# LaTeX 2ε, pict2e and complex numbers

Claudio Beccari
Politecnico di Torino
Turin, Italy
claudio dot beccari (at) polito dot it

**Abstract**

In 2003 the endless list of LaTeX packages was enriched by the package pict2e, intended to substitute for the dummy one that has accompanied every LaTeX distribution since 1994. This package implements everything as stated by Lamport in the second edition of his LaTeX manual (for LaTeX 2ε). But if you explore the inner workings of the new pict2e, you discover the new package has some unexpected potential applications, especially if complex number arithmetic operations are included in it.

## 1 Introduction

The original package pict2e which accompanied the first release of LaTeX 2ε in 1994 was just a dummy package that would simply type out an info message that the *real* package was not yet available. Nevertheless, the LaTeX manual by Leslie Lamport [2] already described the features of this expected package; its primary function was to relieve the strong limitations of the picture environment, mainly due to the fact that graphic objects were realized by means of special fonts which necessarily contained a limited number of "graphic" glyphs.

Anyone who has used the original picture environment in LaTeX may have looked forward to the new pict2e package, so as to be able to draw the usual graphics available with other drawing facilities, even those that are an integral part of commercial and/or open source text processors.

The new pict2e [1] relieves all the limitations of the old picture environment, in particular: the small set of possible inclinations of segments and vectors; the limited number of radii for drawing circles; the rigidity in drawing ovals, whose corners suffered from the limited number of quarter circle arcs; the shortest length of segments and vectors limited to 10pt except for horizontal and vertical ones; the line thickness limited to two values due to the very limited number of special picture fonts; only second order Bézier curves which were made up of small dots partially superimposed on one another.

The new pict2e resorts to the output driver facilities, in the sense that it is dvips or pdf(la)tex[1] that takes care of drawing straight and curved lines, filled and unfilled contours, arrow tips, and the like, with all the facilities offered by the powerful PostScript language, even in its simplified form as used in PDF documents.

Figure 1 shows an example of a set of lines with slopes of 10°, 20°, ..., 80°. The following picture code reflects the usual syntax, with the only exception that line slopes are three digit integers, instead of the relatively prime one digit integers limited to a magnitude of 6 as in the "old" picture environment. The coefficients of the line slopes are simply obtained by rounding to the closest integers the sines and cosines of the angular slopes multiplied by 1000.

```
\unitlength=1mm
\begin{picture}(70,70)
...
\put(0,0){\line(985,174){68.95}}
\put(0,0){\line(940,342){65.80}}
\put(0,0){\line(866,500){60.62}}
...
\put(0,0){\line(342,940){23.94}}
\put(0,0){\line(174,985){12.18}}
\end{picture}
```

Depending on the output driver, pict2e inserts the necessary \special commands with the appropriate syntax, so that when running pdflatex the output PDF file contains the drawings that are directly visible with the PDF viewer. When running latex, the DVI file generally[2] must be processed with dvips to get a PostScript file where the drawings are directly visible with the PostScript viewer, and/or the PostScript file may be processed with ps2pdf to get a self-contained PDF file.

[1] Some other drivers are partially or totally supported.

[2] Several DVI file previewers can interpret the PostScript \specials, but this is not universally true.

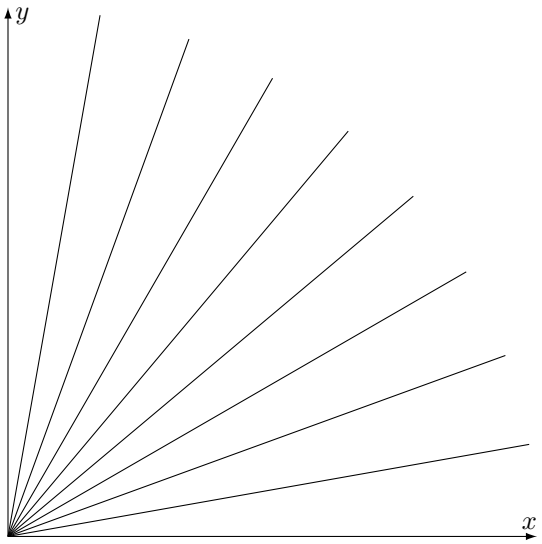


**Figure 1**: Line segments with angular slopes that are multiples of 10°, drawn with `pict2e`

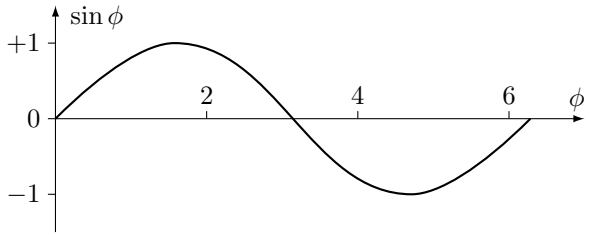


**Figure 2**: A sine wave

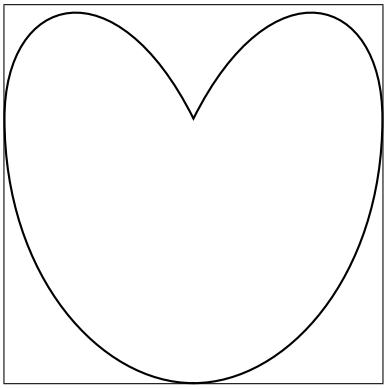

**Figure 3**: A curve containing a cusp

When dealing with `pict2e`, I believe the `latex` + `dvips` + `ps2pdf` procedure is less interesting than the direct production of a PDF file by means of `pdflatex`, because in the former case the author may alternatively use the well-known and more powerful `PSTricks` [3].

