ePiX Tutorial

Andrew D. Hwang
ahwang@mathcs.holycross.edu

Version 0.8.1, June 6, 2002

1 Introduction

ePiX is a LaTeX pre-processor that creates mathematically accurate (mostly 2-dimensional) plots and figures using easy-to-learn syntax. The user interface is superficially that of LaTeX itself: You prepare a short input file and "run ePiX" on this file, which produces a text file that is included into a LaTeX document. Because the output is plain text, the output can be edited manually if necessary. However, for most visual tweaking it is easier and safer to change the source and re-run ePiX.

There are several reasons to use ePiX.

- Ease of use: ePiX was written by a mathematician for mathematicians. Figure objects are specified by simple, mnemonic commands, as in LaTeX. The learning curve is gentle: Using only built-in primitives, you can duplicate almost everything xfig can do, and do many things xfig cannot.

- Quality of output: ePiX creates publication-quality, mathematically accurate figures whose appearance matches that of LaTeX. Paragraph-mode LaTeX typography may be put in an ePiX figure as easily as in an ordinary LaTeX document; this is one of the strongest advantages of ePiX over plain PostScript, and indeed one of ePiX's distinguishing features.

- Economy of storage and transmission: ePiX *output* is native LaTeX, and is generally 50–70% smaller than comparable Postscript. Some *source* files (vector fields, for example) can be considerably less than 1% the size of their output, making ePiX a potentially non-trivial form of compression for documents containing many complicated figures.

- Flexibility: In ePiX, you refer almost exclusively to Cartesian coordinates, letting the software handle conversion to LaTeX picture coordinates. Resizing a printed figure or its Cartesian bounding box is a matter of changing a couple of numbers and re-running `ePiX`; you needn't re-calculate or re-type the LaTeX coordinates of all your picture objects. ePiX's mechanism for placement of text in a figure is easy, accurate, and robust under changes of scale.

- Power: `ePiX` retains the power of `C` as a programming language; variables, loops, and recursion can be used to draw complicated plots and figures with just a few lines of input. Objects' locations can be specified in terms of variables, allowing you to rearrange or otherwise modify a figure by changing a few numbers. You can plot finite sums of functions (e.g., Taylor and Fourier polynomials), generate successive "snapshots" of a figure as parameters vary, or create stereoscopic pairs, for example.

- It is *Free Software*, in the senses laid out by the Free Software Foundation: You are granted the right to use the program for whatever purpose, and to inspect, modify, and re-distribute the source code, so long as you do not restrict the rights of others to do the same. In short, the license is similar to the terms under which theorems are published, rather than the way commercial software is distributed. `ePiX` also happens to be free (no-cost) software.

A graphical interface is not planned, because it is impossible to achieve the same accuracy as with an input file, and because graphical menus do not encourage logical structuring of a figure. Output files, by contrast, may be previewed with any dvi-capable previewer.

Comparison with Existing Programs

There are already programs for drawing figures that can be included in LaTeX documents; why write another one? The short answer is that the author was unable to find a utility that is Free (in the sense of Free Software), mathematically accurate, easy to use, and sufficiently capable. It is difficult to explain the rationale in detail without critiquing existing software by name. Nothing in this section is meant to disparage the good work of others, but only to explain why the author felt the need to write `ePiX`.

`xfig`, while a fast and convenient way of drawing simple figures, is analogous to a WYSIWYG word processor, in the same way `ePiX` is analogous to LaTeX. While the immediate results are often pleasing, it is difficult to achieve mathematical accuracy, and even more difficult to edit an existing figure. Similar remarks are true for other programs, such as `TeXpict` and `sketch`, whose user interfaces are graphical menus.

Large, commercial packages such as `Maple` and `Mathematica` produce mathematically accurate output, but are not Free (nor very affordable for an individual), and are therefore (in the author's view) contrary to the academic ethic. There are many smaller reasons to be unhappy with such programs. They are also, arguably, overkill for `ePiX`'s intended purposes.

`gnuplot` might seem a natural choice, but despite its name, `gnuplot` is not Free Software. In addition, the author found the documentation dense, the learning curve steep (likely an indictment of the author's patience rather than of `gnuplot`), and the output less than perfect. Packages such as `picTeX` and `pstricks` can do many useful things, but do not plot functions.

Finally, the author found a number of "toy" programs scattered on the web, but none were powerful, flexible, or easy to compile (much less use), and none produced publication-quality output.

System Requirements

In contrast to the previous section, which might be titled, "Why to use `ePiX`", this section might be called, "Why not to use `ePiX`", depending on your operating system and available software.

Running `ePiX` requires, in decreasing order of importance, a `C/C++` compiler, GNU `bash` or a similarly functional shell ("command line") or scripting language, and an operating system that supports output redirection. For creating `ePiX` input files, a text editor (such as `emacs` or `vim`) that facilitates formatting `C` code is extremely useful. To incorporate `ePiX` figures in a document and preview the result, you will of course need LaTeX itself—particularly the `epic` and `eepic` packages, but also the `pstricks` and/or `color` packages for color output—and a `dvi` or Postscript previewer. These components are standard GNU[1] software, and should also be available on most Unix mainframes. If not, speak to your sysadmin.

Typographical Conventions

Shell commands, file names, and other text literals are in `typewriter` font. Inside an `ePiX` source file, commands must be typed exactly as shown, in-

[1] "GNU's Not Unix", an operating system usually distributed with the Linux kernel.

cluding (or omitting!) final semicolons and "double quotes".

Installing ePiX

ePiX is distributed as source code. The latest version of ePiX can be downloaded from

```
http://mathcs.holycross.edu/~ahwang/current/ePiX.html
```

Unpack the gzip-ed tar file

```
tar -zxvf epix_src.tar.gz
```

or, if your tar doesn't know about decompression,

```
gunzip epix_src.tar.gz
tar -xvf epix_src.tar
```

cd to the source directory, which is named epix-0.8.x for some small integer x. The README file contains detailed installation instructions. If you're impatient, the short of it is

```
make test
[make contrib]
make install
make clean
```

Respectively, these steps build the epix library and run a test compile on the included sample files; optionally build extra packages (see below); install the library, header file, and two shell scripts; and revert the source directory to its original state. The only optional package at present was kindly contributed by Svend Daugaard Pedersen, and supplies extensions for enhanced Cartesian coordinate systems, and for hatching planar regions. His package is documented separately, in the contrib/ subdirectory of the source package.

By default, ePiX installs in subdirectories of /usr/local; if you do not have root access, see the README for information on personal installation and POST-INSTALL for instructions on setting your PATH variable so your shell can find ePiX.

ePiX is not a stand-alone program, but consists of a C/C++ library and a shell script, and therefore *requires a compiler for normal use.* The GNU compilers (gcc/g++) and C library are strongly preferred, both because they are used to develop `ePiX`, and because they implement many mathematical features not specified by `ANSI C`.

Your life with `ePiX` will be difficult (in the Japanese sense) unless the GNU shell `bash` is available on your system or you hack the scripts. If you have no idea what this means, don't worry; most Unices have `bash` installed. If you port `ePiX` to another operating environment, please send the author copies of your scripts for future inclusion. $\ddot\smile$

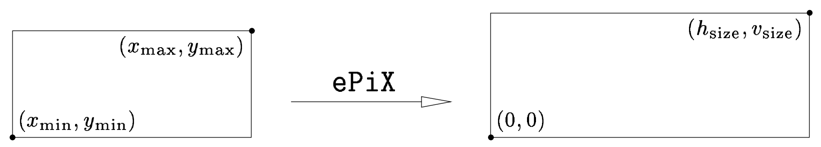
2 Getting Started

An `ePiX` source file is a short `C` program. Even if you speak `C`, you may want to skim this section quickly, as it explains the basic format of an input file. This section assumes you know nothing of `C`, but that you *do* know LaTeX.

