The extension package curve2e*

Claudio Beccari

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. Moreover the xetex.def driver was introduced so that certian commands previously defined in this extension not only become unnnecessary, but also would produce errors when the program is used under XeLaTeX. Therefore these commands were either eliminated or corrected.

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.

*Version number v.1.40; last revised 2012/03/05.

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_ε since its beginnings in 1994. The dummy package was just issuing an info message that simply announced the temporary unavailability of the real package.

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

1. the line and vector slopes were limited to the ratios of relatively 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. 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`); the 2011 upgrade of `pict2e` made these commands superfluous and redefined the internal special commands for the drivers, so that I deicided to completely eliminate these definitions and relay on those produced by `pict2e`;

2. 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`);

3. 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 conflicts with `pict2e` 2009 version; therefore its name is changed to `\LIne` and supposedly it will not be used very often, if ever used;

4. 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`; of course there is no need to use the `\put` command with this line specification;

5. 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) must be specified;

6. 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; `\VECTOR` joins two specified points (without using the `\put` command) with the arrow pointing to the second point;

7. 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`);

8. a new macro `\Arc` is defined in order to draw an arc with arbitrary radius and arbitrary angle amplitude; this amplitude is specified in sexagesimal degrees, not in radians; the same functionality is now achieved with the `\arc` macro of `pict2e`, which provides also the star version `\arc*` that fills up the interior of the generated circular arc. It must be noticed that the syntax is slighltly different, so that it's reasonable that both commands, in spite of producing identical arcs, might be more comfortable with this or that syntax.

9. two new macros are defined in order to draw circular arcs with one arrow at one or both ends;

10. a new macro `\Curve` is defined so as to draw arbitrary curved lines by means of third order Bézier splines; the `\Curve` macro requires only the curve nodes and the direction of the tangents at each node.

In order to make the necessary calculations many macros have been defined so as to use complex number arithmetics to manipulate point coordinates, directions, rotations and the like. The trigonometric functions have also been defined in a way that the author believes to be more efficient than that implied by the `trig` package; in any case the macro names are sufficiently different to accommodate both definitions 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 beta version could become the start for a real extension of the `pict2e` package or even become a part of it.

For this reason I suppose that every enhancement should be submitted to Gäßlein and Niepraschk who are the prime maintainers of `pict2e`; they only 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 whom 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 commands `\linethickness`, `\thicklines`, `\thinlines` together with `\defaultlinethickness` always redefine the internal `\@wholewidth` and `\@halfwidth` so that the former always refer to a full width and the latter to a half of it in this way: if you issue the command `\defaultlinewidth{2pt}` all thin lines will be drawn with a thickeness 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{`⟨*dimension value*⟩`}`
`\defaultlinewidth{`⟨*dimension value*⟩`}`

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

3. 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.

4. 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 the 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.

5. 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 sequence the specified points; the syntax is:

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

   See figure 1 where a pentagon is designed..

```
\unitlength=.5mm
\begin{picture}(40,32)(-20,0)
\polyline(0,0)(19.0211,13,8197)(11.7557,36.1803)%
(-11.7557,36.1803)(-19.0211,13,8197)(0,0)
\end{picture}
```

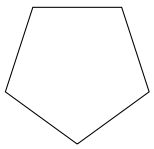

Figure 1: Polygonal line obtained by means of the `\polyline` command

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, therefore closing the path. `pict2e` defines also the starred command that fills up the inside of the generated polygon.