I would like to encourage any LaTeX users who may be unaware of the availability of `pict2e` to download the package from CTAN, if necessary, and experiment with the new features. In particular, I would like to draw to the attention of Linux users that TeX distributions coming with some Linux systems are quite out-of-date with respect to CTAN. My own Linux-based distribution (2005/08/15), for example, contains only issue 14 of `latexnews.dvi` dated 2001/06/01, while my updated MiKTeX distribution contains issue 16 dated 2003/12/01. The new `pict2e` was announced in issue 15, also dated 2003/12/01. The current version (as of September 2006) of `pict2e` was updated 2004/08/06. The current version is available in current and future distributions of MiKTeX and TeX Live.

In the following sections I will give some examples of `pict2e` usage, and then describe some enhancements of the package, how to use some internal commands and how to build powerful new commands to draw arbitrary curves by means of third order Bézier curves. I will also need to describe some elementary properties of complex numbers and therefore how to implement complex number arithmetic by means of LaTeX and the underlying TeX macros and primitives.

## 2 Examples

Our first example is an accurate sine wave, as in figure 2. This can be created as follows:

```
\Curve(0,0)<1,1>%            0 deg
      (1.570796,1)<1,0>%   90 deg
      (4.712389,-1)<1,0>% 270 deg
      (6.283185,0)<1,1>%  360 deg
```

where the parentheses contain the curve node coordinates and the angle brackets contain the direction coefficients of the curve tangents at each node.

A diagram with a cusp is shown in figure 3; the code used is the following:

```
\Curve(2.5,0)<1,0>(5,3.5)<0,1>%
      (2.5,3.5)<-.5,-1>[-.5,1]%
      (0,3.5)<0,-1>(2.5,0)<1,0>
```

Another example is given in figure 4 where the `\polyline` macro, described later, is used. The code for generating the heptagon and star vertices is the following:

```
\begin{picture}(5,5)(-2.5,-2.5)
\DividE 360pt by 7pt to\Seventh
\DirFromAngle\Seventh to\Dir
\CopyVect 0,2.5 to\Vone
\MultVect\Vone by\Dir to\Vtwo
\MultVect\Vtwo by\Dir to\Vthree
\MultVect\Vthree by\Dir to\Vfour
\MultVect\Vfour by\Dir to\Vfive
\MultVect\Vfive by\Dir to\Vsix
\MultVect\Vsix by\Dir to\Vseven
```

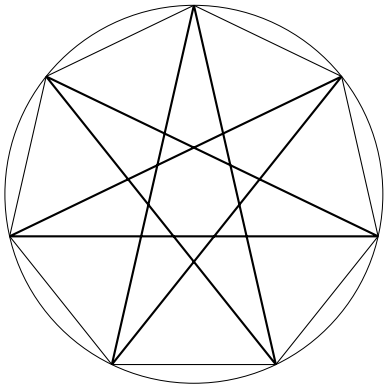

**Figure 4**: Heptagon and seven pointed star

```
\polyline(\Vone)(\Vtwo)(\Vthree)(\Vfour)%
         (\Vfive)(\Vsix)(\Vseven)(\Vone)
\thicklines
\polyline(\Vone)(\Vfour)(\Vseven)%
  (\Vthree)(\Vsix)(\Vtwo)(\Vfive)(\Vone)
\end{picture}
```

## 3 Extensions to the `pict2e` package

The `pict2e` package, according to the description in [2], retains the limitation that the slope parameters of the `picture` segments are represented with integer numbers. According to the authors, Rolf Niepraschk and Hubert Gäßlein, this limitation is due to the specific division routine used, as well as fulfilling the line and vector specifications specified by Lamport.

In a previous paper [4] I complained about the fact that even $\varepsilon$-TEX does not implement real floating number calculations and I invited developers to extend $\varepsilon$-TEX functionality in that direction.

Meanwhile, the LATEX programmer must rely on "poor man" methods. The only TEX object that is representable with a fractional number in the input flow is the *scale factor* used for scaling lengths: when you type

```
\newlength{\dimA} \newlength{\dimB}
\setlength{\dimA}{33.25pt}
\setlength{\dimB}{1.44\dimA}
\showthe\dimB
```

you expect to see on the log file (and on the screen) that the dimension register `\dimB` contains the value of 47.88pt. Actually the log file will exhibit the value of 47.88008pt because of conversion, rounding and truncation errors during the whole process. Here is where the floating point arithmetic would be handy... in the future. But notice that 47.88 is the arithmetic product of the fractional measure in points of the register `\dimA` multiplied by the fractional number 1.44. Multiplication is then relatively an easy task provided we can convert back and forth fractional numbers and dimensions.

The trick is easy, and despite being classified as "dirty" in *The TEXbook* [5, page 375], it has been used by almost everyone needing to use this poor man approach to fractional number multiplication.

Division is trickier because it can produce overflows (like multiplication), the division by zero error, and it does not have any relation to scale factors, the only objects that TEX can use as multipliers.