Coordinates and Dimensions

As a mathematician, you generally want to depict a specific portion of the Cartesian plane, and may want to control the aspect ratio of the figure, for reasons including aesthetics and mathematical accuracy. In the LaTeX picture environment, locations are specified in terms of the current LaTeX `unitlength`, measured from the lower left corner of the LaTeX picture box. There are at least three unfortunate consequences of this design. First, the position of each object must be computed by hand in a coordinate system that usually has little to do with the figure. Second, while a figure can be scaled easily by changing the `unitlength`, it is impossible to change the aspect ratio of a figure without manually changing the positions of each object. Third, raw scaling can easily break a figure, because text does not scale. Thus even changing the size of a LaTeX figure can be non-trivial. `ePiX` circumvents these problems; this is discussed at more length in Section 3.

In order to draw a figure in `ePiX`, you must provide the following pieces of information: the `unitlength` in the figure, the size of the printed figure in picture coordinates, and the Cartesian coordinates of the figure's bounding box. `ePiX` performs the following affine scaling automatically:



Cartesian bounding box LaTeX picture box

Changing the size, aspect ratio, or bounding box of a figure is therefore trivial, and if the figure is well-designed it will scale attractively. Objects may extend outside the bounding box, but you generally have better control over figures that exactly fill their bounding box. The entire figure may be offset, as in the LaTeX picture environment; if unspecified the offset is zero. In contrast to LaTeX, positive offsets in `ePiX` shift the figure up and right.

2.1 A Simple Figure and `C` Tutorial

Suppose we want to depict a rectangle inscribed in the upper unit half-disk, say for an area maximization problem. The figure will contain coordinate axes, the semi-circle, the inscribed rectangle, and labels. The first step is to decide on the bounding box. In this example, $[-1, 1] \times [0, 1]$ is a reasonable choice. If the figure is to have true aspect ratio, the width will be twice the height. Next we decide the true size of the figure (as mentioned already, it is easy to change this later). A width of 2.5 in is reasonable, and the aspect ratio forces the height to be 1.25 in.

The input file is shown in Figure 1. The format should be partially self-explanatory, but a line-by-line explanation is given below. A few things may not be self-explanatory: The first line of input, delimited by "`/*`" and "`*/`", is a comment. (A comment begun with `/*` may span several lines, but ends at the *next* occurrance of `*/`. You should take care when commenting a large portion of an existing file, lest the new comment be terminated prematurely by an existing comment.) A single-line comment, analogous to a LaTeX line beginning with `%`, begins with "`//`". `ePiX` manipulates points and vectors using an ordered pair data structure; the construct `P(a,b)` creates the ordered pair (a, b). Finally, variables in `C` must have a declared type, such as `int` (integer), `double` (double-precision float), or `char` (character).

When `ePiX` is run on this source file and the output is included in a LaTeX document, the result is as depicted in Figure 2.

The `include` line ensures that standard commands will be available to `ePiX` when the file is compiled. Every `ePiX` file must contain this line. The double quotes are *single characters*, not pairs of single quotes. For the most

```
/* semicirc.c -- A rectangle inscribed in a semi-circle */

#include "epix.h"

double w = 0.6;

main()
{
  unitlength("1in");
  bounding_box(P(-1, 0), P(1, 1));
  picture(P(2.5, 1.25));
  offset(P(0,0.125));

  begin();

  line(P(x_min, 0), P(x_max, 0));
  line(P(0, y_min), P(0, y_max));

  ellipse_top(P(0,0), P(1,1));

  boldrect(P(-w,0), P(w, sqrt(1-w*w)));

  label(P(0.5, sqrt(0.75)), P(2,4), "$y=\\sqrt{1-x^2}$");
  h_axis_labels(P(x_min,0), P(x_max,0), x_size, P(-12, -12));

  end();
}
```

Figure 1: The source file for Figure 2.

part, `C` is not picky about spaces and blank lines in a file, but it is *very* sensitive to punctuation. Most lines in a `C` program end with a semicolon. The `include` line is a rare exception.

The next non-blank line is an *assignment statement.* In this example, `w` denotes the half-width of the inscribed rectangle. The final appearance of the figure can be adjusted by changing `w`. Using variables highlights the logical structure of the figure, which is especially valuable when the figure contains many objects whose positions are mathematically related. Generally, variables are similar to LaTeX macros; if a constant appears repeatedly in a single figure, it should probably be a variable. Multiple variable assignments may be placed on a single line, separated by commas, or may be put on separate

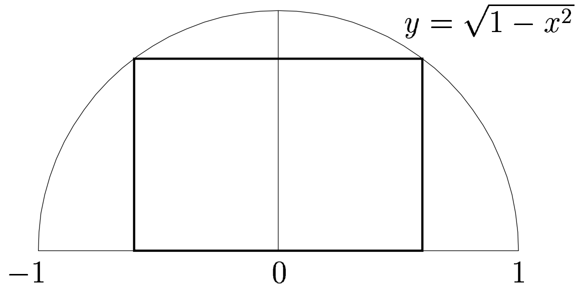


Figure 2: A rectangle inscribed in the upper half disk.

lines. It is a good idea to group together logically related assignment statements, and to use blank lines and spaces as needed to make the file easy to read.

The `include` line, variable assignments, and function definitions (none in this example) constitute the *preamble.* The action begins with the *function call* to `main`. The rest of the input file, the *body*, consists of `ePiX` commands.

The first four lines of the body assign values to several variables that determine the size and positioning of the figure. You are discouraged from accessing these variables directly, but their names are:

```
x_min       x_max       y_min       y_max
h_size      v_size      h_offset    v_offset
pic_size    pic_unit
```

Other than `pic_unit`, which is of type `char`,[2] all of these variables are `double`s.

The `begin();` line prints some commentary and a LaTeX picture head in the output file; everything between `begin();` and `end();` generates a picture object.

Picture object commands are mnemonic. The `line` commands draw the horizontal and vertical axes. The endpoints of the axes are specified in terms of the Cartesian bounding box of the figure; if the bounding box is changed, the axes will adjust automatically. The `ellipse_top` command draws the top half of an ellipse, centered at $(0, 0)$, with radius $(1, 1)$. Remember that the construct `P(a,b)` creates the ordered pair (a, b). The `boldrect` com-

[2]To be accurate, it's a "pointer" to `char`. You can use `ePiX` without ever knowing what this remark means.

mand draws a boldface rectangle, with opposite corners given in terms of the variable `w`.

Finally, there are two commands that print labels in the figure. The first causes output of a LaTeX command to be placed at the Cartesian location $(\frac{1}{2}, \frac{\sqrt{3}}{2})$, offset right by `2pt` and up by `4pt`. In the final figure, the label reads "$y = \sqrt{1-x^2}$", and is placed using the LaTeX basepoint of the entire formula. In `ePiX`, the correct way to position a label is to specify the "coarse" location in Cartesian coordinates, then to fine tune the location visually with the label offset. Label offsets are always specified in true points, because font sizes are given in points and do not change with scaling. The label command

```
  h_axis_labels(P(x_min,0), P(x_max,0), x_size, P(-12, -12));
```

generates horizontal-axis labels. The first label is at $(-1, 0)$, the last is at $(1, 0)$, and there are $2 + 1$ labels, namely one at each integer point. The final pair is the label offset; the labels will be shifted left by `12pt` (which roughly centers them) and down `12pt` (which prints them below the horizontal axis). `ePiX` generates the label values automatically.

There are a few important features (of both `C` and `ePiX`) illustrated by this input file.

- Label commands (and no others) have an offset option; all other objects are placed using Cartesian coordinates.

- In a label command, the label itself is generated by a string enclosed by double quotes. Aside from backslashes, the material between the quotes is printed verbatim to the output file; to get a backslash in the output, you must put a *double* backslash in the input.