6. The new command

   `\Dline(`*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.5mm
\begin{picture}(40,40)
\put(0,0){\GraphGrid(40,40)}
\Dline(0,0)(40,10){4}
\put(0,0){\circle*{2}}
\Dline(40,10)(0,25){4}
\put(40,10){\circle*{2}}
\Dline(0,25)(20,40){4}
\put(0,25){\circle*{2}}
\put(20,40){\circle*{2}}
\end{picture}
```

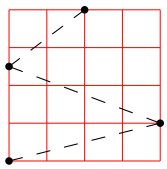

Figure 2: Dashed lines and graph grid

7. `\GraphGrid` is a command that draws a red grid over the drawing area with lines separated `10\unitlengths`; it is described only with a comma separated couple of numbers, representing the base and the height of the grid, see figure **??**; it's better to specify multiples of ten and the grid can be placed anywhere in the drawing plane by means of `\put`, whose 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 plane. This grid is intended as an aid in drawing; even if you sketch your drawing on millimeter paper, the drawing grid turns out to be very useful; one must only delete or comment out the command when the drawing is finished.

8. 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

```
\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}
```

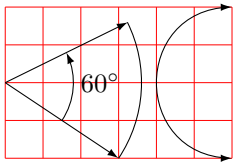

Figure 3: Arcs and curved vectors

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.

9. 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.

10. 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 point 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, possibly stored in macros; ⟨*argument*⟩ means a brace delimiteded numeric value, possibly a macro; *macro* is a valid macro name, a backslash followed by letters, or anything else that can receive a definition.

    - `\MakeVectorFrom`⟨*two arguments*⟩`to`⟨*vector*⟩

- `\CopyVect`⟨*first vector*⟩`to`⟨*second vector*⟩
- `\ModOfVect`⟨*vector*⟩`to`⟨*macro*⟩
- `\DirOfvect`⟨*vector*⟩`to`⟨*macro*⟩
- `\DmodAndDirOfVect`⟨*vector*⟩`to`⟨*first macro*⟩`and`⟨*second macro*⟩
- `\DistanceAndDirOfVect`⟨*first vector*⟩`minus`⟨*second vector*⟩`to`⟨*first macro*⟩`and`⟨*second macro*⟩
- `\XpartOfVect`⟨*vector*⟩`to`⟨*macro*⟩
- `\YpartOfVect`⟨*vector*⟩`to`⟨*macro*⟩
- `\DirFromAngle`⟨*angle*⟩`to`⟨*macro*⟩
- `\ScaleVect`⟨*vector*⟩`by`⟨*scaling factor*⟩`to`⟨*macro*⟩
- `\ConjVect`⟨*vector*⟩`to`⟨*conjugate vector*⟩
- `\SubVect`⟨*first vector*⟩`from`⟨*second vector*⟩`to`⟨*vector*⟩
- `\AddVect`⟨*first vector*⟩`and`⟨*second vector*⟩`to`⟨*vector*⟩
- `\MultVect`⟨*first vector*⟩`by`⟨*second vector*⟩`to`⟨*vector*⟩
- `\MultVect`⟨*first vector*⟩`by*`⟨*second vector*⟩`to`⟨*vector*⟩
- `\DivVect`⟨*first vector*⟩`by`⟨*second vector*⟩`to`⟨*vector*⟩

11. General curves can be drawn with the `pict2e` macro `\curve` but it requires the specification of the Bézier third order 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.

In spite of the relative simplicity of the macros contained in this package, the described macros, as well as the original macros included in the `pict2e` package, allow to produce fine drawings that were inconceivable of with the original LaTeX

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

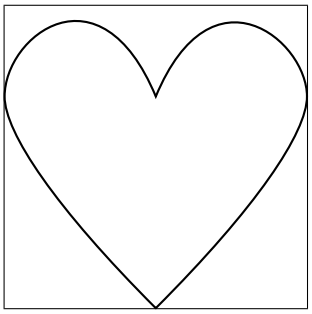

Figure 4: A heart shaped curve with cusps drawn with `\Curve`

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 appeared; 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 Notice

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 are also the package `curves` that is intended to draw almost anything by using little dots or other symbols partially superimposed to one another. It used 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.

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. Some of the work we did together was incorporated in `pict2e` 2009.

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}[2011/04/01]
```

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 developers, not the users.

```
\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
```

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.

