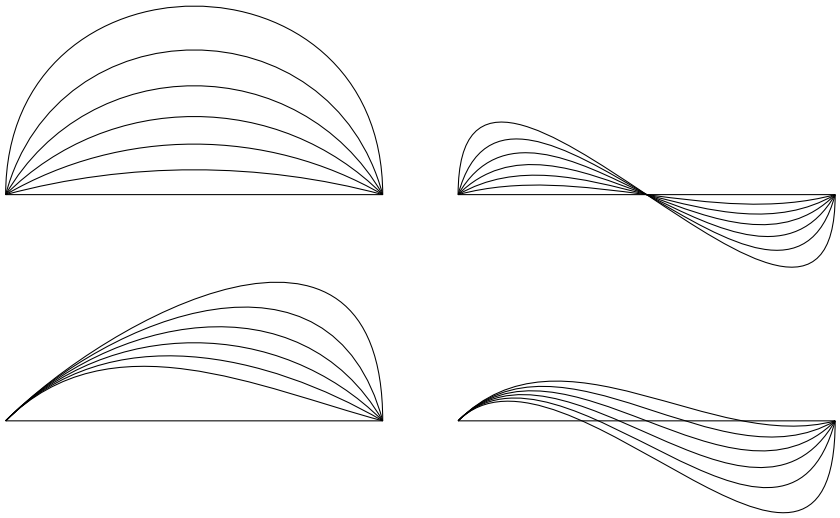

Figure 12: Curves between two points

curve joining point at the cusp is not stroked with the same command, therefore we get two superimposed curve terminations. We therefore need another small macro `\ChangeDir` to perform this task.

It is necessary to recall that the directions 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 normalization and memorization.

The next desirable point would be to design a macro that accepts optional node directions and computes the missing ones according to a suitable strategy. I 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 initializing macro that receives in the first argument the starting point and in the second argument the direction of the tangent (not necessarily normalized 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}
```

And this re-initializes 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 will be therefore:

`\CbezierTo`⟨*end point*⟩`WithDir`⟨*direction*⟩`AndDist`⟨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 point from their respective node points; they must be positive integers or fractional numbers.

This macro uses the input information to use 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 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
```

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

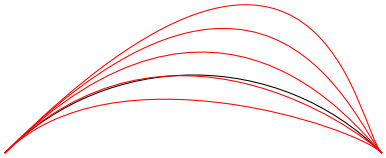

Figure 13: Comparison between similar arcs drawn with `\CurveBetween` (black) and `\CbezierTo` (red)

```
\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 as the other macro `\CurveBetween` described previously.

```
\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; notice that the tangents at the end points are the same for the black curve drawn with `\CurveBetween` and the four 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 of the distances of the control point are sort of unpredictable 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 with another specified direction (final node). 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 I 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, 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.

We use the formula we got for arcs (5), where the half chord is indicated with s, and we derive the necessary distances:

$$K_0 = \frac{4}{3} s \frac{1 - \cos\theta_0}{\sin^2\theta_0} \tag{6a}$$

$$K_1 = \frac{4}{3} s \frac{1 - \cos\theta_1}{\sin^2\theta_1} \tag{6b}$$

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

```
\def\CurveTo#1WithDir#2{%
\def\@Puno{#1}\def\@Duno{#2}\DirOfVect\@Duno to\@Duno
\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 by*\@DirChord to \@Dpzero
\MultVect\@Duno by*\@DirChord to \@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.

```
\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
```

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@)
    \DividE\@KCzero\@tdA by\@SinDzero\p@ to \@KCzero
  \fi
\fi
\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@)
    \DividE\@KCuno\@tdA by\@SinDuno\p@ to \@KCuno
  \fi
\fi
\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 tracing.

```
\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}%
```

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 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 flavors: 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 regular parentheses while direction components are grouped within angle brackets. 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{\@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}
```

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 **??** 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 three times as many pieces of information in order to minimise the effects of the lack of control on the control

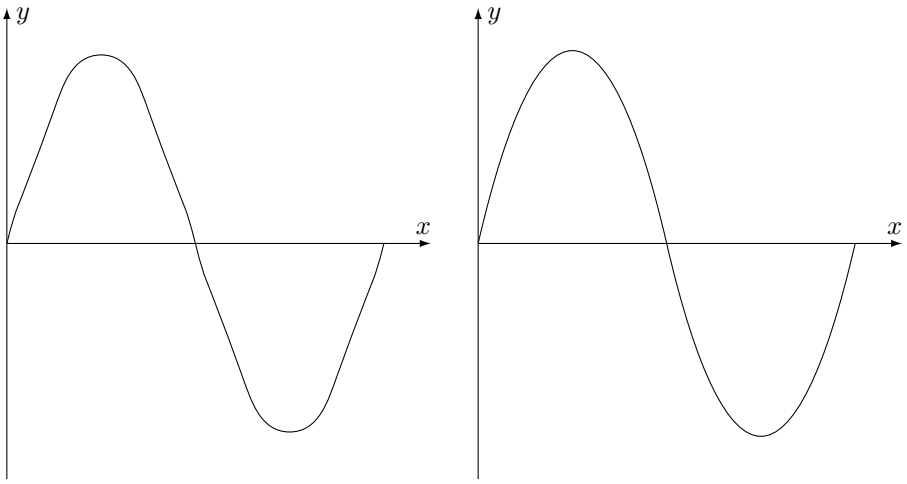

Figure 14: A sequence of arcs; the left figure has been drawn with the `\Curve` command with a sequence of nine couples of point-direction arguments; the right figure has been drawn with two commands `\CbezierBetween` that include also the specification of the control points

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 far much easier to use than with single arcs drawn with the `\CbezierBetween` macro.

I believe that the set of new macrosprovided 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.

References

[1] Gäßlein H., Niepraschk R., and Tkadlec J. *The `pict2e` package*, 2009, PDF document attached to the "new" `pict2e` bundle; the bundle may be downloaded from any CTAN archive or one of their mirrors.