- Global variables must be declared either in the preamble, or in `main` *before* anything else. Variables that are constant should be defined in the preamble; variables that depend on function values (such as `sqrt(3)/2`) or on other variables (e.g., `4*x_size`) must be defined inside `main`.[3]

- `C` knows several mathematical functions (`sqrt`, `log`, `sin`, and so forth) by the same names as LaTeX. However, `C` requires a `*` to denote multiplication, and does *not* recognize the caret as notation for exponen-

[3]To *declare* a variable is to tell the compiler the variable's name and type, but not to assign a value; the latter is to *define* the variable. A variable must be declared exactly once, but can be defined several times.

tiation. To get powers of a variable, you must either specify explicit multiplication, as in `w*w`, or must use `C`'s exponentiation function, as in `pow(w,2)`. The latter sensibly handles arbitrary floating-point bases and exponents.

- Almost all `ePiX` objects can be drawn in any of four styles: plain, dashed, dotted, and bold. The naming convention is that the "plain" object has a root name (such as `rect`), and the other styles are gotten by prefixing (e.g., `dashrect`, `dotrect`, `boldrect`).

`ePiX` provides about two dozen graphics primitives, including lines, arrows, triangles and rectangles, whole and half ellipses, circular arcs and arrows, quadratic and cubic splines, Cartesian and polar coordinate grids, axes with tick marks, and various types of point marker. These are descriptively named, and are invoked as in the example above. A detailed list and description is given in Section 3.

If you have the `pstcol` (or `color`) package for LaTeX, you can create basic color figures by delimiting portions of the input file with (say) `red();` and `end_red();` Colors available in this way are red, blue, green, magenta, cyan, and yellow. `ePiX` allows generation of essentially arbitrary colors. Detailed instructions on creating and previewing color files are given in Section 2.2.

Plotting Functions

`ePiX` has several commands for creating plots of user-specified functions. Suppose we wish to graph the sin function on the interval $[-3,3]$, emphasizing that while not invertible, it *is* invertible on the interval $[-\pi/2, \pi/2]$. Because the sin function is provided in `C`, we need not define it. Our input file might look like Figure 3. The resulting LaTeX output is Figure 4.

The figure is `300pt` wide and `100pt` high, and uses the constant `M_PI_2`. The axis-generating commands are new, but the syntax is easily divined; note that their positions (and the number of tick marks) are specified completely in terms of the bounding box. This works well as long as the bounding box has integer coordinates. The `plot` commands are of primary interest:

```
plot(sin, x_min, -x0, 30);
```

draws a plot of the sin function between `x_min` and $-\pi/2$, using 31 data points. The other `plot` lines are similar.

```
/* sine.c -- The graph y = sin x on [-3,3] */
#include "epix.h"
double x0 = M_PI_2; // \pi/2 to 20 decimals

main()
{
  unitlength("1pt");
  picture(P(300, 100));
  bounding_box(P(-3, -1), P(3,1));

  begin();

  h_axis(P(x_min, 0), P(x_max, 0), x_size);
  v_axis(P(0, y_min), P(0, y_max), y_size);

  plot(sin, x_min, -x0, 30);
  plot(sin, x0, x_max, 30);

  dashline(P(x0,0), P(x0, sin(x0)));

  label(P(x0, 0), P(-3, -12), "$\\frac{\\pi}{2}$");
  h_axis_masklabels(P(x_min,0), P(x_max,0), x_size, P(-14,-12));

  boldplot(sin, -x0, x0, 60);

  end();
}
```

Figure 3: The source file for Figure 4.

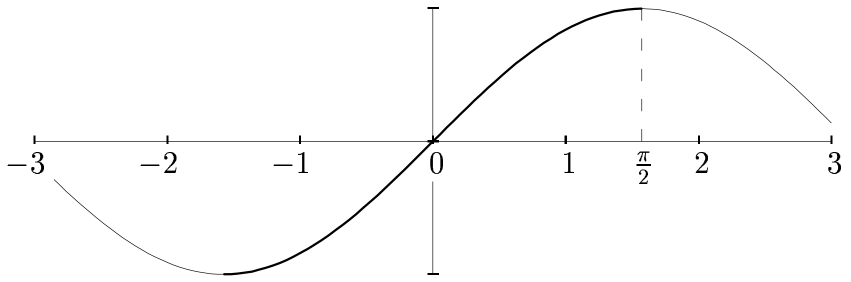


Figure 4: Plotting the sin function with `ePiX`.

An `ePiX` figure is built in layers, with later parts of the file set atop earlier parts. Generally you needn't pay much attention to the order in which objects appear in a source file, but in the sin graph example there are two opaque layers whose occlusion of lower layers is deliberate. First, the axis labels are "masked", meaning they are set atop opaque white rectangles; this is done to prevent the graph from colliding with the label at -3. Second, the bold portion of the graph passes over the mask at the origin. Consequently, the plain sin graph, then the axis labels, then the bold sin graph must appear in order in the source file. Layering tends to be more of an issue with color figures, since it is more apparent which of two curves passes over the other when their colors are different. Generally, masked axis labels should come last in a file, since the purpose of masking is to make the labels legible.

The number of points in a function plot should be as small as possible without compromising quality, an aesthetic decision. LaTeX reads in all the points in a path before typesetting anything, and is likely to run out of memory if the path contains more than about 700 points. There may be other limits on the total number of points a previewer is able to display.

`ePiX` provides plotting commands that do clipped, polar, or parametric plotting, draw slope or vector fields, or plot solutions to systems of ODEs, but they all operate similarly to `plot`: You provide the function(s) to be plotted, a range of values over which to plot, and the number of sample points.

Eventually, you will want to define your own functions for plotting. This is described in Section 3. Function (and variable) names may consist of letters, numerals, and underscores, and may not begin with a numeral. Names are case sensitive; the convention is to use lower-case names, but you need not do so. It's a good idea to use descriptive, relatively short names that are logically related to what they stand for. If the compiler complains that a function (or variable) is "multiply defined", it means the name has been previously declared.

2.2 Running `ePiX`

The zeroth step in using `ePiX` is to have a LaTeX document, say `sample.tex`, that requires a mathematically accurate figure. If you use LaTeX2e, your file should begin

```
\documentclass[12pt]{article}
\usepackage{amsmath,latexsym,epic,eepic}
```

The `epic` and `eepic` packages are the relevant ones here. If you plan to create color figures, you will also need `pstcol` or `color`, which are part of modern LaTeX2e. If you are using plain LaTeX, your file will start

```
\documentstyle[12pt,epic,eepic]{article}
```

and color is not available. Somewhere in `sample.tex` is the figure itself:

```
\begin{figure}[hbt]
\begin{center}
\input{tutorial-plot.eepic}
\caption{Plotting the sin function with \ePiX.}
\label{fig:example-plot}
\end{center}
\end{figure}
```

Before you run LaTeX, you need a file named `tutorial-plot.eepic` in the current directory. This is where `ePiX` comes in. If you have created an input file called `sine.c`, as directed in Section 2.1, and if `ePiX` is correctly installed, then you type

```
epix sine.c tutorial-plot.eepic
```

After a short interval, during which some reassuring messages will be printed on the screen, you will get the prompt back and the `.eepic` file will be there. That's all there is to it. You may leave off the file extensions, and may even omit the name of the output file if you want it to be the same as the name of the input (`sine.eepic` in this example).

To process the LaTeX file, either run LaTeX as usual, or do

```
laps sample
```

which runs LaTeX on `sample.tex`, then uses `dvips` to convert the `dvi` to the Postscript file `sample.ps`. This is a convenient option if your document includes color figures. `laps` (for "LaTeX to Postscript") is a shell script included with `ePiX`.

`ePiX` comes with a sample document containing side-by-side comparison of over a dozen source files and their output. Section 3 is a complete description of `ePiX`'s features, and Section 5 describes troubleshooting. Good luck, and happy drawing!