5.1.1 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. The two arguments define the horizontal and the vertical component respectively.

```
\def\LIne(#1,#2){\pIIe@moveto\z@\z@
      \pIIe@lineto{#1\unitlength}{#2\unitlength}\pIIe@strokeGraph}%
```

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 is to 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 the ISO regulations allow the decimal point only for English speaking countries, while in all other countries 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
```

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 muche 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

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

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, it 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.

```
    \pIIe@moveto\z@\z@
    \pIIe@lineto{\d@mX\@linelen}{\d@mY\@linelen}%
    \pIIe@strokeGraph
  \fi
\endgroup\ignorespaces}%
```

The new definition of the command `\line`, besides tha ease with which is readable, does not do different things from the definition of `pict2e` 2009, but it did preform in a better way whith the 2004 version that was limited to integer direction coefficients up to 999 in magnitude.

Another usefull line-type macro creates a dashed line between two given points with a dash length that must be specified; actually the specified dash length is a desired dash length; the actual length is computed by integer division between the distance of the given points and the desired dash length; this integer is tested in order to see if it's odd; if it's not, it is increased by one. Then the actual dash length is obtained by dividing the above distance by this odd number. Another vector is created from $P_1 - P_0$ by dividing it by the magic odd number; then it is multiplied by two in order to have the increment from one dash to the next, and finally the number of patterns is obtained by integer dividing the magic odd 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.

```
\ifx\Dline\undefined
\def\Dline(#1,#2)(#3,#4)#5{%
\begingroup
   \countdef\NumA254\countdef\NumB252\relax
   \MakeVectorFrom{#1}{#2}to\V@ttA
   \MakeVectorFrom{#3}{#4}to\V@ttB
   \SubVect\V@ttA from\V@ttB to\V@ttC
   \ModOfVect\V@ttC to\DlineMod
   \DividE\DlineMod\p@ by#5\p@ 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\endgroup
   \noexpand\multiput(\V@ttA)(\V@ttC){\number\NumB}{\noexpand\LIne(\V@ttB)}}%
   \@mpt\ignorespaces}%
\fi
```

The new macro `\GetCoord` splits a vector (or complex number) specification into its components:

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

But the macro that does the real work is `\SplitNod@`:

```
\def\SplitNod@(#1,#2)#3#4{\edef#3{#1}\edef#4{#2}}%
```

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
      \pIIe@fillGraph
```

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 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@
        \pIIe@moveto\z@\z@
        \pIIe@lineto{\@xnum\@linelen}{\@ynum\@linelen}%
        \pIIe@strokeGraph\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,#2){%
\ifdim#1\p@=\z@\vector(#1,#2){#2}
\else
\vector(#1,#2){#1}\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 5.

5.1.2 Polygonal lines

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

`\polygonal`$(P_0)(P_1)(P_2)$ `...`(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.

```
\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}
```

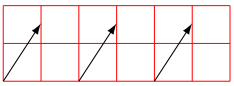

Figure 5: Three (displaced) identical vectors obtained with the three vector macros.

```
\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 allow the line join specification.

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

```
\providecommand*\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}%
    {Polygonal lines require at least two vertices!\MessageBreak
     Control your polygonal line 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}{\pIIe@strokeGraph\ignorespaces}}
```

5.1.3 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){\begingroup\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}%
\endgroup\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}%
```

5.2 The new division macro

Now comes one of the most important macros in the whole package: 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 `pic2e` and this macro package already contains a division maro, we do not define any other division macro; nevetheless, since the macro in `pic2e` 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, if the extended mode is enabled, that 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 represent the numbers of scaled points another two dimensions correspond to, in the philosophy that floating point numbers are represented by the measures of lengths in points.