Integer division is generally unusable and so the routine Gäßlein and Niepraschk used accepts integer dividend and divisor transformed into lengths, but yields a length whose measure in points is the required fractional quotient.

At the time `pict2e` became available I had been using a division routine for several years; it was part of a package of mine that was never published. The good point is that I had been using that package for years and that routine always worked reliably, although no controls were actually performed to avoid overflows or divisions by zero (they could be possibly be done before calling the routine). This routine actually divides two lengths and yields their ratio as a fractional signed decimal number. The code may be seen in figure 5.

TEX programming was used together with some plain TEX macros that are also available in the kernel of LATEX. I chose to use the delimited argument facility of TEX, which is not available in LATEX, because coding becomes more readable; the funny choice of the name with initial and final capitals has a long and insignificant history, but I did not want to change it here, for the sake of avoiding contradictions. For the same reason I did not translate `\segno` into, say, `\Sign`, but I suppose that its meaning is understandable by everybody. In practice `\DividE` implements a long division between the numerator and denominator lengths translated into scaled points (this is what TEX does when a counter is assigned a length value) stored into two numerical counters; at every iteration the remainder is multiplied by ten and the single digit new quotient `\q` is appended to the overall quotient `\Q`.

The only test I added to this last version of the routine is to assign a positive maximum TEX value to the quotient in case of division by zero, so that the best TEX approximation to infinity is used.

With this division routine at hand, we can extend the slope argument of segments to any reasonable fractional number; the `\line` of `pict2e` may be changed to the code in figure 6.

Some comments are in order because some of

```
\def\DividE#1by#2to#3{%
    \begingroup
    \dimendef\Numer=254\relax \dimendef\Denom=252\relax
    \countdef\Num 254\relax \countdef\Den 252\relax \countdef\I=250\relax
    \Numer #1\relax \Denom #2\relax
    \ifdim\Denom<\z@ \Denom -\Denom \Numer -\Numer\fi
    \def\segno{}\ifdim\Numer<\z@ \def\segno{-}\Numer -\Numer\fi
    \ifdim\Denom=\z@
      \ifdim\Numer>\z@\def\Q{16383.99999}\else\def\Q{-16383.99999}\fi
    \else
      \Num=\Numer \Den=\Denom \divide\Num\Den
      \edef\Q{\number\Num.}%
      \advance\Numer -\Q\Denom \I=6\relax
      \@whilenum \I>\z@ \do{\DividEDec\advance\I\m@ne}%
    \fi
    \xdef#3{\segno\Q}\endgroup
}%

\def\DividEDec{\Numer=10\Numer \Num=\Numer \divide\Num\Den
            \edef\q{\number\Num}\edef\Q{\Q\q}\advance\Numer -\q\Denom}%
```

**Figure 5**: Another division routine for fractional values

the code may seem redundant. The `\line` macro behaves as in the original `pict2e`, except that the "only" argument `#1` actually has the usual format of two fractional or integer numbers separated by a comma; this is the form I will give to the representation of complex numbers; the `\line` macro does not actually need this machinery, but since the necessary macros are already there, why not?

The `\DirOfVect` macro takes the two direction comma-separated coefficients passed in argument `#1`, interprets them as the horizontal and vertical components of a vector and determines the directing cosines, or, if you prefer, normalizes these two vector components to the length of the vector itself, so that they are both fractional numbers whose magnitude does not exceed unity. This is good for the following operations and division calculations. The rest of the macro is very similar to the original one. But it may be observed that the above normalization does not depend on the integer or fractional nature of the directional coefficients; it even neglects the fact that their magnitude may be larger than 1000, which is the last constraint remaining in the original `pict2e \line` macro.

Of course these coefficients should not be too large, even though the length of the vector computation implies some powers of two and a square root; computations are made in such a way as to extract from the root the largest of the two components, so that a number not exceeding unity gets squared and the radicand never exceeds 2. The `\ModOfVect` macro actually executes this square root and I have never observed any deficiency in its calculations.

This extension suggests another one; since the direction coefficients may be of any reasonable magnitude, why should we maintain the `picture` syntax for defining lines, where along with the direction coefficients it is necessary to specify the horizontal projection of the segment? Why not define the line with its absolute horizontal and vertical components? Everything would be much cleaner and the execution time would be much shorter. Thus, I defined an alternative line description, as follows:

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

where the arguments passed to the macro represent the actual components of the segment, and no length is specified. With `\put` you put the segment origin as usual and the `\Line` macro does the rest. The "`moveto`", "`lineto`" and "`stroke`" keywords are those used in PostScript and in many descriptive graphic languages; these are some of the new keywords introduced by `pict2e` and they may induce a small revolution in considering graphics with LaTeX.