3 Advanced Features

The introduction of Section 2.1 does not describe all of `ePiX`'s features, even ones that you may use frequently. You may also find that you need to know more of `C`, either to understand why a source file isn't compiling, or to achieve effects using more of the power of `C`. Further, if the installation does not go smoothly, you will need to know how `ePiX` is implemented in order to get it working. Finally, you may be curious, or a better programmer than the author, and may have improvements to make. This section describes all of `ePiX`'s features, some of the design rationale, and the implementation.

3.1 `ePiX`, and the `epic` and `eepic` Styles

The LaTeX `epic` and `eepic` styles define commands `\path`, `\dashline`, and `\dottedline`, that take a list of ordered pairs as argument and print the corresponding connect-the-dots path (or dashed/dotted path). These commands are used in a LaTeX picture environment just like any LaTeX object. Computers are good at generating lists of numbers. In `ePiX`, you write some lines such as

```
/* Comment: The squaring function */
double f(double t) {
  return t*t;
}

/* Graph f for t in [-1,1] using 20 steps */
plot(f, -1, 1, 20);
```

When ePiX is run, the plot command writes the following ePiX *stanza* to the output file:

```
%% plot:
\path(0,50)(5,40.5)(10,32)(15,24.5)(20,18)(25,12.5)(30,8)
  (35,4.5)(40,2)(45,0.5)(50,0)(55,0.5)(60,2)(65,4.5)(70,8)
  (75,12.5)(80,18)(85,24.5)(90,32)(95,40.5)(100,50)
%%
%%-----------------------------------------------------%%
```

The actual numbers depend on the Cartesian bounding box, and on h_size and v_size, the LaTeX dimensions of the figure; in this example, ePiX has generated a parabola that fits into a LaTeX picture 100 units wide and 50 high.

The .eepic file written by ePiX is human-readable (even formatted and commented), and it may be instructive to look at the output of a short figure. You may edit the output file by hand if necessary, though it is safer (and usually easier) to change the source file and re-run ePiX. Note that ePiX will overwrite an existing .eepic file if so directed. In summary, ePiX is a LaTeX pre-processor; it turns a human-friendly picture description into a marked-up list of numbers for LaTeX.

An ePiX figure is layered: Objects are drawn over objects that come earlier in the file. Most ePiX objects are transparent, but when using color it is noticeable whether a blue curve goes over or under a magenta one (say). There are only a few ePiX commands that explicitly cover objects under them, but in a complicated figure even the implicit "masking" effect bears consideration. Among the commands that *do* cover previous layers are the masklabel commands, which draw an opaque white rectangle under labels, and circ, which draws a white-filled circle 4 true pt in diameter.

3.2 Coordinates and Dimensions

LaTeX requires you to specify a unitlength, the picture size, and an offset, which determines the picture coordinates of the lower-left corner. Picture objects' locations are specified to LaTeX in picture coordinates. When composing a mathematical figure, however, it is generally easier to use Cartesian coordinates. Letting software handle this conversion not only frees you from worrying about (non-portable) picture coordinates, but also makes it easy to change the size of the figure, and to place the figure exactly where LaTeX has left space for it. If you've ever had to change the unit length of a figure (for example, because you are American and your coauthor is not), only to have the picture become horribly ugly, overlap the surrounding text, or otherwise break the document, you'll understand why automatic coordinate conversion is a useful feature.

There is one situation where Cartesian coordinates are not adequate, namely when you are doing visual editing to place a text label in a figure. A label's size is independent of both picture coordinates and Cartesian coordinates, so if you use either to position a label, changing the scale or the Cartesian bounding box will probably break the positioning. However, *true coordinates* (measured on the page) are no good, either, because the size and/or scale of the picture may change with further editing. `ePiX` circumvents this difficulty by allowing labels to be placed in Cartesian coordinates, but to be offset in true coordinates. For instance, the labels on a coordinate axis need to be positioned at the proper location in the Cartesian plane, but because LaTeX uses the basepoint to position a box, the raw Cartesian coordinates are likely to mis-position the labels. If you fine-tune the labels' positions in Cartesian coordinates, however, the picture will break if the scale is changed. Using both Cartesian and true coordinates allows you to position labels easily so that they are placed correctly if the `\unitlength` is changed, or even if the Cartesian bounding box is changed.

Aside from labels, all positions in an `ePiX` file are specified in Cartesian coordinates. `ePiX` asks LaTeX to set aside a box of size `h_size`×`v_size`, then does affine scaling on the rectangle `[x_min, x_max]`×`[y_min, y_max]` to get an exact fit. You can still force objects to go outside the bounding box, by giving them Cartesian coordinates outside the box. Positive offsets in `ePiX` shift the picture **up** and **right** (the opposite of LaTeX picture offsets). `ePiX`'s output is accurate to 10^{-4}`*unitlength`.

3.3 More about `C`

An `ePiX` input file is really source code for a `C` program that writes an `.eepic` file as output. If you do not speak `C`, the main things to remember (principally differences between LaTeX and `C` syntaxes) are:

1. Comments, which may span several lines, are delimited by the strings `/*` and `*/`. One-line comments, similar to LaTeX comments, are begun with `//` (analogous to the LaTeX `%`) and end with the next newline. A `//`-style comment may appear within a multiline comment, but a `/* */`

comment may not; the `C` compiler will mistake the first `*/` it encounters as the end of the current multiline comment.

2. Every statement and function call must end with a semicolon. If you omit a semicolon, the compiler will give you a cryptic error message (such as 'parse error at line N' if the semicolon is missing on the last non-blank line before line N).

   Lines that begin with `#include` or `#define` are `C` *pre-processor* directives, and **do not** end with a semicolon.

3. As in LaTeX, the backslash `\` is an escape character in `C`. To print a `\` from `C`, you must use `\\`, as in

```
/* Put label $y=\sin x$ at (2,1) */
/* Note single  ^ backslash in output */
label(P(2,1), P(0,0), "$y=\\sin x$");
/*       Double backslash ^^ in source */
```

4. Variables in `C` must have a declared *type*, such as `int` (integer), `double` (double-precision floating point), or `char` (string of characters), and should be defined in the preamble or at the beginning of `main`. (Variables local to a function must be defined at the beginning of the function.) Variable and function names may contain letters (including underscore) and digits, are case sensitive, and must begin with a letter.

5. `C` requires explicit use of `*` to denote multiplication; simple juxtaposition is not enough. `C` does not support the use of `^` for exponentiation, e.g., `t^2` is invalid. Instead, you must use `t*t` or `pow(t,2)`.

6. The following words are reserved in `C`, and may not be used as function or variable names:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

In addition, `ePiX` reserves the following variables:

`x_min`	`x_max`	`y_min`	`y_max`
`h_size`	`v_size`	`h_offset`	`v_offset`
`x_size`	`y_size`	`pic_size`	`pic_unit`

`C` is case-sensitive, so capitalized variants are valid (but discouraged).

The aspect ratio of the figure is controlled by adjusting the LaTeX size of the figure and its Cartesian bounding box. The following length units may be used: `cm`, `in`, `mm`, `pc`, and `pt`. (One pica equals 12 pt.) Font-dependent units `em` and `ex`, and "exotic" units (Didot points, Ciceros, etc.) are not recognized. See `lengths.cc` to add support for other length units.

Aside from `#include` and `#define` statements, and variable declarations, the preamble consists of function definitions, for plotting functions you have specified. Functions may not be defined inside other functions; in particular, all definitions of user-specified functions *must* come in the preamble, before the call to `main`.

`C` knows the following functions of one variable by name:

`sqrt`	`sin`	`sinh`	`asin`
`ceil`	`cos`	`cosh`	`acos`
`floor`	`tan`	`tanh`	`atan`
`exp`	`log`	`log10`	`fabs` (abs val)

The inverse trig functions are principle branches. In addition, `pow(x,y)` returns x^y when real, and `atan2(y,x)` returns $\mathrm{Arg}(x + iy)$, the principle branch of arg. `C` knows many constants to 20 decimal places (such as `M_PI`, `M_PI_2`, and `M_E` for π, $\pi/2$, and e respectively; there are over a dozen in all). `ePiX` defines a few additional functions, such as `recip` (the reciprocal, defined to be 0 at 0) and `cb` (for "Charlie Brown"), the period-2 extension of the absolute value function on $[-1, 1]$. If you use certain functions frequently, add them to `functions.cc` and recompile `ePiX`.