Therefore first we test if the macro is already defined:

```
\ifx\DividE\undefined
```

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

```
\ifx\dimexpr\undefined\else
```

Notice that `\dimexpr` is the specific extended mode control sequence we are going to use in order to perform our task; if the interpeter 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, 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 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 end the termination of the program. During the division a scaling down by 16 bits, the result is not simply truncated, but it is rounded to the nearest integer (in scaled points)

```
\def\DividE#1by#2to#3{%
\begingroup
\countdef\Num2254\relax \countdef\Den2252\relax
\dimendef\@DimA 2254
\Num=\p@ \@DimA=#2\relax \Den=\@DimA
\ifnum\Den=\z@
\edef\x{\noexpand\endgroup\noexpand\def\noexpand#3{\strip@pt\maxdimen}}%
\else
```

```
\@DimA=#1\relax
\@DimA=\dimexpr\@DimA*\Num/\Den\relax
\edef\x{\noexpand\endgroup\noexpand\def\noexpand#3{\strip@pt\@DimA}}%
\fi
\x}
\fi\fi
```

The next two macros are one of the myriad variants of the dirty trick used by Knuth for separating a measure from its units that *must* be points, "`pt`". One has to call `\Numero` with a control sequence and a dimension; the dimension value in points is assigned to the control sequence.

```
\ifx\undefined\@Numero%
  {\let\cc\catcode \cc`p=12\cc`t=12\gdef\@Numero#1pt{#1}}%
\fi
\ifx\undefined\Numero
  \def\Numero#1#2{\dimen254#2\relax
  \edef#1{\expandafter\@Numero\the\dimen254}\ignorespaces}%
\fi
```

For both macros the `\ifx`...`\fi` constructs avoid messing up the definitions I have in several packages.

5.3 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 (less than 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 $4 \cdot 10^{-9}$).

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.

```
\def\g@tTanCotanFrom#1to#2and#3{%
\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=254\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 $0^\circ < \theta < 180^\circ$; 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{\begingroup%
\@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\endgroup
  \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{\begingroup%
\@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\endgroup
    \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 $0^\circ < \theta < 90^\circ$; for the odd multiples of 90° we assign the result a TEX infinity value, that is the maximum a dimension can be.

```
\def\TanOf#1to#2{\begingroup%
\@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\endgroup
   \noexpand\def\noexpand#2{\@tempA}\noexpand\ignorespaces}%
\endTanOf}%
```

5.4 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*⟩ is positive the arc runs counterclockwise from the starting point; clockwise if it's negative.

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 scale-rotate operators.

5.5 Complex number macros

We need therefore macros for summing, subtracting, multiplying, dividing complex numbers, for determining they directions (unit vectors); a unit vector is the complex number divided by its magnitude so that the result is the Cartesian form of the Euler's formula

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

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>\@tempdimb
   \DividE\@tempdimb by\@tempdima to\@T
   \@tempdimc=\@tempdima
\else
   \DividE\@tempdima by\@tempdimb to\@T
   \@tempdimc=\@tempdimb
\fi
\ifdim\@T\p@=\z@
\else
   \@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
```

```
\Numero#2\@tempdimc
\ignorespaces}%
```

As a byproduct of the computation the control sequence `\@tempdimc` contains the vector or complex number magnitude multiplied by the length of one point.

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
\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#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}%
```

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}%
```

5.6 Arcs and curved vectors

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

5.6.1 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@
\ifdim\@tdA=\z@\else
  \@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
\pIIe@strokeGraph\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. Their distance K from the adjacent nodes is determined with the formula

$$K = \frac{4}{3}\frac{1-\cos\theta}{\sin\theta}R$$

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

```
\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}
```

5.6.2 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 option of the `pict2e` package.

But the specific drawing done here shortens the arc 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 (h) then drawing the final 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
\pIIe@strokeGraph\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=\ifx\Segno--\fi\@gradi\p@ \Numero\@tempa\@tdD
```

```
\DirFromAngle\@tempa to\@Dir
\MultVect\@V by\@Dir to\@sPun% corrects the end point
\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\@tempB\@tdD
\DirFromAngle\@tempB to\@Dird
\MultVect\@Dir by*\@Dird to\@Dir
\GetCoord(\@Dir)\@xnum\@ynum
\put(\@tdX,\@tdY){\vector(\@xnum,\@ynum){0}}% arrow tip at the end point
\@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{\ifx\Segno-\else-\fi\@ne}%
\MultVect\@V by0,\@tempa to\@vPun
\@tdE\ifx\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}}% arrow tip at the starting point
\edef\@tempa{\ifx\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
\pIIe@strokeGraph\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.