For example it is possible to define a macro for tracing a polygonal line joining an arbitrary number

```
\def\line(#1)#2{\begingroup
  \@linelen #2\unitlength
  \ifdim\@linelen<\z@\@badlinearg\else
    \expandafter\DirOfVect#1to\Dir@line
    \GetCoord(\Dir@line)\d@mX\d@mY
    \ifdim\d@mX\p@=\z@\else
       \ifdim\d@mX\p@<\z@ \@tdB=-\p@\else\@tdB=\p@\fi
       \DividE\@tdB by\d@mX\p@ to\sc@lelen \@linelen=\sc@lelen\@linelen
    \fi
    \pIIe@moveto\z@\z@
    \pIIe@lineto{\d@mX\@linelen}{\d@mY\@linelen}%
    \pIIe@strokeGraph
  \fi
\endgroup\ignorespaces}%

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

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

\def\DirOfVect#1to#2{\GetCoord(#1)\t@X\t@Y
  \ModOfVect#1to\@tempa \DividE\t@X\p@ by\@tempdimc to\t@X
  \DividE\t@Y\p@ by\@tempdimc to\t@Y
  \MakeVectorFrom\t@X\t@Y to#2\ignorespaces}%

\def\ModOfVect#1to#2{\GetCoord(#1)\t@X\t@Y
  \@tempdima=\t@X\p@ \ifdim\@tempdima<\z@ \@tempdima=-\@tempdima\fi
  \@tempdimb=\t@Y\p@ \ifdim\@tempdimb<\z@ \@tempdimb=-\@tempdimb\fi
  \ifdim\@tempdima>\@tempdimb
     \DividE\@tempdimb by\@tempdima to\@T
     \@tempdimc=\@tempdima
  \else
     \DividE\@tempdima by\@tempdimb to\@T
     \@tempdimc=\@tempdimb
  \fi \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
  \Numero#2\@tempdimc
\ignorespaces}%

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

**Figure 6**: Redefinition of the `\line` macro using the new division routine

of nodes as such, as in \polyline here:[3]

```
\def\polyline(#1){\beveljoin
  \GetCoord(#1)\d@mX\d@mY
  \pIIe@moveto{\d@mX\unitlength}%
              {\d@mY\unitlength}%
  \p@lyline}%

\def\p@@lyline(#1){%
  \GetCoord(#1)\d@mX\d@mY
  \pIIe@lineto{\d@mX\unitlength}%
            {\d@mY\unitlength}%
  \p@lyline}%

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

\def\p@lyline{%
  \@ifnextchar\lp@r{\p@@lyline}%
      {\pIIe@strokeGraph\ignorespaces}%
}%
```

This simple macro \polyline would be rather difficult to realize without the moveto, lineto and stroke keywords.

A final small improvement consists in setting the shape of the line terminators; by default they are square caps but when tracing thick lines that meet at the same point it is better to set them round. See figure 7 for a comparison. The following code gives access to these settings:

```
\ifcase\pIIe@mode\relax
\or %PostScript
 \def\roundcap{\special{ps:: 1 setlinecap}}%
 \def\squarecap{\special{ps:: 0 setlinecap}}%
 \def\roundjoin{\special{ps:: 1 setlinejoin}}%
 \def\beveljoin{\special{ps:: 2 setlinejoin}}%
\or %pdf
 \def\roundcap{\pdfliteral{1 J}}%
 \def\squarecap{\pdfliteral{0 J}}%
 \def\roundjoin{\pdfliteral{1 j}}%
 \def\beveljoin{\pdfliteral{2 j}}%
\fi
```

I prefer to have the round cap version as the default setting, but this is a question of personal taste. Apparently these settings, set up by means of the special programming language of the destination file, are global ones so it is necessary to countermand them once the default has to be restored; it is not possible to rely on groups in the usual TeX way. I also have the impression that at each closing of a picture environment any setting is lost; I do not know the PostScript language well enough to understand if some internal pict2e command executes this reset, but after all it is no trouble to reset the preferred settings at the beginning of each picture.

[3] Of course with delimited arguments it is not possible to use the LaTeX macro definition commands.

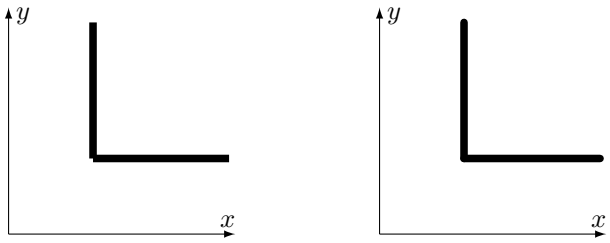


**Figure 7**: Square and round caps

It happens that the line terminator choice does not work with the original pict2e \line definition when the drawn segments are purely horizontal or vertical, while it does work with my redefinition, as can be seen in figure 7. After all the original pict2e definition of \line mimics the original "LaTeX 2.09" one, where it was important to avoid drawing lines by means of the special graphic fonts when horizontal and vertical lines could be more easily and efficiently drawn with the low level DVI commands TeX uses for vertical and horizontal rules. When lines are drawn with the device driver facilities it is no longer necessary to make horizontal and vertical lines special cases.

I should remark that several other graphic packages are available; among them the curves package by Ian Maclaine [6] certainly is the first one that might benefit from these new facilities introduced by pict2e. There is also the package bundle pgf by Till Tantau [7]; PGF stands for "portable graphic format" and its intention is to provide LaTeX with a portable set of macros providing nearly as much as PSTricks, even when running pdflatex. The latter program is at the base of the excellent presentation document class beamer and is certainly worth using because of its fine properties. In the event, I did not extend pgf because I found some difficulties in writing some macros, such as to relocate output to specified coordinates. Moreover I believe that pict2e, although much simpler then pgf, is part of LaTeX, not a major extension as pgf is.[4]

## 4 Complex numbers

As has partially been seen, drawing implies treating *directions*; in particular, it is necessary to manipulate vectors and their directions. METAFONT [8], the program for drawing fonts written by Knuth himself, treats all these objects with complex numbers. Knuth hardly ever cites complex numbers in

[4] With pict2e I had no difficulties rewriting the macros of a package of mine for drawing electronic circuits; I was not able to do the same with pgf; of course the one to blame is just myself.

*The METAFONTbook*, but all the inner and outer workings are done by means of *pairs* that are nothing else but complex numbers. The manipulation of directions and angles is always done in an alternating change from Cartesian to polar representation of complex numbers; here and there some of the operations available on pairs are explicit complex number operations.

Most people don't know or don't like complex numbers; perhaps this is due to the fact that they contain imaginary quantities, something far away from the everyday reality of numbers.

Mathematicians, on their side, usually do little to ease students' learning of complex numbers, and with their love for abstraction and generalization they sometimes miss conveying the message that these entities are nothing more than "scale-rotate" operators: they simply scale an object up and down and rotate it around a pivoting point. Almost everybody is familiar with these operators, from using one of the many interactive drawing programs, even relatively simple ones.

To see this, take a vector $\vec{v}$ drawn from the origin of a Cartesian plane defined with axes $x$ and $y$; if you project the above vector on the $x$ axis you get the horizontal component $\vec{v}_x$, while if you project it on the $y$ axis you get the vertical component $\vec{v}_y$. If you define two unit vectors, $\vec{u}_x$ parallel to the $x$ axis and pointing to increasing $x$ values, and similarly $\vec{u}_y$ for the $y$ axis, you can separate in every component the information of its magnitude from that of its direction and you can write

$$\vec{v} = \vec{v}_x + \vec{v}_y = v_x\vec{u}_x + v_y\vec{u}_y$$

Now let us emphasize the link the vector $\vec{v}$ has with the unit direction along the $x$ axis by writing

$$\vec{v} = [v_x + (\vec{u}_y/\vec{u}_x)v_y]\vec{u}_x$$

so that we may interpret the contents of the square brackets as the operator that acts on the unit $x$ vector, by scaling it according to the magnitude of $\vec{v}$, and by rotating it by a certain angle, the angle of $\vec{v}$ with respect to the $x$ axis. The contents of the square brackets have the same characteristics as those we anticipated for complex numbers.