The GNU `C` library defines many other functions, including inverse hyperbolic functions, log and exp with base 2, 10, or arbitrary b, the error and gamma functions, and Bessel functions of first and second kind.

You may use known functions in subsequent definitions. Functions of two (or more) variables are defined in direct analogy to functions of one variable:

```
double f(double t)
{
  return t*t*log(t*t); // t^2 \ln(t^2)
}

double g(double s, double t)
{
  return exp(2*s)*cosh(sin(M_PI*t));
}
```

3.4 More Advanced Uses of C

Because C is a programming language, ePiX figures can be specified using variables, loops, and recursion, to depict lots of interesting effects. This section presents a few ideas by example. The sample files contain other examples.

Piecewise-Defined Functions

The syntax for defining a function such as $f = \max(\sin, \cos)$ is:

```
double f(double t)
{
  if (cos(t) <= sin(t))
    return sin(t);
  else
    return cos(t);
}
```

If there are more than two formulas in the definition, use an else if construction, as in

```
double max_10_f(double x)
{
  if ( fabs(f(x)) < 10 ) // |f(x)| < 10
    return f(x);
  else if ( f(x)>0 )
    return 10;
  else
    return -10;
}
```

This truncates f at the lines $y = \pm 10$. (The function f must be defined separately.)

Finite Sums and Other Algorithms

Function definitions in `C` may be as simple as an algebraic formula, or as complicated as an arbitrary finite "rule". Sums of finite series are achieved with the following sort of definition, which defines $\text{fourier9}(t) = \sum_{k=1}^{9} \frac{1}{k} \sin kt$:

```
double fourier9(double t)
{
  int k; // summation index
  double y=0; // running total of the sum

  /* Run from k=1 to 9; increment k at each step */
  for (k=1; k < 10; ++k)

    // Add (1/k)sin(kt) to the running total
    y += (1.0/k)*sin(k*t);

  return y;
}
```

The Weierstrass non-differentiable function `sample` file is an example.

Generally, a function definition may be specified by an arbitrary algorithm; see the definitions of gcd, sup, and inf in `functions.cc` for more examples. Of course, numerical error can become an issue if the algorithm is complicated.

Parametrized Figures and Animation Frames

As already mentioned, variables may be used to clarify the logical structure of a figure. The way the inscribed rectangle figure was implemented makes it easy to change the width of the rectangle. When a figure is complicated, judicious use of variables makes the figure flexible. The `sample` file depicting upper rectangles for the sin function is an example; to change the number of rectangles, one need change only a single number in the preamble. To change the integrand to a different increasing function also requires just one change in the preamble. (Labels' text must still be changed individually.)

If a figure depends suitably upon a collection of parameters, then a loop can be used to draw the entire figure for multiple values of the parameters, yielding successive "snapshots" of the figure as time progresses.[4] In the `sample` files is a simple example, the cycloids traced by a rolling wheel. Other possibilities are to solve a planar ODE for varying lengths of time (illustrating the flow), or to plot solutions of a $(1+1)$-dimensional PDE (for example, to depict heat flow or wave motion).

3.5 Summary of `ePiX` Objects

After the `begin` line, an `ePiX` file consists of a sequence of style declarations (for color and boldface plots) and commands that draw the following kinds of objects:

- Polygons
  - Lines
  - Triangles
  - Rectangles parallel to coordinate axes
  - Arrows
- Coordinate axes and decorations
  - Axes with tick marks
  - Axis labels
  - Cartesian and polar coordinate grids
  - Empty and filled circles
  - Other labels (arbitrary text)
- Curve primitives
  - Ellipses and half ellipses
  - Circular arcs and arrows, hyperbolic lines

[4] LaTeX provides no direct way to animate frames, but the author has used a toy zeotrope together with `ePiX`-generated figures to depict animated mechanical configurations.

  - Quadratic splines
  - Cubic splines
  - Simple knot diagram primitives

- Plots of user-defined functions
  - Ordinary graph plots
  - Adaptive plots
  - Parametric and polar plots
  - Truncated graph and parametric plots
  - Space curves
  - Planar mesh plots
  - Plots of numerical data from files

- Calculus operations on user-defined functions
  - Derivative plots
  - Tangent field plots (along a parametrized path)
  - Definite integral plots
  - Slope fields
  - Vector fields
  - Solution curves to planar ODEs

4 Reference Manual

In addition to the ordinary (plain) style, all curve primitives and plots can be drawn in dashed, dotted, or bold styles, by prepending `dash`, `dot`, or `bold` to the object's name. Dotted and dashed graphs may look better when plotted adaptively. Bold dashed/dotted plots are obtained by constructions such as

```
bold();
dashline(P(0,0), P(2,1));
end_bold();
```

All input coordinates are Cartesian, and are of type double.

Pair manipulation (`pairs.cc`)

The `ePiX` function `P` turns a pair of `doubles` into a `<pair>` (an ordered pair data structure). `ePiX` treats pairs as complex numbers for arithmetic purposes; standard `C`-like constructions may be used with pairs.

- Pair creation, angles in radians:

  ```
  polar(r, theta);     cis(t);
  ```

  `polar(1.0, t)`, `cis(t)`, and `P(cos(t),sin(t))` are equivalent. `ePiX` recognizes the standard basis `e_1` and `e_2`.

- Arithmetic and incrementation (p and q are pairs, k is double):

  ```
  p + q, p - q, k*p, p*q, p/q;
  p += q, p -= q, p *= k, p *= q; p /= q;
  ```

  When forming symbolic expressions involving pairs, scalars (doubles) must be collected together at left, vectors (pairs) at right. If results of an arithmetic expression are unexpected, use parentheses to force a particular association.

  `ePiX` provides the Euclidean dot product (think of Dirac's bra-ket notation), the componentwise product, and a 1/4-turn counterclockwise:

  ```
  (a,b)|(x,y) = ax+by
  (a,b)&(x,y) = (ax,by)
  J(a,b) = (-b,a)
  ```

  For example, `P(a,b)&e_2 = P(0,b)`.

Typographical commands (`output.cc`)

- Assignment of variables

  ```
  bounding_box(P(x_min, y_min), P(x_max, y_max));
  picture(P(h_size, v_size));
  offset(P(h_offset, v_offset));
  ```

```
unitlength("1pt");
begin();
```

In an actual file, numbers are used, but the variables listed above are assigned values when these functions are called. `offset` is zero by default, hence optional. In `unitlength` the quoted argument is a floating point number followed by a two-letter LaTeX length. `begin` sets the variables `x_size` and `y_size`, the width and height of the bounding box.

- Line style:

  Entire sections of a figure may be drawn with LaTeX `thicklines` by delimiting the corresponding portion of the input file with `bold();` and `end_bold();` An `arrow` inside a `bold` environment is not the same as a `boldarrow`, but plot-type objects behave as expected in their `bold` versions.

- Color:

  `ePiX` provides color output via the `pstcol` or `color` packages, and can generate essentially arbitrary colors, using either the subtractive red-green-blue model (`rgb`, better for displaying) or the additive cyan-magenta-yellow-black model (`cmyk`, better for printing). An `rgb` color is determined by three floating-point densities between 0 (no color) and 1 (full saturation). Red, green, and blue are, respectively:

  ```
  rgb(1,0,0);    rgb(0,1,0);    rgb(0,0,1);
  ```

  A `cmyk` color is similarly specified by four floats. Densities outside the range $[0, 1]$ are "clipped". `ePiX` truncates color densities to 2 decimal places; if you need finer control, modify the functions `rgb` and `cmyk` in `outputs.cc`.