5.7 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}%
```

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.

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 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 after a cusp

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

```
\ignorespaces}
```

The next macro is the finishing one; it strokes the whole curve and closes the group that was opened with `\StartCurve`.

```
\def\CurveFinish{\pIIe@strokeGraph\endgroup\ignorespaces}%
```

The "real" 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 parts each of which should be interpreted as half the chord of the osculating circle; this curve chord division is made proportionally to the projection of the tangent directions on the chord itself. Excluding degenerate cases that may be dealt with directly, imagine the triangle built with the chord and the two tangents; this triangle is straightforward if there is no inflection point; otherwise it is necessary to change one of the two directions by reflecting it about the chord. This is much simpler to view if a general rotation of the whole construction is made so as to bring the curve chord on the x axis, because the reflection about the chord amounts to taking the complex conjugate of one of the directions. In facts with a concave curve the "left" direction vector arrow and the "right" direction vector tail lay in the same half plane, while with an inflected curve, they lay in opposite half plains, so that taking the complex conjugate of one of directions re-establishes the correct situation for the triangle we are looking for.

This done the perpendicular from the triangle vertex to the cord divides the chord in two parts (the foot of this perpendicular may lay outside the chord, but this is no problem since we are looking for positive solutions, so that if we get negative numbers we just negate them); these two parts are taken as the half chords of the osculating circles, therefore there is no problem determining the distances K_{left} and K_{right} from the left and right nodes by using the same formula we used with circular arcs. Well... the same formula means that we have to determine the radius from the half chord and its inclination with the node tangent; all things we can do with the complex number algebra and macros we already have at our disposal. If we look carefully at this computation done for the circular arc we discover that in practice we used the half chord length instead of the radius; so the coding is actually the same, may be just with different variable names.

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)\@Xpzero\@Ypzero
\GetCoord(\@Dpuno)\@Xpuno\@Ypuno
```

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

We now examine the various degenerate cases, when either tangent is perpendicular to the chord, or when it is parallel pointing inward or outward, with or without inflection.

We start with the 90° case for the "left" direction separating the cases when the other direction is or is not 90° ...

```
\ifdim\@Xpzero\p@=\z@
  \ifdim\@Xpuno\p@=\z@
    \@tdA=0.666666\p@
    \Numero\@Mcpzero{\@Chord\@tdA}%
    \edef\@Mcpuno{\@Mcpzero}%
  \else
    \@tdA=0.666666\p@
    \Numero\@Mcpzero{\@Chord\@tdA}%
    \SetCPmodule\@Mcpuno from\@ne\@Chord\@Dpuno%
  \fi
```

... from when the "left" direction is not perpendicular to the chord; it might be parallel and we must distinguish the cases for the other direction ...

```
\else
  \ifdim\@Xpuno\p@=\z@
    \@tdA=0.666666\p@
    \Numero\@Mcpuno{\@Chord\@tdA}%
    \SetCPmodule\@Mcpzero from\@ne\@Chord\@Dpzero%
  \else
     \ifdim\@Ypzero\p@=\z@
       \@tdA=0.333333\p@
       \Numero\@Mcpzero{\@Chord\@tdA}%
       \edef\@Mcpuno{\@Mcpzero}%
```

... from when the left direction is oblique and the other direction is either parallel to the chord ...

```
     \else
       \ifdim\@Ypuno\p@=\z@
         \@tdA=0.333333\p@
         \Numero\@Mcpuno{\@Chord\@tdA}%
         \SetCPmodule\@Mcpzero from\@ne\@Chord\@Dpzero
```