The ratio $\vec{u}_y/\vec{u}_x$ is generally given the name of 'i' by the mathematicians and 'j' by most technologists.[5] Its application has the geometric meaning of changing the unit vector $\vec{u}_x$ to the unit vector $\vec{u}_y$, i.e. rotating the unit vector $\vec{u}_x$ 90° counterclockwise.

If we apply twice in a row the 90° rotation operator 'i' to the unit vector $\vec{u}_x$, producing a total rotation of 180°, we get the renowned expression

$$\mathrm{i}\,\vec{u}_y = \mathrm{i}(\mathrm{i}\,\vec{u}_x) = \mathrm{i}^2\,\vec{u}_x = -1\vec{u}_x$$

that is

$$\mathrm{i}^2 = -1 \qquad \text{or} \qquad \mathrm{i} = \sqrt{-1}$$

which induced XVI century mathematicians to call 'i' the *imaginary unit*.

If we further process the above results, we get

$$v_x + (\vec{u}_y/\vec{u}_x)v_y = v_x + \mathrm{i}\,v_y = |\vec{v}|\left(\frac{v_x}{|\vec{v}|} + \mathrm{i}\,\frac{v_y}{|\vec{v}|}\right)$$

where

$$|\vec{v}| = \sqrt{v_x^2 + v_y^2}$$

We recognize that if the original vector $\vec{v}$ is inclined by an angle $\theta$ counterclockwise with respect to the $x$ axis, then the two above fractions represent the cosine and sine of such an angle

$$\frac{v_x}{|\vec{v}|} = \cos\theta$$
$$\frac{v_y}{|\vec{v}|} = \sin\theta$$

The scaling factor of the operator acting on the unit $x$ vector is $|\vec{v}|$ and the direction of the $x$ unit vector is changed by the angle $\theta$ counterclockwise. The operator is actually a scale-rotate operator, i.e. a complex number.

If we apply two scale-rotate operators in a row to the unit $x$ vector, we make the following observations:

1. the two scaling factors behave as two multipliers and are commutative;
2. the two rotation angles add up and are commutative;
3. the total effect produced by the two operators is therefore equivalent to that of a single operator whose magnitude is the product of the two magnitudes and whose angle is the sum of the two angles.

In order to represent such effects with the operation of multiplication it is advisable to use magnitudes as regular factors, and to use angles as exponents of a suitable base; the mathematicians tell us that a scale-rotate operator of magnitude $a$ and of angle $\theta$ can be represented as

$$a(\cos\theta + \mathrm{i}\,\sin\theta) = a\,\mathrm{e}^{\mathrm{i}\,\theta}$$

which is called Euler's formula. There are many serious reasons for choosing 'e' as the base and for representing the exponent as an imaginary quantity, but we are not concerned here with them; we simply note that given two scale-rotate operators $a\exp(\mathrm{i}\,\theta)$ and $b\exp(\mathrm{i}\,\phi)$ their total effect is $(ab)\exp[\mathrm{i}(\theta+\phi)]$.

This observation together with Euler's formula lets us understand the meaning of division by a complex number, i.e. a scale-rotate operator; in fact, the

[5] Notice that this mathematical *operator* is not a variable and therefore according to international standards must be written with an upright font.

division is nothing but the inverse operator of a multiplication, and this can be expressed as

$$\left[a\,\mathrm{e}^{\mathrm{i}\,\theta}\right]^{-1} = \frac{1}{a}\,\mathrm{e}^{-\mathrm{i}\,\theta}$$

where we observe that the scaling factor is simply the reciprocal of the multiplicative one, while the rotation term is just in the opposite direction relative to the multiplicative one.

The scale-rotate interpretation of complex numbers also lets us understand very easily the meaning of addition and subtraction of such entities — which end up being the same as addition and subtraction of vectors. Moreover the vector notation becomes redundant, since the scale-rotate operators always act on the unit $x$ vector, which can thus be taken for granted and omitted from the complex number expressions. These expressions therefore maintain the meaning of vector operations *and* of complex number relationships.

For our present purposes, let us emphasize that $\exp(\mathrm{i}\,\theta)$ has unit magnitude, and therefore contains only the information on the direction. Furthermore, $\exp(-\mathrm{i}\,\theta)$ represents a rotation in the opposite direction; if we have the means of multiplying by such factors we can change the direction of any vector the way we like, either counterclockwise or clockwise.

For TeX arithmetic it is better to use simple multiplications without exponentials, but Euler's formula lets us change back ad forth from the exponential form to the Cartesian one; the exponential form gives us an easy interpretation of the rotating effects while the Cartesian form gives us an easy mechanism for executing the complex multiplication and therefore the required rotation.

## 5 Complex number TeX macros

I am not going to include here the code for every complex number operation [9]; let me just list the macro names of the operations I wanted to realize, with some explanations to clarify detail.

Notice that I decided to maintain most if not all fractional numbers in control sequences. I also use control sequences to pass complex values to the macros, so that in order to operate on the complex number parts a macro (`\GetCoord`) is needed to separate them, and another (`\MakeVectorFrom`) to reassemble them.

Most macros have delimited arguments, with the main command is followed by the sequence of arguments separated by keywords; rarely, arguments must be enclosed in the traditional curly braces, as normally necessary in LaTeX. For example in order to extract the magnitude (modulus) and the direction from a given vector the macro name is `\ModAndDirOfVect` but the various arguments are separated by the words `to` and `and` so that a typical call might be