  These color specifiers are declarations, and remain in force until superceded. The scope of a color declaration is ended with a call to `black();` The six "elementary" colors can be called and ended by name, e.g., `red();` and `end_red();`

  Colors will not appear if the file is previewed in `xdvi`, but will appear if you use `laps` to process your LaTeX files, then view the Postscript with (say) `gv`.

- Perspective figures:

  ```
  viewpoint(viewpt1, viewpt2, viewpt3);
  V(x1, x2, x3);
  ```

  `ePiX` does orthogonal perspective drawing. The `viewpoint` of a figure is a vector onto whose orthogonal complement certain `plot` objects are projected. The overall scale of `viewpoint` is immaterial, e.g., $(1, 2, -1)$ and $(2, 4, -2)$ are identical, while $(-1, -2, 1)$ reverses the orientation of the figure. If `viewpoint` is not parallel to the z axis, then the z axis is drawn vertically; otherwise the figure is drawn in "standard" Cartesian form, regardless of a_3. By default, `viewpoint` is the zero vector, hence optional. The `viewpoint` may be changed anywhere in a source file. Using red and cyan together with slightly different viewpoints can be used to give a 3D-glasses effect.

  The function `V` produces a `pair` from an ordered triple. In polygons, arrows, splines, and ellipses, and in positioning labels, objects' location may be specified in terms of triples. There is a separate function for 3-d curve plotting, see below.

Polygons (`objects.cc`)

- The line joining (a, b) to (c, d):

  ```
  line(P(a,b), P(c,d));
  ```

- The triangle with vertices (a, b), (c, d), and (e, f):

  ```
  triangle(P(a,b), P(c,d), P(e,f));
  ```

- Coordinate rectangle with opposite corners (a, b) and (c, d):

  ```
  rect(P(a,b), P(c,d));
  ```

  Either pair of opposite vertices specifies the rectangle.

- Gray-filled rectangle (unaffected by color declarations), without or with boundary:

  ```
  swatch(P(a,b), P(c,d));
  boldswatch(P(a,b), P(c,d));
  ```

Coordinate axes and decorations (`objects.cc`)

- Axes with tick marks

  ```
      h_axis(P(a,b), P(c,d), n);
  ```

  Draws an axis between (a, b) and (c, d) with $n + 1$ evenly-spaced tick marks appropriate for a horizontal axis. If the bounding box has integer sides, then the command

  ```
      h_axis(P(x_min, 0), P(x_max, 0), x_size);
  ```

  draws a horizontal axis the width of the figure with tick marks spaced one Cartesian unit apart.

  ```
      v_axis(P(a,b), P(c,d), n);
  ```

  Same, but with tick marks appropriate for a vertical axis.

- Cartesian grid

  ```
      grid(P(a,b), P(c,d), n1, n2);
  ```

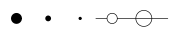
  Draws an `n1` by `n2` grid in the coordinate rectangle with opposite vertices (a, b) and (c, d).

- Polar grid

  ```
      polar_grid(r, n1, n2);
  ```

  Draws a polar grid of radius r with n1 rings and n2 sectors.

- S. D. Pedersen's `contrib/` package contains a number of routines for enhanced Cartesian graphs. See `contrib/doc` in the source for documentation.

- Point markers 

  ```
      spot(P(a,b));  dot(P(a,b));  ddot(P(a,b));
      circ(P(a,b));  ring(P(a,b), r);
  ```

  `spot`, `dot`, and `ddot` are black; `circ` is white filled, cannot be colored, and masks what is underneath. A `ring` has diameter `r` true pt, can be colored, and is transparent. `circ` is useful for denoting the end of an open interval. `ePiX` also provides a `marker` function, whose syntax is

  ```
    marker(P(a,b), <MARKER TYPE>);
  ```

○ `CIRC`	● `SPOT`	◯ `RING`	• `DOT`	· `DDOT`
+ `PLUS`	⊕ `OPLUS`	× `TIMES`	⊗ `OTIMES`	
◇ `DIAMOND`	△ `UP`	▽ `DOWN`	■ `BOX`	▪ `BBOX`

Table 1: `ePiX`'s marker types.

These marker types are also available when plotting data from a file, see `dataplot` below. Markers of type `SPOT`, `DOT`, and `DDOT` cannot be colored; this is a LaTeX issue.

- Arrows and darts

  ```
  arrow(P(a,b), P(c,d));    dart(P(a,b), P(c,d));
  ```

  Draws an arrow from (a, b) to (c, d) whose head is 3 true pt wide and 8.25 true pt long. If the true distance from (a, b) to (c, d) is less than 8.25 pt, only the arrowhead is drawn, with its base at (a, b). A `dart` is similar, but the head has half the linear dimensions. Dart and arrow dimensions are `#define`d in `globals.h`, and may only be changed at compile time.

- Axis labels

  ```
  h_axis_labels(P(a,b), P(c,d), n, P(u,v));
  v_axis_labels(P(a,b), P(c,d), n, P(u,v));
  h_axis_masklabels(P(a,b), P(c,d), n, P(u,v));
  v_axis_masklabels(P(a,b), P(c,d), n, P(u,v));
  ```

  `h_axis_labels` puts $(n+1)$ evenly-spaced labels on the segment joining (a, b) and (c, d). (Usually $b = d$, but this is not necessary.) The "mask" version draws an opaque white rectangle under the label text, and requires the `strcol` or `color` package. The labels are automatically generated to match their horizontal location. The pair `P(u,v)` gives the offset in true pt. For example,

  ```
  h_axis_labels(P(x_min,0), P(x_max,0), 8, P(-4,-12));
  ```

  divides the x axis into 8 subintervals, generates labels accordingly, and places them in Cartesian coordinates, but shifted left by 4 pt and down by 12 pt. The "v" versions are analogous, but for a vertical axis.

- Other labels

```
    label(P(a,b), P(u,v), "<text string>");
    masklabel(P(a,b), P(u,v), "<text string>");
```

  Prints `<text string>` with reference point at the Cartesian point (a, b), offset in true points by (u, v). The "mask" version puts an opaque white rectangle under the label text, and requires the `pstcol` or `color` package. To get a \ in output, there must be a \\ in the string. For example, the call

```
label(P(4,2), P(0,0), "$\\sigma=\\phi(\\tau)$");
```

  prints the line

```
\put(360,180){$\sigma=\phi(\tau)$}
```

  in the output file. (The `put` location depends on user-supplied dimensions.)

Curve primitives (`curves.cc`)

- Ellipses and half ellipses

```
    ellipse(P(a,b), P(u,v));
    native_ellipse(P(a,b), P(u,v));
    ellipse_top/bottom/left/right(P(a,b), P(u,v));
```

  Plots the (half) ellipse centered at (a, b) with radius (u, v), and with axes parallel to the coordinate axes. The "native" version uses an `eepic` macro, resulting in a shorter output file, and is available in plain only.

  Right half ellipse, rotated counterclockwise by θ *degrees.*

```
    ellipse_half(P(a,b), P(u,v), theta);
```

- Circular arcs

  Circular arc of center (a, b) and radius r, subtending the angle (counterclockwise, in radians) from θ_1 to θ_2.

```
    arc(P(a,b), r, theta1, theta2);
    arc_arrow(P(a,b), r, theta1, theta2);
```

If θ_2 is smaller, the arc goes clockwise. The arrowhead goes at θ_2. If an `arc_arrow` is too short, nothing is drawn.

Hyperbolic line in upper half-plane. No output if $b < 0$ or $d < 0$.

```
hyperbolic_line(P(a,b), P(c,d));
```

Hyperbolic line in unit disk. No output if either endpoint is outside the unit circle.