... and, finally, from when both directions are oblique with respect to the chord; we must see if there is an inflection point; if both direction point to the same half plane we have to take the complex conjugate of one direction so as to define the triangle we were speaking about above.

```
      \else
        \@tdA=\@Ypzero\p@ \@tdA=\@Ypuno\@tdA
        \ifdim\@tdA>\z@
          \ConjVect\@Dpuno to\@Dwpuno
        \else
          \edef\@Dwpuno{\@Dpuno}%
        \fi
```

The control sequence `\@Dwpuno` contains the right direction for forming the triangle; we can make the weighed subdivision of the chord according to the horizontal components of the directions; we eventually turn negative values to positive ones since we are interested in the magnitudes of the control vectors.

```
        \GetCoord(\@Dwpuno)\@Xwpuno\@Ywpuno
        \@tdA=\@Xpzero\p@ \@tdA=\@Ywpuno\@tdA
        \@tdB=\@Xwpuno\p@ \@tdB=\@Ypzero\@tdB
        \DividE\@tdB by\@tdA to\@Fact
        \@tdC=\p@ \advance\@tdC-\@Fact\p@
        \ifdim\@tdC<\z@ \@tdC=-\@tdC\fi
        \DividE\p@ by \@Fact\p@ to\@Fact
        \@tdD=\p@ \advance\@tdD-\@Fact\p@
        \ifdim\@tdD<\z@ \@tdD=-\@tdD\fi
```

Before dividing by the denominator we have to check the directions, although oblique to the chord are not parallel to one another; in this case there is no question of a weighed subdivision of the chord

```
        \ifdim\@tdD<0.0001\p@
          \def\@factzero{1}%
          \def\@factuno{1}%
        \else
          \DividE\p@ by\@tdC to\@factzero
          \DividE\p@ by\@tdD to\@factuno
        \fi
```

We now have the subdivision factors and we call another macro for determining the required magnitudes

```
        \SetCPmodule\@Mcpzero from\@factzero\@Chord\@Dpzero
        \SetCPmodule\@Mcpuno from\@factuno\@Chord\@Dwpuno
      \fi
    \fi
  \fi
\fi
```

Now we have all data we need and we determine the positions of the control points; we do not work any more on the rotated diagram of the horizontal chord, but we operate on the original points and directions; all we had to compute, after all, were the distances of the control points along the specified directions; remember that the “left” control point is along the positive “left” direction, while the “right” control point precedes the curve node along the “right” direction, so that a vector subtraction must be done.

```
\ScaleVect\@Dzero by\@Mcpzero to\@CPzero
```

```
\AddVect\@Pzero and\@CPzero to\@CPzero
\ScaleVect\@Duno by\@Mcpuno to\@CPuno
\SubVect\@CPuno from\@Puno 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 memorizes the final point as the initial point of the next spline

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

The next macro is used to determine the control vectors lengths when we have the chord fraction, the chord length and the direction along which to compute the vector; all the input data (arguments from #2 to #4) may be passed as control sequences so the calling statement needs not use any curly braces.

```
\def\SetCPmodule#1from#2#3#4{%
\GetCoord(#4)\t@X\t@Y
\@tdA=#3\p@
\@tdA=#2\@tdA
\@tdA=1.333333\@tdA
\@tdB=\p@ \advance\@tdB +\t@X\p@
\DividE\@tdA by\@tdB to#1\relax
\ignorespaces}%
```

We finally define the overall `\Curve` macro that recursively examines 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 initializes 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 command and exits the recursive process. The `@ChangeDir` macro is just an interface for executing the regular `\ChangeDir` macro, but also for recursing again by recalling `\@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\CurveFinish}}
\def\@ChangeDir[#1]{\ChangeDir<#1>\@Curve}
```

As a concluding remark, please notice the 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` for a more versatile set of drawing macros; evidently nobody forbids to exploit the full power of the `\cbezier` original macro for cubic splines.

I believe that the set of new macros 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.