```
\ModAndDirOfVect\VectorA to\ModA and\DirA
```

In this context the word "vector" is synonymous with complex number or scale-rotate operator; the word "direction" refers to a complex number with unit magnitude so that the scaling factor is unity.

In the following list of macros the parameters `#1`, `#2`,... are the arguments passed to the various macros. The macro names are assumed to be self explanatory.

```
\SinOf#1to#2
\CosOf#1to#2
\TanOf#1to#2
\MakeVectorFrom#1#2to#3
\CopyVect#1to#2
\ModOfVect#1to#2
\DirOfVect#1to#2
\ModAndDirOfVect#1to#2and#3
\GetCoord(#1)#2#3
\DistanceAndDirOfVect#1minus#2to#3and#4
\XpartOfVect#1to#2
\YpartOfVect#1to#2
\DirFromAngle#1to#2
\ScaleVect#1by#2to#3
\ConjVect#1to#2
\AddVect#1and#2to#3
\SubVect#1from#2to#3
\MultVect#1by#2to#3
\MultVect#1by*#2to#3
\DivVect#1by#2to#3
```

The list ends with the usual four arithmetic operations performed on any mathematical entity; the variant of the multiplication that contains an asterisk performs the multiplication of the first operand by the *complex conjugate* of the second operand; the complex conjugate of a complex number is just the scale-rotate operator where the rotation direction has been reversed.

The above list starts with the usual trigonometric functions. Actually, I stated earlier that arithmetic in TeX should be done with numbers and directions; angles, that are so expressive in Euler's formula, should be avoided. Nevertheless, at some point it's necessary to convert angles to their sines and cosines, but switching back and forth from the Euler representation to the Cartesian one implies the computation of both direct and inverse trigonometric functions. TeX can do both operations (with acceptable approximations) but it slows down quite a bit with frequent transformations. I implemented the computation of the direct trigonometric functions of angles *in degrees*, not in radians, by means of the continued fraction expansion of the half angle

tangent and the parametric formulas:

$$\begin{aligned}\sin\theta &= \frac{2\tan x}{1+\tan^2 x}\\ \cos\theta &= \frac{1-\tan^2 x}{1+\tan^2 x}\\ \tan\theta &= \frac{2\tan x}{1-\tan^2 x}\end{aligned}$$

where

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

and $x = \theta/114.591559$ is the half angle in degrees converted to radians.

This iterative formula for the tangent is quite fast and its precision is remarkable if we consider the modest performance of TEX calculations with fractional numbers. I decided to stop the continued fraction with the term containing the coefficient '11'; probably it is a little too much for TEX capabilities but I prefer to perform one extra cycle than to miss the target.

Unfortunately I could not find similar fast algorithms for the inverse trigonometric functions; I decided to avoid using such inverse functions. METAFONT implements both algorithms, but METAFONT is not TEX: the former was designed to perform fractional number calculations (although represented in fixed radix notation) while the latter was designed for efficiently typesetting text, with calculations reduced to integer operations with some simple tricks to cope with the necessity of fractional "factors".

The macro `\DirFromAngle` is the only one that uses trigonometric functions; further on, just direction vectors are used.

## 6 Circular arcs

`pict2e` implements only the drawing commands specified by Lamport in [2]; it can draw full circles or quarter circles but it cannot draw arcs of any other specified angle amplitude. Or better: it cannot draw them because of the lack of user commands, but it has all the potentialities.

Suppose we want to draw an arc by specifying its center, its starting point and its angle amplitude. The center and the starting point may be absolute coordinates in the `picture` environment space or may be relative to the position specified with a `\put`.

With `pict2e` we can resort to third order Bézier curves as it is done in METAFONT; if the third order curve is not misused, it can approximate up to a half

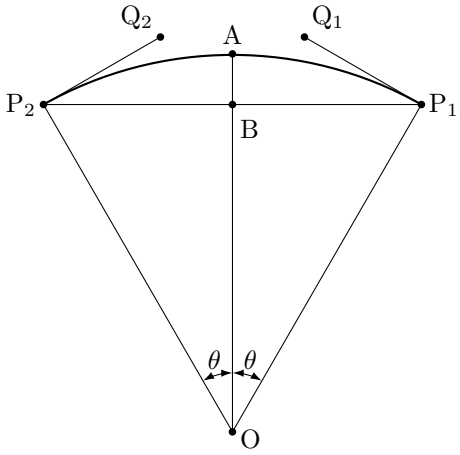


**Figure 8**: Circular arc elements

circle with remarkable precision. The problem is to find the arc end point and the correct control points of the Bézier curve.

With reference to figure 8 it is a simple exercise, given the center O, the starting point $P_1$, and the arc angle $2\theta$, to determine the coordinates of the end arc point $P_2$ and the two control points $Q_1$ and $Q_2$.

For the end point $P_2$ it suffices to take the vector $\overrightarrow{P_1 - O}$ and rotate it about the center point by the given angle $2\theta$; this operation is simplified by the complex number arithmetic described above.

A little trickier is the determination of the control points. It is evident that they lie on the segments perpendicular to the vectors $\overrightarrow{P_1 - O}$ and $\overrightarrow{P_2 - O}$, but how long are the vectors $\overrightarrow{Q_1 - P_1}$ and $\overrightarrow{Q_2 - P_2}$? To answer, it is necessary to know the cubic Bézier equation:

$$P = P_1(1-t)^3 + 3Q_1 t(1-t)^2 + 3Q_2 t^2(1-t) + P_2 t^3$$

that can be found (using other symbols) in *The METAFONTbook*.

P is the generic point on the curve; the start and end points and the control points form the coefficients of the equation; $t$ is a parameter that runs from 0 to 1 while P moves from $P_1$ to $P_2$. The above equation in reality represents the pair of equations obtained when the point coordinates are substituted; therefore it represents the pair of parametric equations that describe the curve in the usual $xy$ Cartesian plane.

If we move the origin of the coordinates to point B of figure 8 and exploit the obvious symmetry, the similar triangles, the Bézier equation, and the fact that point A must be distant from O just as $P_1$ and $P_2$, it turns out that the length $K$ of the required

vectors is

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

where $R$ is the arc radius, the length of the vector $\overrightarrow{P_1 - O}$. Again, with the complex number operations we have at our disposal, it is straightforward to exploit the information we have to trace the cubic Bézier curve from $P_1$ to $P_2$; the result is indistinguishable from a true circle when the total arc angle does not exceed 90° and is not noticeable with the naked eye when the total arc angle does not exceed 180°.

Therefore a good `\Arc` macro should check the amount of the total arc angle and possibly split the total arc into sub-arcs none of which exceeds a half circle, or, even better, a quarter circle. This is why in actual computations it is much better to measure angles in sexagesimal degrees than in radians; with radians the reduction of angles by amounts corresponding to quarter or half circles intrinsically requires approximation due to the irrational nature of $\pi$; TEX introduces its own approximation errors, so let us not contribute with further ones.

## 7 General curves

The abovementioned package `curves` [6] by Ian Maclaine offers the user the possibility of tracing arbitrary curves by stating just the curve nodes; METAFONT is entirely built on this possibility, although it uses much finer mathematics and it offers the user the opportunity to optionally specify node tangents and arc tensions.

From the user's point of view these differences are great by themselves, but there is another important difference: `curves` uses quadratic Bézier curves, while METAFONT uses cubic ones. This produces dramatic differences when the curve nodes imply the presence of inflection points. In this case the algorithm devised by Maclaine more often than not produces anomalous loops; such loops are very rare with cubic Bézier curves — it is necessary to work hard just to find examples of such loops.

Of course Maclaine had to sacrifice some graphical functionality in favor of simpler mathematics, which, as we know, is not TEX's best feature.

I tried to devise a chain of macros that trace one cubic Bézier arc at a time, and pass one another the end point tangent directions. These macros are