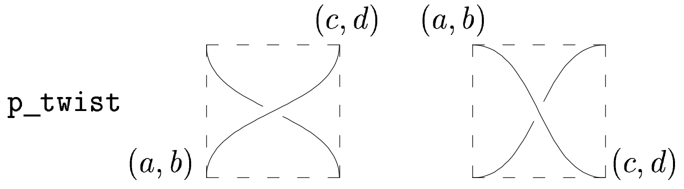
```
disk_line(P(a,b), P(c,d));
```

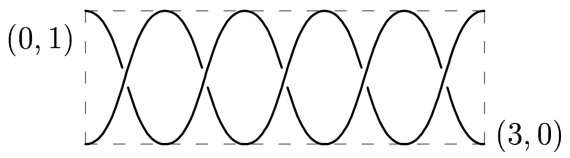
- Quadratic and cubic splines with specified control points

```
quad_spline(P(x1,y1), P(x2,y2), P(x3,y3));
cubic_spline(P(x1,y1), P(x2,y2), P(x3,y3), P(x4,y4));
```

- Simple knot diagram primitives (plain and bold only, shown with bounding boxes below)

```
p_twist(P(a,b), P(c,d));
n_twist(P(a,b), P(c,d));
twists(P(a,b), P(c,d), n);
```





`boldtwists(P(0,1), P(3,0), -5);`

Non-standard functions (`functions.cc`)

- Common functions with isolated discontinuities: `sgn` (signum), `recip` (reciprocal, defined to be 0 at 0), `sinx` ($\sin x/x$ with discontinuity removed)

- The "Charlie Brown" function, `cb`, namely the period-2 extension of absolute value on the interval $[-1, 1]$.

- Integer-valued functions `abs` and `gcd`.
- Non-printing utility functions: `sup(f, a, b);` `inf(f, a, b);`
  Numerical approximation of max/min value of f on $[a,b]$.

Plotting (`plots.cc`)

Functions must be defined in the preamble, before `main`, or in a separately compiled file, as in `functions.cc`. See the samples files for practical advice on usage.

- Ordinary graph plots

  ```
  plot(f, t_min, t_max, n);
  ```

  Plots the graph of f for input values between the limits, using $n+1$ points equally spaced in the domain. (`t_min` and `t_max` are doubles, `n` is an int)

- Adaptive plotting

  ```
  adplot(f, t_min, t_max, n);
  ```

  Same as `plot`, but attempts to space data points equidistantly along the graph. Good for dashed and dotted potting of differentiable functions whose derivative has large absolute value on small intervals, but is much slower than `plot`, and the output is not always more attractive.

- Parametric plots

  ```
  plot(f, g, t_min, t_max, n);
  ```

  Parametric plot of $\big(f(t), g(t)\big)$, with $n+1$ points. For backward compatibility, `paramplot` may also be used.

- Polar plots

  ```
  polarplot(f, t_min, t_max, n);
  ```

  Polar plot of $r = f(t)$ for $t \in$`[t_min,t_max]`, with $n+1$ points. Angles are measured in **revolutions**; $[0,1]$ is a full turn.

- Truncated plots

  ```
  clipplot(f, t_min, t_max, n);
  clipplot(f, g, t_min, t_max, n);
  ```

  Same as `plot`, but clips the plot to lie inside the bounding box

  `[x_min,x_max]` $\times$ `[y_min,y_max]`.

  Clipping is not currently supported in polar plots, but would be an easy feature to add.

- 2-D Mesh Plotting

  ```
  multiplot1(f1, f2, P(a,b), P(c,d), Net(n1, n2), num_pts);
  ```

  If f_1 and f_2 are functions of two variables, then (f_1, f_2) determines a mapping of the rectangle $[a, c] \times [b, d]$ to the plane. `multiplot1` divides this rectangle into an $n_1 \times n_2$ grid, then plots the images of the *vertical* segments (first variable held constant) using `num_pts` points per curve. `multiplot2` does the analogous plot when the second variable is held constant. In conjunction, these plot routines depict the image of the gridded rectangle under the stated mapping.

- Perspective plots

  ```
  plot(f1, f2, f3, t_min, t_max, n);
  ```

  Draws an orthogonal perspective plot of $\big(f_1(t), f_2(t), f_3(t)\big)$, with $n+1$ points. If `viewpoint` is not set, `plot` simply drops the third coordinate. See other notes on perspective drawing above. The alternative name `plot3d` may be used for backward compatibility.

- Data plotting from a file

  The format for a data file is two floating-point numbers per line; comment lines begin with a `%`, and improperly formatted lines are ignored.

  ```
  data_plot("filename", STYLE);
  ```

  Reads data in the named file and plots the corresponding points. In addition to the `marker` styles listed in Table 1 above, `STYLE` may be `PATH`, which connects the dots in the named file in the order they appear.

- Derivatives

  ```
  plot_deriv(f, a, b, n);
  ```

  Plots the derivative f' for $t \in [a, b]$, computed as a symmetric Newton quotient

  $$\frac{f(t+dt) - f(t-dt)}{2\,dt} \qquad dt = \frac{b-a}{2 * n * \texttt{ITERATIONS}},$$

  with `ITERATIONS` equal to 1000 by default.

- Tangent fields

  ```
  tan_field(f1, f2, t_min, t_max, n);
  ```

  Plots $n + 1$ tangent vectors to the corresponding parametric curve. Because tangents are rendered at true length, in total they form a rough piecewise-linear approximation of the curve, especially if n is large. The curve itself is not plotted.

- Integrals

  ```
  plot_int(f, a, b, n);
  ```

  Plots the definite integral $\int_a^x f(t)\,dt$ for $x \in [a, b]$. Uses the trapezoid rule for computing a running sum; the step size is determined by the constant ITERATIONS.

- Vector fields (plain and bold only)

  ```
   slope_field(f1, f2, P(a,b), P(c,d), n1, n2);
    dart_field(f1, f2, P(a,b), P(c,d), n1, n2);
  vector_field(f1, f2, P(a,b), P(c,d), n1, n2);
  ```

  Graphs the slope field (fixed length, no arrows), dart field (fixed length, small arrowheads) or vector field (true length, arrows) $F = (f_1, f_2)$ on the specified rectangle, with $(n_1 + 1) \times (n_2 + 1)$ sample mesh. Slope lines and darts are guaranteed not to overlap, while vectors are drawn at true length.

- Solutions of planar ODEs

  ```
  ode_plot(f1, f2, P(a,b), t_max, n);
  ```

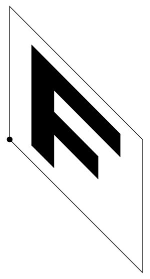
Uses Euler's method to solve a system of ODEs starting at (a, b), attempting to follow the solution for time $t_{\max}$ in $(n+1)$ steps. The plot terminates if it leaves the bounding box.

Arcane features (`arcana.cc`)

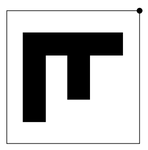
- Planar linear transformations

  `std_F(P(a1,b1), P(a2,b2));` draws the image of the unit square under the linear transformation corresponding to the matrix whose columns are (a_1, b_1) and (a_2, b_2). The parallelogram contains the image of the "standard F":

$$\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$



$$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$$



- Profiles of surfaces of revolution

```
plot_profile(f, a, b, n);
```

  Not intended for general use, but documented anyway. Suffice it to say that a "profile" plot solves a certain differential equation and plots the solution. If f is a quadratic polynomial, the resulting curve will be the profile of a surface of revolution of constant Gaussian curvature.