```
\StartCurveAt#1WithDir#2
\CurveTo#1WithDir#2
\CurveFinish
```

where the first argument is a point coordinate pair (a complex number) and the second argument is a direction (a complex number with unit magnitude). The first macro initializes the process and memorizes the first point direction; the second macro gives the destination program the necessary information on the arc nodes and control points, and the third macro eventually strokes the curve with the syntax of the destination program.

The first macro basically uses the `moveto` keyword, the second macro `curveto`, and the final macro `stroke`; we have already partially seen these keywords while discussing lines and polylines. The first and second macros also normalize the directions given, so the end user does not need to make preliminary calculations in order to normalize the direction magnitude. They also memorize the specified and normalized direction for the benefit of the next `\CurveTo` call.

The `\CurveTo` macro is the one that has to do the main work in determining the position of the control points. Obviously it must start by checking the trivial situations where the directions form zero or 180° angles with the arc chord; it must also distinguish the situations where the tangents form 90° angles with the chord. It must behave correctly even if the end nodes and the directions imply an inflection point. But in most cases it has to deal with normal situations where the control point directions relative to the respective end points are given but the distances of the control points from the nodes must be determined.

There is a great margin for arbitrary decisions. I decided to divide the chord in two parts that are more or less proportional to the projection of the directions on the chord and to determine the distance $K$ of each control point from its neighboring node with the same formula as for circular arcs. The chord fraction is treated as half the chord of a circular arc and the corresponding radius is determined so as to use the mentioned formula. This choice is totally arbitrary but, as it is easily understandable, it is a reasonable one.

This done, the usual complex number arithmetic can be used to locate the position of the control point and to give the internal command for instructing the destination program how to draw the desired curve. In figure 2 there is a simple example of a sine curve that has been drawn with three arcs: from the origin to the maximum, from this point to the minimum, and lastly from the minimum to the end of the cycle.

Actually the three above macros are the ingredients of the general macro `\Curve` that operates on an arbitrary number of couples of nodes and directions:

`\Curve(`⟨$P_0$⟩`)<`⟨$dir_0$⟩`>(`⟨$P_1$⟩`)<`⟨$dir_1$⟩`>...`
`(`⟨$P_n$⟩`)<`⟨$dir_n$⟩`>`

whose code is the following:

```
\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}
```

For each node it is necessary to specify the direction of the tangent to that node, but these tangent direction coefficients need not be normalized. They are normally given within angle brackets. If there is a cusp, the tangent changes abruptly so that a new direction must be specified before continuing to draw the curve; this "optional" change in direction is indicated with a direction enclosed in square brackets; see the code implementing the heart shape in figure 3.

In the light of that example it is not a burden to specify all the directions at each node, although a simpler syntax such as that used in METAFONT would be desirable.

## 8 Conclusion

I wanted to illustrate the use of fractional number TEX arithmetic applied to complex numbers. These are formidable tools for graphics applications and the necessary macros are actually within the range of every TEXnician; there is no need to be a guru.

I hope the ideas I gave here may be exploited better than I can do for extending the existing graphic packages so as to get the best from the `pict2e` package. This package in particular may benefit from some simple extensions, or may incorporate the user macros for drawing circular arcs and arbitrary curves, possibly even with the filling capabilities that are being offered by other programs.

There might even be some expert programmer who feels challenged to write a user graphical interface that exploits the suggested extensions.

The `pict2e` package may still have minor glitches,[6] but even right now it opens many possibilities that were unthinkable when the standard LATEX 2.09 `picture` environment provided the only native graphics for LATEXers. They eventually had to give up and move to other dedicated programs; these are fine, but they do not necessarily produce completely compatible code and generally, with some remarkable exceptions (such as `pgf`), require special treatment in order to insert the same fonts used in the main text.

I hope the features described here will be included in future releases of the mentioned packages. Until that time, I have made a package `curve2e` available from CTAN [9] for anyone who wishes to make use of them.

## References


[1] Gäßlein H. and Niepraschk R., *The `pict2e` package*, PDF document attached to the "new" `pict2e` bundle; the bundle may be downloaded from CTAN.

[2] Lamport L., *LATEX: A Document Preparation System*. Addison Wesley Publishing Co., Reading, Massachusetts, 1994.

[3] van Zandt T., *`PSTricks`*, CTAN. See also `http://www.tug.org/PSTricks` for further documentation and examples.

[4] Beccari C., *Floating point numbers and Metafont, MetaPost, TEX, and PostScript Type 1 fonts*, *TUGboat* 23:3/4, 2002, pp. 261-269.

[5] Knuth D.E., *Computers & Typesetting volume A: The TEXbook*, Addison Wesley Publishing Co., Reading, Massachusetts, Millennium Edition.

[6] Maclaine I., *`curves`* and *`curvesls`*, CTAN.

[7] Tantau T., *User's Guide to the PGF Package*, included in the `pgf` bundle downloadable from CTAN.

[8] Knuth D.E., *Computers & Typesetting volume C: The METAFONTbook*, Addison Wesley Publishing Co., Reading, Massachusetts, Millennium Edition.

[9] Beccari C., *`curve2e`*, CTAN.


[6] With the version I have at hand, `pict2e` traces vectors a little bit thicker than segments, although the line thickness is maintained constant; with my redefinition of `\vector` this glitch appears to be corrected.