- Recursively-generated piecewise-linear fractal curves

  Consider a path made up of equal-length segments that can point at any angle of the form $2\pi k/n$, for $0 \le k < n$, like spokes on a wheel. The path is specified by a finite sequence of integers, taken modulo n. For example, if $n = 6$, then the sequence $0, 1, -1, 0$ corresponds to the ASCII path `_/\_`. `ePiX`'s fractal generation routine starts with such a "seed" then recursively (up to a specified depth D) replaces each segment with a scaled and rotated copy of the seed. The seed above generates the Koch fractal, for instance.

```
    fractal(P(a,b), P(c,d), D, seed);
```

`seed` is declared in the preamble as (e.g.)

```
    const int seed[] = {6, 4, 0, 1, -1, 0};
```

The first entry (here 6) is the number of "spokes" n, the second (4) is the number of terms in the seed, and the remaining entries are the seed proper. The final path joins (a, b) to (c, d). The number of segments in the final path grows exponentially in the depth, so depths larger than 5 or 6 are not likely to work: LaTeX may crash, or the machine may even run out of memory, depending on the length of the seed. At large depth, the output will probably suffer from round-off error even if the figure renders.

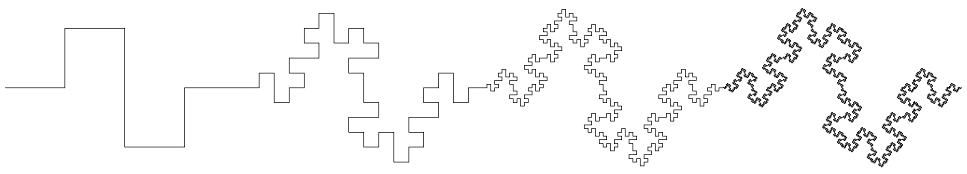


Figure 5: The fractal generated by {`4,8,0,1,0,3,3,0,1,0`}

5 Troubleshooting

5.1 Installation

Three files comprise `ePiX`: a shell script (`epix`), a header file (`epix.h`), and a compiled library (`libepix.a`). The utility script `laps` is part of the distribution, but independent from `ePiX` proper. These components are placed into standard locations where they can find each other. The likeliest problems with `ePiX` are that some component (usually `epix` itself) cannot find the other pieces, or that the shell cannot find `epix`.

Installation Problems

Public installation (in `/usr/local`) is the default. Unpack the tar file, then `cd` to the source directory (`epix-0.8.x`) and do

```
make test
make install
```

(the latter as `root`). The instructions for private installation (in `$HOME`) are almost identical; you need only change one line in the `Makefile` and one line in the shell script `epix`, and follow the directions in `POST-INSTALL`. The directory in which you install `ePiX` is denoted `$INSTALL`. After `make install`, you will have the following files:

```
    $INSTALL/bin/epix    $INSTALL/include/epix.h
    $INSTALL/bin/laps    $INSTALL/lib/libepix.a
```

Command not found In order to use `ePiX`, the directory `$INSTALL/bin` must be in your `PATH` (see `POST-INSTALL`); type

```
echo $PATH
```

to see your `PATH`. If the directory `$INSTALL/bin/` is not in your path, please read `POST-INSTALL` or ask someone knowledgeable at your site for help; the procedure varies depending on what shell you use. In any case, you will need to modify your shell's configuration file.

If you get a "`g++`: command not found" error, do

```
which g++
```

to see which compiler your system uses. In the shell script `epix`, modify the "compiler" line accordingly.

Bad substitution Another likely cause of trouble is that `bash` is not the default shell on your system. (This is mostly a problem for Unix users; GNU systems almost uniformly use `bash` as the default.) Do

```
ls -l /bin/sh
```

to find your system's default shell. If this is not `/bin/bash` (or something equivalent), you have a bit more work to do. First do

```
which bash
```

to find the path to `bash`, then modify the top line of each of the shell scripts, replacing `#!/bin/sh` with the path to `bash` on your system.

Permission denied This is very unlikely, but conceivable. For each component of the program, do a long listing, e.g.
`ls -l $INSTALL/bin/epix`
and so forth. The header and library must be readable, and the shell scripts and directories must be readable and executable. From the install directory, do

```
chmod 0755  bin  include  lib  bin/epix  bin/laps
chmod 0644  include/epix.h  lib/libepix.a
```

If you have installed `ePiX` in your home directory, use `0700` and `0600` instead.

If you still cannot get `ePiX` to run, please send email to the author, briefly describing the errors you are getting, the type of system you are on, and so forth.

5.2 LaTeX Errors

There are a few things that can cause LaTeX to stop with an error message when reading an `.eepic` file written by `ePiX`. The most common is the appearance of `nan` (not a number) where LaTeX expects a number. This generally indicates division by zero or bad exponentiation.

When a number is very small, `ePiX` may write it in exponential notation. If this happens, LaTeX will pause with an error message when it tries to read, e.g., `1.4142135e-14`. This bug in `ePiX` has been addressed; please send the author a bug report if you encounter this behavior. You can manually edit the `.eepic` file, replacing underflows with 0. In this eventuality, it's wise to rename the edited file, lest `ePiX` overwrite your changes the next time you run it.

Overflow errors are possible if a point has coordinates larger than 2^{16}; make sure you're not trying to plot the graph of a pole or something similar.

Occasionally `ePiX` will not generate any data points after it starts a path; this may signify a variable whose value is not what you expect (particularly in a loop), a badly defined function, or a bug in `ePiX`. If you cannot resolve the problem, and have narrowed down the issue to a small input file, please send the author a copy of the file and an explanation of the problem.

LaTeX has limited memory, and cannot plot arbitrarily many points. Make sure you did not accidentally append a digit to the number of points to be plotted; 500 data points should be fine, but 5000 will almost surely cause problems. If necessary, a long list of data points in a `path` can be broken into groups of a few hundred (see `fractal`, in `arcana.cc`), but even so LaTeX will eventually overflow. Don't expect to render a bitmap as an array of colored boxes in LaTeX. ⌣̈

One of the benefits of `.eepic` files is that they're plain text. `ePiX` comments its output, which should assist you in debugging. Each object or plot command in the input file generates a stanza in the output file that has a label saying which function wrote the stanza.

If a run of `ePiX` is taking unusually long (more than a few seconds), it's a good idea to kill the process by typing `ctrl-C` and to inspect the output for signs of an infinite loop. There is output size checking, but conceivably some infinite loop will be missed. On a fast PC, `ePiX` can easily write 15 MB of data per second, which will quickly fill up your disk and possibly damage other files. Do not run `ePiX` as `root` for normal usage!

6 Feedback and Political Blurb

Feedback about this program (bug reports, requests for features, etc.) is welcome. If you find `ePiX` useful, please tell your colleagues about it.

Academics in general, and mathematicians in particular, depend on Free software in their work. A good case can be made that proprietary software is contrary to the academic ethic. Issues of access aside, if one does not know what exactly went into a program, then one cannot fully trust the results that come out, any more than one can trust (for purposes of scientific publication) results of a commercial testing lab. Access to the source code is not all that is required, though. To promote the dissemination of information, access to software should be free in the four senses laid out in the GNU General Public License authored by the Free Software Foundation:

- To run a program for any purpose
- To study how the program works, and adapt it to your needs
- To redistribute copies of the program
- To improve the program, and release your improvements to the public, so that the whole community benefits

Just as theorems are not licensed, I believe that software we use in our academic work should not be licensed. Releasing software under a standard commercial license agreement is (to me) the equivalent of publishing the statement of a theorem, while keeping the proof secret, and charging people for each citation of the theorem. Releasing source code alone, without giving users the freedom to modify it for their own needs, is analogous to publishing a proof, but forbidding readers from using the ideas of the proof in their own work.

The ultimate purpose of software is to allow us to be productive and creative. I hope that this modest program is, in conjunction with the much larger efforts of others (especially Donald Knuth, and the many people who have contributed to the authorship of LaTeX and its many packages), useful to you in your mathematical work.

Please visit the Free Software Foundation, at `http://www.fsf.org`, to learn more about Free Software and how you can contribute to its development and adoption.

Andrew D. Hwang `<ahwang@mathcs.holycross.edu>`
Current version: 0.8.x (see `CHANGELOG` for details)
Last Change: June 6, 2002